



UNIVERSITÉ DE
SHERBROOKE

Faculté de génie
Département de génie électrique et de génie informatique

Un système de test pour systèmes répartis utilisant
le logiciel standard CORBA

A CORBA-based system for testing
distributed systems

Mémoire de maîtrise en sciences appliquées
Spécialité: génie informatique

Kianoosh AHMADY SIMAB

Sherbrooke (Québec), Canada

Mai 2002

SOMMAIRE

Le test est considéré comme une des étapes du cycle de vie d'un logiciel, et la dernière phase de la méthodologie de création de logiciel (analyse, conception, développement, et test). Dans ce mémoire, nous contribuons à la phase de test. Nous définissons les systèmes répartis et étudions les méthodes et les architectures pour tester un système réparti, à savoir : l'architecture de test centralisé, l'architecture de test réparti (ou distant), et l'architecture de test coordonné. Si l'architecture centralisée ne pose pas de problème particulier, l'architecture répartie cause plusieurs problèmes en terme de contrôlabilité et d'observabilité, qui sont des caractéristiques fondamentales du test de conformité. Après une présentation des problèmes de contrôlabilité et d'observabilité, nous proposons une solution à ces deux problèmes, qui consiste à utiliser une architecture de test coordonné. Ensuite, nous proposons et concevons une architecture de test coordonné constituée de trois parties : le contrôleur de test, le système de test, et l'implémentation sous test. Ensuite, nous présentons CORBA (Common Object Request Broker Architecture), qui s'occupe de la communication entre les trois parties de notre architecture de test. Nous présentons une implémentation en Java et CORBA de notre architecture de test. Et enfin, nous illustrons l'application de notre architecture pour le test d'une version temporisée du protocole X.25.

SUMMARY

Testing is considered as one of the steps in software life cycle and is the last phase in software creation methodology (Analysis, Design, Development, Testing). In this work, we contribute to *testing* phase. We define distributed systems, and study methods and architectures to test a distributed system, namely : centralized test architecture, distributed (or remote) test architecture, and coordinated test architecture. If the centralized architecture does not pose any particular problem, the distributed architecture raises several problems in terms of *controllability* and *observability*, which are fundamental features of conformance testing. After presenting controllability and observability problems, we propose a solution to these two problems, which consists of using a coordinated test architecture. Then, we propose and design a coordinated test architecture, consisting of three parts : Test Controller, Test System and Implementation Under Test. Then, we introduce CORBA (Common Object Request Broker Architecture), which is responsible for communications between the three parts of our test architecture. Then, we present an implementation in Java and CORBA of our test architecture. And finally, we illustrate the application of our architecture for testing a timed version of the X.25 protocol.

To My Dear Parents, Mahin and Shahpoor

ACKNOWLEDGEMENT

First of all, I thank God for giving me good health and patience to finish this work.

I would like to thank my supervisor, Professor Ahmed Khoumsi, for his patience, cooperation, encouragement and financial support during this work.

I would like to thank my sister Kamelia, who was always a big support to me. I also thank my friend Younouss Patel, who helped me a lot with great ideas at the beginning of this work.

Last but not least, I would like to thank my parents for their great love and support, especially my father who passed away during my studies.

I dedicate this work to my dear parents, Mahin and Shahpoor.

TABLE OF CONTENT

Chapter 1	Introduction	1-5
Chapter 2	Testing	6-15
	2.1 <i>Introduction</i>	
	2.2 <i>Methods for testing a component</i>	
	2.3 <i>Methods for testing a distributed system</i>	
	2.3.1 <i>Distributed systems</i>	
	2.3.2 <i>Test architectures</i>	
	2.4 <i>Model for describing IUT and its specifications</i>	
	2.5 <i>Fault model</i>	
	2.6 <i>Test sequence</i>	
Chapter 3	Controllability and Observability	16-24
	3.1 <i>Introduction</i>	
	3.2 <i>Controllability and observability</i>	
	3.3 <i>Solution to controllability and observability problems</i>	
Chapter 4	Proposed test architecture	25-46
	4.1 <i>Architecture</i>	
	4.1.1 <i>General architecture</i>	
	4.1.2 <i>Adding an interaction system</i>	
	4.1.3 <i>Constructing an IUT</i>	
	4.2 <i>Components</i>	
	4.2.1 <i>Test Controller</i>	
	4.2.2 <i>Test System</i>	
	4.2.3 <i>Interaction system</i>	
	4.2.4 <i>IUT</i>	
Chapter 5	CORBA and its event service	47-60
	5.1 <i>What is middleware?</i>	
	5.2 <i>What is CORBA?</i>	
	5.3 <i>The OMG Event Service</i>	
	5.3.1 <i>Introduction</i>	
	5.3.2 <i>Event Service Basics</i>	
	5.3.3 <i>Event Service Limitations</i>	
	5.4 <i>Event channel in realization</i>	
Chapter 6	Classes and interfaces used in the CORBA-based realization of the proposed test architecture	61-77
	6.1 <i>Classes of the Test Controller</i>	
	6.2 <i>Classes of the Test System</i>	
	6.3 <i>Classes of Interaction System (IS) and IUT</i>	

Chapter 7	Example of application	78-91
	<i>7.1 Simplified X.25</i>	
	7.1.1 Introduction to X.25 protocol	
	7.1.2 Simplified service provided by X.25 protocol	
	7.1.3 Formal specification of the simplified X.25 service	
	<i>7.2 IUT Construction</i>	
Chapter 8	Conclusion and future work	92-94
	<i>8.1 Conclusion</i>	
	<i>8.2 Future work</i>	
References		95-96
Appendix A	Requirements and Instructions to run the program	97-105
	<i>1 Requirements</i>	
	<i>2 Instructions</i>	
Appendix B	Algorithm to obtain coordinated local test sequences from global test sequence	106-109

LIST OF FIGURES

Figure 2.1	Representing of a distributed system	8
Figure 2.2	Different architectures for testing a distributed system	10
Figure 2.3	Example of a 2p-FSM	12
Figure 2.4	The test sequence	14
Figure 3.1	Specifications of an IUT	17
Figure 3.2	Faulty implementation	18
Figure 3.3	Projection of a global test sequence	19
Figure 3.4	Specifications	20
Figure 3.5	Faulty Implementation	20
Figure 3.6	Specifications	21
Figure 3.7	faulty Implementation	21
Figure 3.8	The coordinated test architecture	22
Figure 3.9	The coordinated local test sequences for three ports	24
Figure 4.1	proposed test architecture	26
Figure 4.2	Improved proposed test architecture	27
Figure 4.3	Architecture of our example	28
Figure 4.4	The general text file	31
Figure 4.5	Service and protocol	31
Figure 4.6	The local text files	32-33
Figure 4.7	The port table for port 1	34
Figure 4.8	Constructed IUT after taking the steps a to e	34
Figure 4.9	The components of Test Controller	35
Figure 4.10	Test System	36
Figure 4.11	Components of a tester	38
Figure 4.12	Text files between an agent and a port	40
Figure 4.13	An agent	41
Figure 4.14	The components of an Agent	42

Figure 4.15	Represents text file (port1.atm) for port number 1 in	43
Figure 4.16	Represents PortTable for port 1 in example 3.1	43
Figure 4.17	Displays the two states and the transition in the first line of table	44
Figure 4.18	Components of a port in IUT	46
Figure 5.1	Push-style event delivery model	50
Figure 5.2	Pull-style event delivery model	50
Figure 5.3	canonical push model	51
Figure 5.4	canonical pull model	52
Figure 5.5	Hybrid push/pull model	52
Figure 5.6	Hybrid pull/push model	53
Figure 5.7	Mixing event delivery model	54
Figure 5.8	Representing the event channels	56
Figure 6.1	The structure of folders and files of realisation program	62
Figure 6.2	User Interface displayed by running User.java	63
Figure 6.3	User Interface displayed by running Tester.java	66
Figure 6.4	User Interface displayed by running Agent.java	70
Figure 6.5	User interface by running Port.java	73
Figure 7.1	Specification of $S_{i,j}$, i.e., service specification of the simplified X.25, where $Site_i$ is the sender and $Site_j$ is the receiver	80
Figure 7.2	Global service specification of X.25	80
Figure 7.3	The complete FSM for both blocks	82
Figure 7.4	The global text file	84
Figure 7.5	The local text files	85
Figure 7.6	The port tables obtained in step d	86
Figure 7.7	Represents the specific transitions in GTS chosen by user	88
Figure 7.8	The flow of data	91
Figure 8.1	Transition	93
Figure 8.2	Transition	93

LIST OF ABBREVIATIONS

- **CORBA**: Common Object Request Broker Architecture
- **CSCI**: Computer Software Configuration Item
- **FIFO**: First In First Out
- **GTS**: Global Test Sequence
- **HWCI**: Hardware Configuration Item
- **I/O FSM**: Input/Output Finite State Machine
- **IS**: Interaction System
- **IUT**: Implementation Under Test
- **Java RMI**: Java Remote Method Invocation
- **LTS**: Local Test Sequence (from chapter 4 page 25, LTS means: coordinated local test sequence)
- ***np*-FSM**: multi-port FSM with *n* ports
- **OMG**: Object Management Group
- **POC**: Points of Observation and Control
- **PTR**: Document and maintain the Problem Tracking Reports
- **SDF** : Software Development File
- **TC**: Test Controller
- **TS**: Test System

CHAPTER 1

INTRODUCTION

In this report, we will introduce distributed systems and then we will design a *distributed test system*, which is able to test distributed systems. We will discuss the specifications of distributed systems and different architectures of a distributed test system. Since *testing* is a step of software life cycle, so first we remind the software creation methodology steps [1], [13]. This contains four phases:

- 1- Analysis
- 2- Design
- 3- Development
- 4- Testing

Phase I -- Analysis

This phase entails the review and finalizing of the user requirements. A functional overview documentation template will also be prepared at the close of this phase. Upon approval by project and user management, this template will be the defining medium for the next phase, General Design.

Phase II -- Design

This phase consists of two sub-phases: General Design and Detailed Design

General Design

This phase involves the logical design of the Application system. In this phase the following activities will be carried out:

- functional overviews will be prepared for each business function,
- all the data attributes related to screens, reports, and process flows will be included with the functional overviews,
- logical data models will be finalized and table definitions prepared for review,

- program specifications and standards templates,
- screen and report layouts,
- program decompositions.

Detailed Design

The Detail Design Phase facilitates the physical design of the system. The attention will be around the following objects, which are already identified, in earlier phases:

- User requirements
- Functions and Data supporting the User requirements
- Processes to be implemented to carry out the Functions and maintain the Data.

In this phase the details on 'how' the system should function are finalized. The technical architecture is drawn up, taking into consideration the application development and the operating environment.

Phase III -- Development

This phase deals with the coding, unit testing and documentation of the programs. The programmers develop the program units, using the templates, as per the respective program specifications. These programs are tested at the unit level, and at the module level.

Phase IV – Testing

This phase consists of:

unit integration and testing

The developer shall establish test cases (in terms of inputs, expected results, and evaluation criteria), test procedures, and test data for conducting unit integration and testing. The test cases shall cover all aspects of the CSCI-wide and CSCI architectural design. The developer shall record this information in the appropriate software development files (SDFs).

CSCI(Computer Software Configuration Item) qualification testing

The developer shall define and record the test preparations, test cases, and test procedures to be used for CSCI qualification testing and the traceability between the test cases and the CSCI requirements. The developer shall prepare the test data needed to carry out the test cases and provide the acquirer advance notice of the time and location of CSCI qualification testing.

CSCI/HWCI (Computer Software Configuration Item/Hardware Configuration Item) testing

The developer shall participate in developing and recording test cases (in terms of inputs, expected results, and evaluation criteria), test procedures, and test data for conducting CSCI/HWCI integration and testing. The test cases shall cover all aspects of the system-wide and system architectural design. The developer shall record software-related information in appropriate software development files (SDFs).

System qualification testing

The developer shall participate in developing and recording the test preparations, test cases, and test procedures to be used for system qualification testing and the traceability between the test cases and the system requirements. The developer shall participate in preparing the test data needed to carry out the test cases and in providing the acquirer advance notice of the time and location of system qualification testing.

The purpose of the system testing is to ensure that the system is sound from the business perspective and to validate the limits of the system. What type of volume can the system handle? Are there any potential response time problems? Is there any potential deadlocking? The data used for this test should be as close to real data as possible. This phase also serves the System Testing for the Conversion Procedure. For effective testing, the testers have to simulate various different dates in the system.

Since we emphasize on testing in this report, so we walk through the testing procedure details.

System Testing Procedure

The following activities are carried out in System Testing phase:

- Develop integrated system test plan
- Define strategy for test environment
 - define general approach, objectives
 - define testing procedures
 - choose test data generator
- Establish the System Test Environment
- Identify external resources required for testing
- Develop test model
 - define test cycles
 - * List programs in cycle

- * List screens in cycle
- * List files in cycle
- * List reports in cycle
- create test scenarios
- define levels of security
- define test data required
- generate test data
- create expected results
- Finalize test model
 - review planned test environment
 - review test scripts
 - confirm test environment is in place
- Execute test cycles at each security level
 - test all manual and computer operating procedures
 - monitor performance
 - verify results
 - make changes, if required
 - retest
- Test recovery and restart
- Review test results
- Document and maintain the Problem Tracking Reports (PTR)
- Review the System Test results with the Users

In this report we will contribute to the fourth phase, which is *testing*.

The chapters are structured as follows. In chapter 2, there is a general speaking about testing, then different methods of testing a component are discussed, then an introduction to distributed systems and different architectures for testing a distributed system. In chapter 3, we introduce Controllability and Observability problems that arise with methods for testing distributed systems, and we show solution to resolve these problems. In chapter 4, we propose and study an

architecture, which consists of four parts: Test Controller (TC), Test System (TS), Interaction System (IS) and Implementation Under Test (IUT). In chapter 5, CORBA (Common Object Request Broker Architecture) is introduced and discussed in details. CORBA is a standard middleware, which allows a transparent communication between the parts of the architecture introduced in chapter 4. In chapter 6, we show how the test architecture of chapter 4 is realized by using CORBA. In chapter 7, we present an example of our architecture's application. In chapter 8, we conclude, mention a restriction of our realization and then we propose some future work. Appendix A contains instructions and requirements to run the program, and appendix B represents an algorithm related to chapter 3.

CHAPTER 2

TESTING

2.1 Introduction

After the implementation of a system, the best way to check if it works is to try it. This is called *testing* and it can be seen as the practical way to check the behavior of a system. One may ask why do we test? A simple answer might be: for detecting errors in implementation. But testing is seen as a process for demonstrating the conformance to a reference specification, e.g. protocol conformance testing. One may also consider testing as a way for the assessment of the correctness of an implementation. In fact, testing is a method of software verification that deduces from execution results or traces that the software under test possesses certain “good” properties. The intuitive meaning of test is the following, “a good test provides convincing evidence that an implementation is correct”[2].

Different kinds of testing are generally used depending on what kind of behavior one wants to check. Testing involves extracting information about a concrete system by means of experimentation. It consists of carrying out experiments according to a certain scenario, by a human being or by a machine, in a certain environment, with the intent of getting information about the system under test from the observations that can be made during the experiment.

In this chapter we will introduce testing methods and distributed systems and then will focus on conformance testing of distributed systems.

2.2 Methods for testing a software component

Existing software testing techniques are divided into two categories: static and dynamic testing.

Each of these methods contains two methods: [4]

Static: syntactic, semantic.

Dynamic: black-box, white-box.

Now we briefly explain these four methods.

Static testing techniques are concerned with the analysis and checking of system representations such as the requirements documents, design diagrams and the program source code, either manually or automatically, without actually executing the code. In comparison to dynamic testing, static testing does not require inputs, since they do not require that the software be executed. This is convenient when the inputs are not known. Static testing techniques can be classified according to whether or not the technique requires examining the syntax of the source code. If so, the technique is syntactic testing, if not, the technique is semantic testing. Syntactic testing may include the reviews and walk-through to check that the refinements of accepted requirements are proceeding as desired through each transformation. Semantic testing includes formal methods such as proof of correctness.

Dynamic testing techniques are generally divided into two categories, black-box and white-box testing, which correspond to two different starting points for software testing: the internal structure of the software (white-box) and the requirements specification (black-box). They involve the execution of a piece of software with test data and a comparison of the results with the expected output, which must satisfy the users' requirements. Black-box testing uses a 'toaster mentality': You plug it in, it is supposed to work. Created input data is designed to generate variation of outputs without regard to how the logic actually functions. The results (i.e., outputs) are predicted and compared to the actual results to determine the success of the test. In contrast to this, white-box testing opens up the 'box' and looks at the specific logic of the application to verify how it works. The black-box testing method is sometimes called "functional" or "specification-based" while white-box testing method may be referred to as "structural" or "code-based" or even "glass-box".

In this study, we deal with conformance testing (dynamic black-box), therefore "testing" means "conformance testing". The principle of conformance testing is to apply inputs to the implementation under test (IUT) and to compare the observed outputs to the expected outputs. In this way, a mismatch between them reveals that IUT is faulty. A set of inputs and the expected outputs is generally called a test case and is automatically or manually generated from the IUT specification. [3]

2.3 Methods for testing a distributed system

2.3.1 Distributed systems

A distributed processing system (or simply *distributed system*) is a system which can exploit a physical architecture consisting of multiple, autonomous processing elements that do not share memory but cooperate by sending messages over a communication network. The components of a distributed system may be located in different places, and communication between components may suffer delay or may fail.

Distributed systems offer several advantages in comparison to centralized systems such as the ability to share resources, to be dynamically extended with new ones and to potentially increase availability and performance. Typically, distributed systems are heterogeneous in terms of interconnected networks, operating systems and middleware platforms they are based on, as well as in terms of programming languages used to develop individual components.

The goal of *open* distributed systems is to enable access to components of a distributed system from anywhere in the distributed environment without concern of its heterogeneity. Openness includes openness to various levels of the distributed system architecture: communication network, middleware platform and application level. Openness requires the definition of interfaces of the components on the different levels of distributed systems and a notion of compliance to these interfaces in order to ensure compatibility, interoperability and portability.

Figure 2.1 represents a general distributed system.

A distributed system contains several distributed interfaces, called ports and also denoted PCO (Points of Control and Observation), and interacts with the environment through these ports. [10]

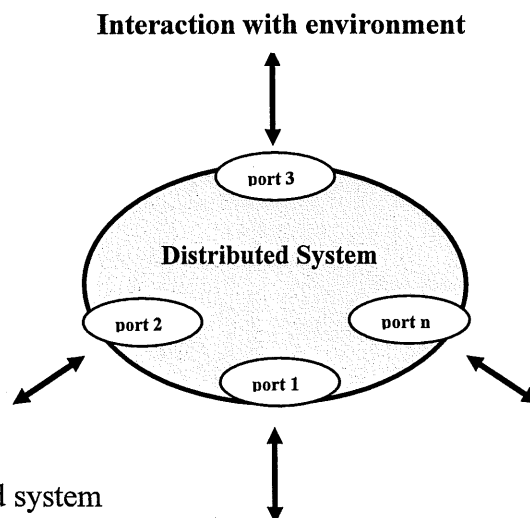


Figure 2.1 Representing of a distributed system

2.3.2 Test architectures

The ISO 9646 standard provides a methodology and framework applicable to conformance testing of OSI product. This standard has been mainly oriented towards practical needs. It incorporates a great deal of practical experience from test experts who have been involved in concrete testing issues. Four test methods are defined in the ISO 9646 standard to test a *centralized* implementation: local test method, distributed test method, coordinated test method and remote test method. [3]

Considering this standard, three architectures to test a *distributed* implementation have been proposed [6]. Depending on the number of testers in the test system, two cases are considered:

- **There is only one tester in the Test System:**

Centralized test architecture: (Fig. 2.2.A)

The test system consists of a single tester that is able to communicate with all ports existing in the IUT (Implementation Under Test). This architecture corresponds to traditional software testing.

- **There are several testers in the Test System:**

Distributed (or remote) test architecture: (Fig. 2.2.B)

The test system consists of several testers, as much as the ports in the IUT, located at different sites. Each tester is able to communicate with its corresponding port in the IUT. Testers cannot communicate with each other, unless indirectly through the IUT.

Coordinated test architecture: (Fig. 2.2.C)

The difference with the distributed test architecture is that a communication medium independent from the IUT is designed which makes the testers able to communicate directly with each other.

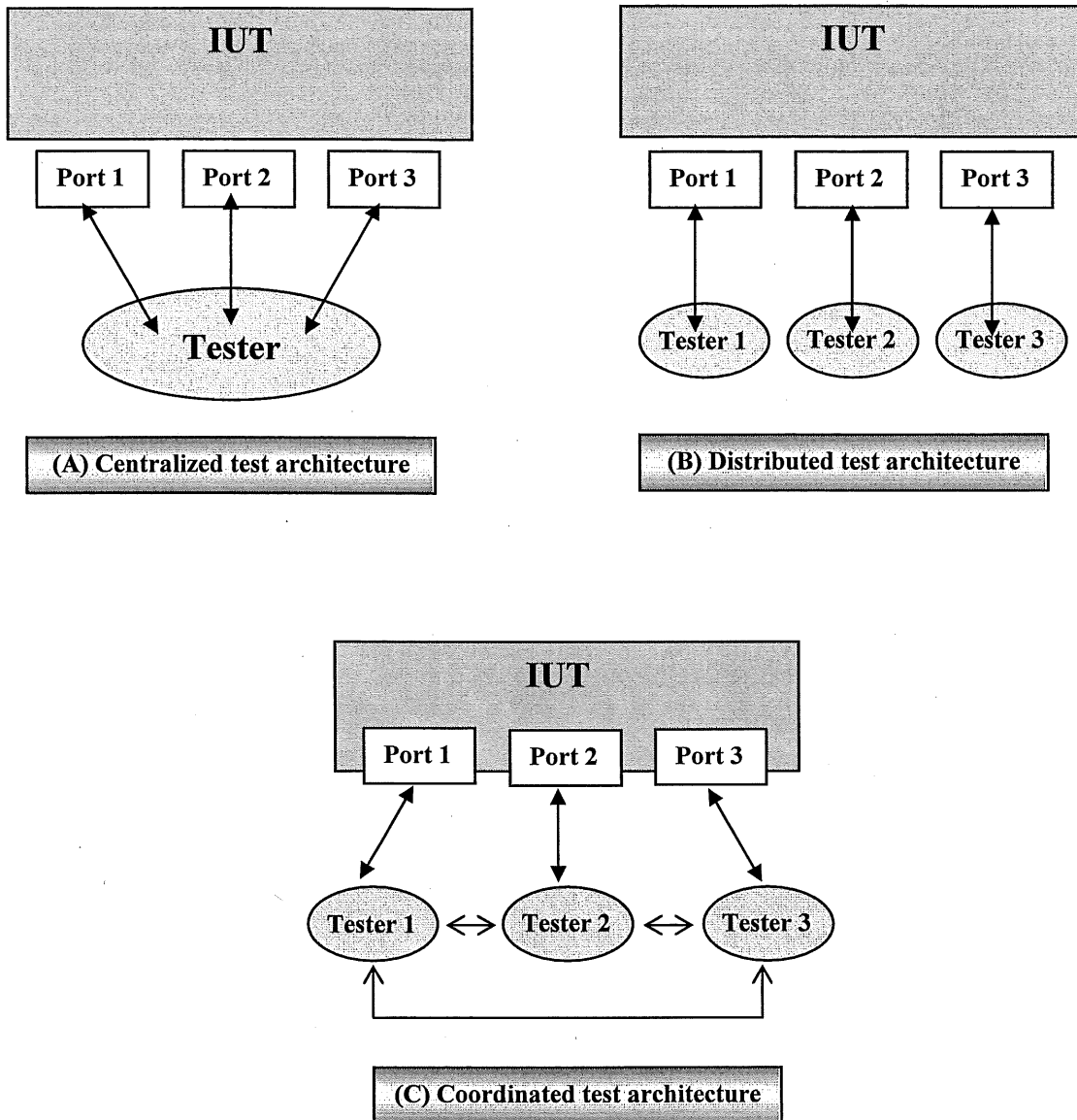


Figure 2.2 Different architectures for testing a distributed system

2.4 Model for describing IUT and its specification

Generally speaking, specifications and related implementations are described in different formalisms. In order to be able to reason about the testing process in a formal setting, the specification and IUT must be modelled by using the same concepts. Therefore, conformance of IUT to its specification may be defined by means of relations between the IUT model and the specification model. I/O FSMs (Input/Output Finite State Machine) [3] that are widely used in the

communication protocol area may be easily adapted with some extensions for modelling distributed systems. In a communication protocol, a protocol entity communicating with a peer entity, can be described by an I/O FSM with one input and one output queue. Distributed applications that are supposed to communicate with multiple partners, lead to the notion of multi-port FSM, which may use several input/output queues called ports.

Definition 2.1. [3]

A multi-port FSM with n ports (np -FSM) is a 6-tuple $A = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where:

- Q is the finite set of states of A ;
- $q_0 \in Q$ is a distinguished state, the initial state of A ;
- Σ is an n -tuple $(\Sigma_1, \Sigma_2, \dots, \Sigma_n)$ where Σ_k is the input alphabet of port k , and $\Sigma_i \cap \Sigma_j = \emptyset$ for $i \neq j$. We write $\bar{\Sigma}$ for $\Sigma_1 \cup \Sigma_2 \cup \dots \cup \Sigma_n$;
- Γ is a n -tuple $(\Gamma_1, \Gamma_2, \dots, \Gamma_n)$ where Γ_k is the output alphabet of port k , and $\Gamma_i \cap \Gamma_j = \emptyset$ for $i \neq j$. We write $\bar{\Gamma}$ for $(\Gamma_1 \cup \{\epsilon\}) \times (\Gamma_2 \cup \{\epsilon\}) \times \dots \times (\Gamma_n \cup \{\epsilon\})$;
- δ is the transition function, it is a partial function $Q \times \bar{\Sigma} \rightarrow Q$;
- λ is the output function; it is a partial function $Q \times \bar{\Sigma} \rightarrow \bar{\Gamma}$. Moreover, $\lambda(q, \alpha)$ is defined if and only if $\delta(q, \alpha)$ is.

A transition of np -FSM A is therefore a 4-tuple $t = (q, \alpha, \gamma, q')$ where $q, q' \in Q$, $\alpha \in \bar{\Sigma}$ and $\gamma \in \bar{\Gamma}$ are such that $\delta(q, \alpha) = q'$ and $\lambda(q, \alpha) = \gamma$.

q and q' are the origin and destination states, α is the input and γ consists of one or several outputs in response to α .

In other words, np -FSM has n ports, numbered from 1 to n . An input alphabet Σ_k and an output alphabet Γ_k are associated with each port k . The input alphabets are disjoint sets as well as the output alphabets. np -FSM A can be represented by a directed graph whose vertices are the states of A and whose arcs are the transitions of A .

Example 2.1.

Here is an example of 2p-FSM A with:

- $Q = \{q_0, q_1, q_2\}$, q_0 being the initial state;
- $\Sigma_1 = \{\alpha\}$, $\Sigma_2 = \{\beta\}$;

- $\Gamma_1 = \{a\}, \Gamma_2 = \{b\}$

- δ and λ are defined by:

$$\lambda(q_0, \alpha) = q_1 \quad \delta(q_0, \alpha) = \langle a, \epsilon \rangle$$

$$\lambda(q_1, \alpha) = q_1 \quad \delta(q_1, \alpha) = \langle a, \epsilon \rangle$$

$$\lambda(q_1, \beta) = q_2 \quad \delta(q_1, \beta) = \langle a, b \rangle$$

$$\lambda(q_2, \beta) = q_0 \quad \delta(q_2, \beta) = \langle a, \epsilon \rangle$$

A has four transitions: t_1, t_2, t_3, t_4

For example, if currently we are in the state q_1 and we receive α , then the output is a in port 1 and we remain in the state q_1 .

And if while we are in state q_1 , we receive β , then the outputs are a and b in ports 1 and 2, respectively.

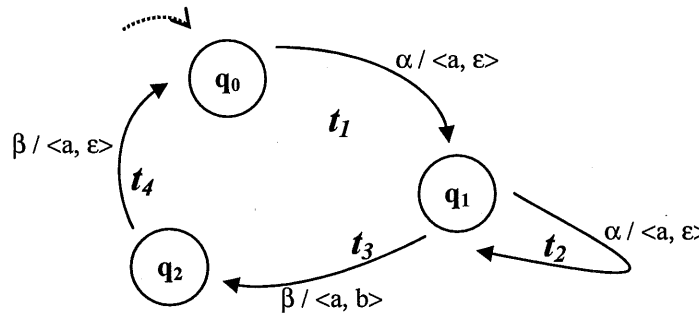


Figure 2.3. Example of a 2p-FSM

2.5 Fault model

The fault model allows to define formally the type of faults we intend to detect. [2].

FSM based fault model includes:

- Output fault:

A transition has an output fault if, for the corresponding state and input received, the implementation provides an output different from the one specified by the output function.

- Transfer fault:

A transition has a transfer fault if, for the corresponding state and input received, the implementation enters a different state than the one specified by the transfer function.

- Transfer faults with additional states:

In most cases, one assumes that the number of states of the system is not increased by the presence of faults. (Note that a smaller number of states could be explained by normal transfer faults making a subset of the states unreachable.) Certain types of errors can only be modelled by additional states, together with transfer faults, which lead to these additional states.

- Additional or missing transitions:

In many cases, it is assumed that specification is described by a deterministic and completely specified FSM, that is, for each pair of present state and input, there is exactly one specified transition. In the case of incompletely specified machines, no transition may be specified for a given pair. While in the case of non-deterministic machines, more than one transition may be defined. In these cases, the fault model could include additional and / or missing transitions.

In this study, we consider the output faults.

2.6 Test sequence

In section 2.4, we presented a model to describe the IUT and its specifications. Now we present test sequences. Test sequences are generated from the specification of the IUT and are sequences of inputs, which are sent to IUT, each sequence followed by the expected outputs from IUT. As we mentioned before, we use np-FSM to describe the specification of IUT, and a test sequence can be seen as a path through the specification. In fact, test sequences are executions of the specification in order to check whether IUT conforms to each generated test sequence. Test sequences are characterized by the types of faults they allow to detect (fault models are introduced in section 2.5) and in this study we consider only output faults.

Definition 2.2.

We consider a np-FSM(definition 2.1) $A = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ that is a specification of an IUT. A test sequence of A is a trace of A and is defined in the form:

$\omega = !X_1 ?Y_1 !X_2 ?Y_2 \dots !X_t ?Y_t$, where for $i = 1, 2, \dots, t$, $X_i \in \bar{\Sigma}$ and $Y_i \in \bar{\Gamma}$.

$!X_i$ denotes the sending of X_i by the tester and $?Y_i$ denotes the receptions by the tester of all outputs of Y_i .

In testing, there is a phase, which is called *test case generation* whose aim is to generate a set of test sequences from the specification. For example, classical test sequence generation methods, based on graph traversal, can be used. For instance, *transition tour* method[12], consists of taking a path in the specification, starting at the initial state and executing each transition at least once. Clearly, determining a transition tour corresponds to solve the Chinese Postman Problem [9] for the graph of np-FSM. A transition tour exists if and only if the graph is strongly connected [3].

Example 2.2.

For the example 2.1, we obtain the following transition tour: $t_1. t_2. t_3. t_4.$

This transition tour gives the following test sequence: $! \alpha ? a ! \alpha ? a ! \beta ? \{a,b\} ! \beta ? a$

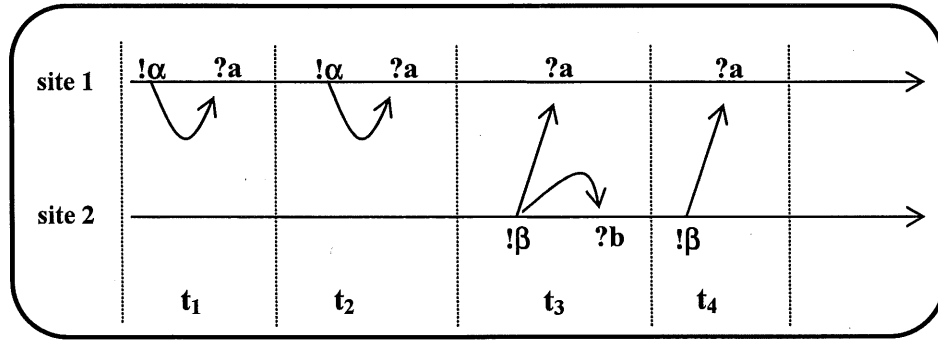


Figure 2.4 The test sequence

Figure 2.4 represents the test sequence for the example 2.2, where arrow(s) originally from the same input correspond to the same transition and associate the input to the output(s) of the transition.

With a distributed test architecture (see Fig. 2.2.b), each test sequence is called *global test sequence (GTS)*, which must be projected into *local test sequences (LTS)*, where each local test sequence is executed by a local tester.

It should be mentioned that we would not study methods for generating GTS, since we are interested in how to execute given GTSSs.

Definition 2.3 (local test sequence (LTS))

Let A be np-FSM (definition 2.1), with a distributed test architecture, local test sequence for port k of A is a sequence:

$\alpha_1 \alpha_2 \dots \alpha_t$

Where each α_i is one of the following:

- $!x$: sending of message $x \in \Sigma_k$ to IUT,
- $?y$: receiving of message $y \in \Gamma_k$ from IUT.

For the example 2.1, these are the LTSs:

$LTS_1: !\alpha ?a !\alpha ?a ?a ?a$

$LTS_2: !\beta ?b !\beta$

LTS_1 and LTS_2 are represented in Fig. 2.4 in the axes of site 1 and site 2, respectively.

CHAPTER 3

CONTROLLABILITY AND OBSERVABILITY

3.1 Introduction

If the centralized method does not pose any particular problem, the distributed (also called remote) architecture raises several problems in terms of *controllability* and *observability*, which are fundamental features of conformance testing. In this chapter we discuss these two issues.

3.2 Controllability and Observability

Controllability and Observability are two concepts in testing because they have an effect on the capability of the test system (TS) to check the conformance of an IUT.

Controllability

Definition 3.1

Controllability is the capability of the TS to force the IUT to receive the inputs in a given order. For a given GTS $W = !X_1 ?Y_1 !X_2 ?Y_2 \dots !X_t ?Y_t$, a controllability problem arises when the TS cannot guarantee that the IUT will receive X_i before X_{i+1} , for $i < t$. With a distributed test architecture, such a problem arises when there exists $i < t$ such that the port of X_{i+1} : [5]

- (1) is different from the port of X_i and
- (2) is not included in the set of ports of Y_i .

Observability

Definition 3.2

Observability is the capability of the TS to observe the outputs of the IUT and to determine the input, which is the cause of every output. For a given GTS $W = !X_1 ?Y_1 !X_2 ?Y_2 \dots !X_t ?Y_t$, an

observability problem arises when the TS receives a message $a \in Y_i$ and cannot determine whether a has been sent by the IUT after the latter has received X_i and before it receives X_{i+1} [5].

Example 3.1

To clarify the Controllability and Observability problems, an example is brought here:

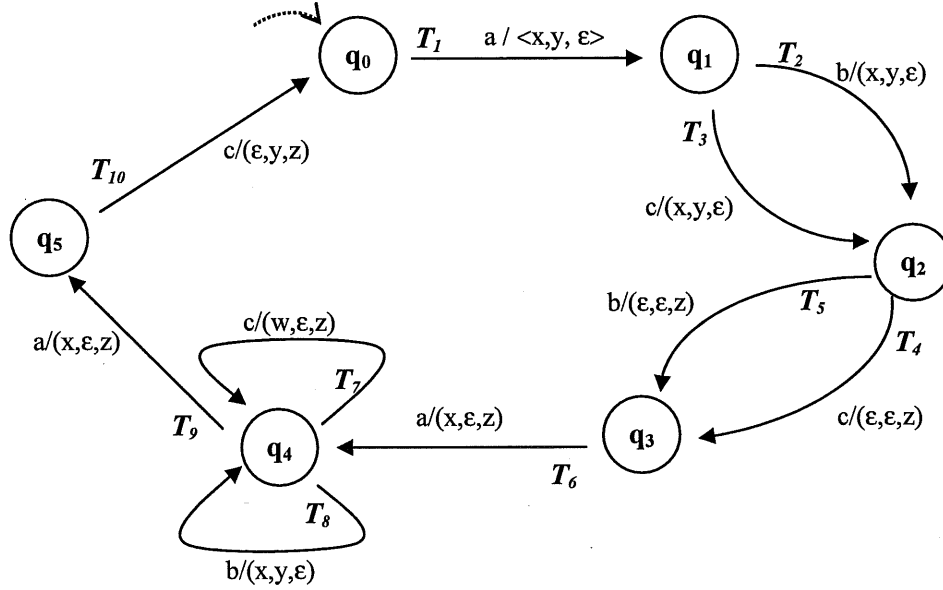


Figure 3.1 Specifications of an IUT

Figure 3.1 is a 3p-FSM, which describes the specification of an IUT.

In this 3n-FSM, a is input of port 1, b is input of port 2, c is input of port 3, x and w are outputs of port 1, y is output of port 2 and z is output of port 3.

The following sequence is an example of *sequence transition*:

$T_1 T_2 T_4 T_6 T_8 T_7 T_9 T_{10}$

The global test sequence (GTS) corresponding to this sequence transition is:

$$W = !a?\{x,y\} !b?\{x,y\} !c?\{z\} !a?\{x,z\} !b?\{x,y\} !c?\{w,z\} !a?\{x,z\} !c?\{y,z\} \quad (1)$$

Figure 3.2 models a faulty IUT, that is, the IUT does not conform to the specification of Fig. 3.1.

The six faults are underlined.

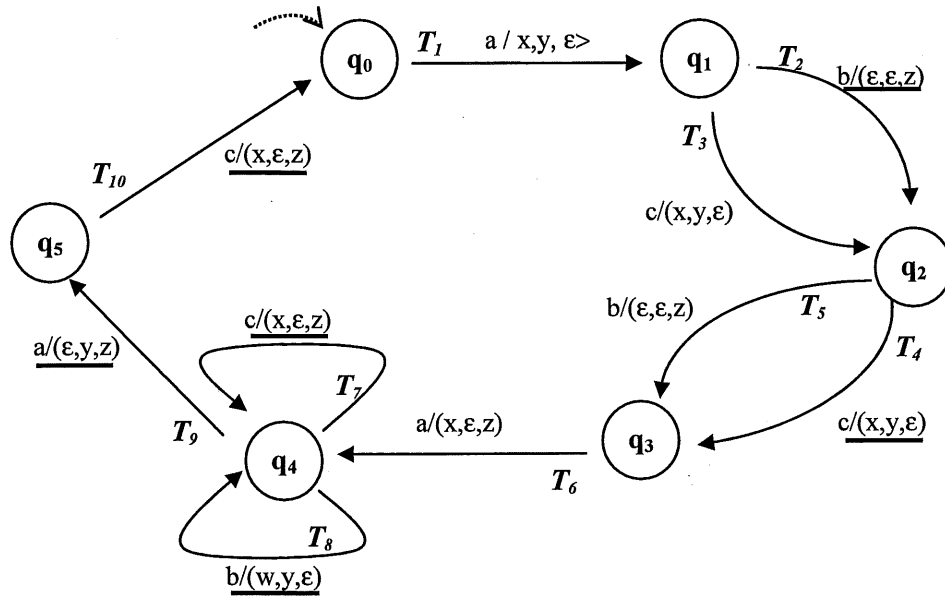


Figure 3.2 Faulty implementation

Let us apply the input sequence of the GTS (1) to this faulty IUT, using the centralized test architecture (see Fig. 2.2.A). We start in q_0 (starting state) and in transition T_2 after the TS sends b to the IUT, it detects a fault because the IUT sends z instead of (x,y) to the TS.

If we use the distributed test architecture (see Fig. 2.2.B), we have to project the GTS (1) in each port in the IUT, to obtain the local test sequences (LTS):

$$W1 = !a ?x ?x !a ?x ?x ?w !a ?x$$

$$W2 = ?y !b ?y !b ?y ?y \quad (2)$$

$$W3 = !c ?z ?z !c ?z ?z !c ?z$$

$W1$, $W2$ and $W3$ are represented in Fig. 3.3 in the 3 axes respectively.

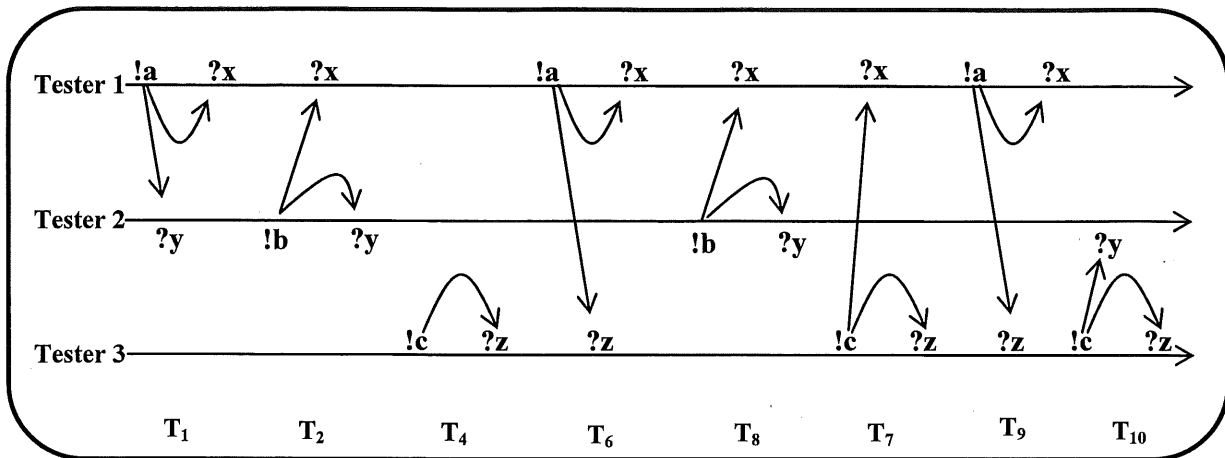


Figure 3.3 Projection of a global test sequence

In figure 3.3, originating from the same input correspond to the same transition. They represent a causality relation, because in each transition T : Input of T is the cause of output(s) of T .

With this method of projection for obtaining LTSs from GTS, there are two types of problems, called controllability and observability problems. Here we bring two examples, which illustrate these two problems:

Controllability problem

Example 3.2

To illustrate the controllability problem, we use figures 3.4 and 3.5 corresponding to figures 3.1 and 3.2, but focused on state q_4 . Figure 3.4 corresponds to the specification and figure 3.5 corresponds to the faulty implementation.

According to GTS (1), when we arrive to state q_4 , T_8 must be executed before T_7 , that is, IUT has to receive b which is sent by Tester 2 and then it has to receive c which is sent by Tester 3. To respect this order, Tester 3 must receive a message to be aware that IUT has received b . If we use a distributed test architecture, such a message may come only from IUT. But as we can see in Fig. 3.4, Tester 3 doesn't receive any output from IUT, in response to the input b in transition 8 (indicated by dotted arrow). This is an example where the TS cannot guarantee a given order of inputs.

From state q_4 , the three testers observe the same outputs and in the same order:

- Either after T_8T_7 (when a correct IUT receives b before c),
- Or after T_7T_8 (when the faulty IUT receives c before b).

Since the Test System (TS) is not aware of the order of inputs b and c , it cannot deduce if the IUT is faulty.

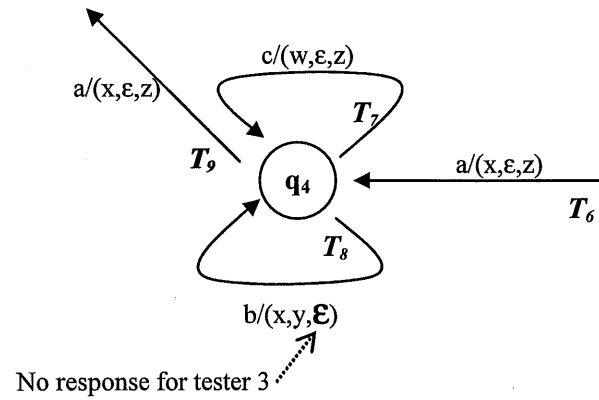


Figure 3.4 Specifications

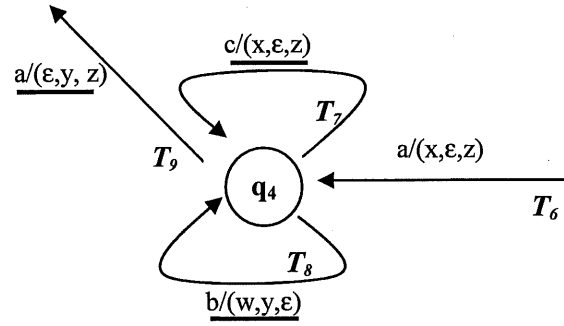


Figure 3.5 Faulty Implementation

Observability problem

Example 3.3

To illustrate the observability problem, we choose two transitions T_9 and T_{10} from figures 3.1 and 3.2, which are showed in figures 3.6 and 3.7. As we can see, in the faulty implementation (fig.3.7), in comparison with the specification (fig. 3.6), input x has been shifted from T_9 to T_{10} and y has been shifted from T_{10} to T_9 .

If we use the distributed test architecture, these faults are not detectable because, although the IUT is faulty, the three testers execute the LTSs (2) generated from GTS (1), which are illustrated in Fig. 3.3. The testers observe the same thing after T_9T_{10} or $T_{10}T_9$.

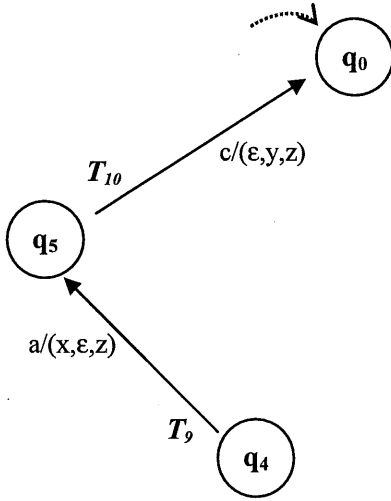


Figure 3.6 Specifications

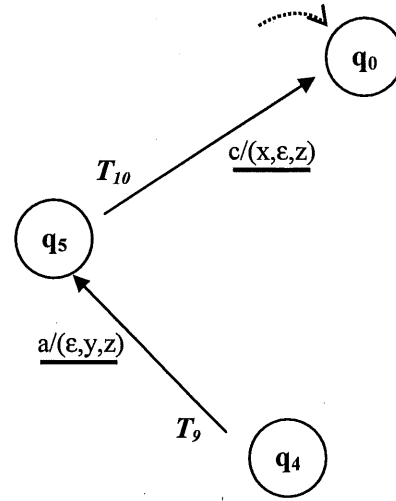


Figure 3.7 faulty Implementation

3.3. Solution to controllability and observability problems

As we observed, in distributed test architecture we have controllability and observability problems. Now in order to solve these two problems, we introduce the coordinated test architecture, represented in figure 2.2.C and 3.8 in two different ways.

In this architecture, testers are able to exchange messages through a communication service. This communication service is independent of IUT and allows the test system to coordinate the related testers. To carry out communication between testers, we use a multicast channel, through which testers may exchange coordinated messages. In this approach, the local test sequences are replaced by *coordinated local test sequences* that contain coordination messages to and from other testers. Each tester executes the coordinated local test sequence constructed from the global test sequence of the IUT.

A coordinated local test sequence is in the form $\alpha_1, \alpha_2 \dots \alpha_t$, where α_i is either:

- !x, sending of message $x \in \Sigma_k$ to the IUT,
- ?y, receiving of message $y \in \Gamma_k$ from the IUT,
- $-X_{h1,\dots,hr}$, sending of coordination message X to testers h_1,\dots,h_r ,
- $+X_h$, reception of coordination message X from tester h .

We use two kinds of coordination message, C and O (i.e., $X = C$ or O)

C for guaranteeing Controllability,

O for guaranteeing Observability.

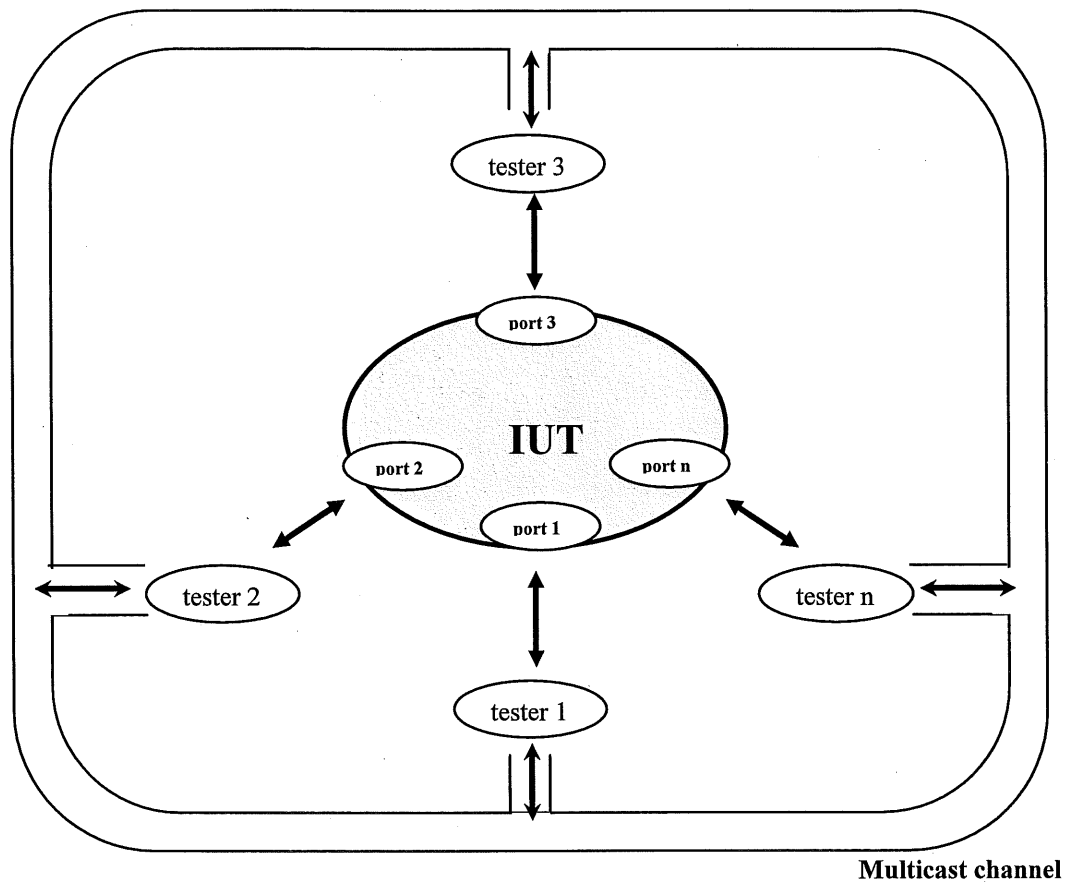


Figure 3.8 The coordinated test architecture

Now we explain the approaches to solve controllability and observability problems.

We consider a global test sequence (GTS) $\omega = !x_1 ?Y_1 !x_2 ?Y_2 \dots !x_t ?Y_t$

Approach to solve the controllability problem [3,5]

The controllability problem arises when, for $i < t$, the tester, which sends x_{i+1} has no possibility to know whether x_i has been received by the IUT. Let $Tester_h$ and $Tester_k$ be the testers sending x_i

and x_{i+1} , respectively, and $Tester_m$ be one of the testers which must receive an output (if any) $a \in Y_i$:

If $Y_i = \emptyset$: after it sends x_i , $Tester_h$ sends a message C (Control) to $Tester_k$;

If $Y_i \neq \emptyset$: after it receives a , $Tester_m$ sends a message C to $Tester_k$.

In both cases ($Y_i = \emptyset$ or $Y_i \neq \emptyset$), after it receives message C $Tester_k$ sends x_{i+1} to IUT.

Approach to solve the observability problem [3,5]

The observability problem arises when, for port p and $i < t$, either Y_i or Y_{i+1} contains an output in port p . Let $Tester_h$ be the tester sending x_{i+1} and $Tester_p$ be any tester which receives an output of Y_i and no output of Y_{i+1} and $Tester_q$ be any tester which receives an output of Y_{i+1} and no output of Y_i .

Before it sends x_{i+1} , $Tester_h$ sends a message O (Observation) to every $Tester_p$ and $Tester_q$.

If we apply these two approaches to the test sequence (1), we obtain the following coordinated local test sequences: (Fig. 3.9)

$$\omega_1 = !a ?x ?x +O_3 + C_3 !a ?x ?x ?w !a ?x +O_3$$

$$\omega_2 = ?y !b ?y -C_3 +C_3 !b ?y -C_3 +O_3 ?y$$

$$\omega_3 = +C_2 -O_1 !c ?z -C_1 ?z -C_2 +C_2 !c ?z ?z -O_{\{1,2\}} !c ?z$$

Fig. 3.9 is obtained by adding coordination messages to Fig. 3.3.

The algorithm, which produces these coordinated test sequences, is introduced in appendix B, and also a step-by-step execution of this algorithm is brought in this appendix.

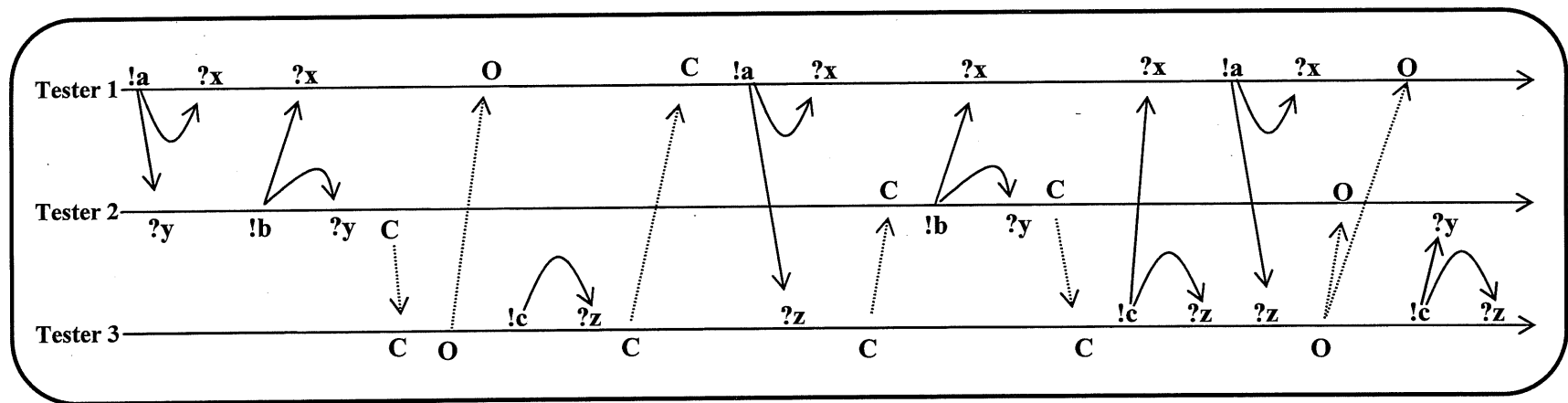


Figure 3.9 The coordinated local test sequences for three ports

CHAPTER 4

PROPOSED TEST ARCHITECTURE

This chapter shows in detail the structure of the proposed coordinated test architecture with its components. In section 4.2, we study the details of each part of this test architecture.

4.1 Architecture

4.1.1 General architecture

In this section we propose a test architecture, which consists of three parts:

- 1- Test Controller (TC)
- 2- Test System (TS)
- 3- Implementation Under Test (IUT)

This architecture corresponds to the coordinated test architecture presented in chapter 3, which solves the controllability and observability problems. In the next page you can see the general idea of the proposed test architecture (Figure 4.1). In the represented architecture, for the sake of clarity and without loss of generality, we consider an IUT with two ports. Therefore we have two testers in the test system.

In this architecture, the user provides the test controller with a Global Test Sequence (GTS), and the aim of TS is to check the conformance of IUT to this GTS. The test controller (TC) produces the coordinated Local Test Sequences (LTS (from now LTS means: coordinated local test sequence)), and provides the testers with these LTSs. Then each tester executes its LTS. After execution, each tester generates a local verdict, either 'pass' or 'fail'. A 'pass' verdict is generated by a tester if and only if its LTS is correctly executed. Then testers send their local verdicts to TC, and based on these local verdicts, TC generates the global verdict. The latter is 'pass' if and only if all local verdicts are 'pass'.

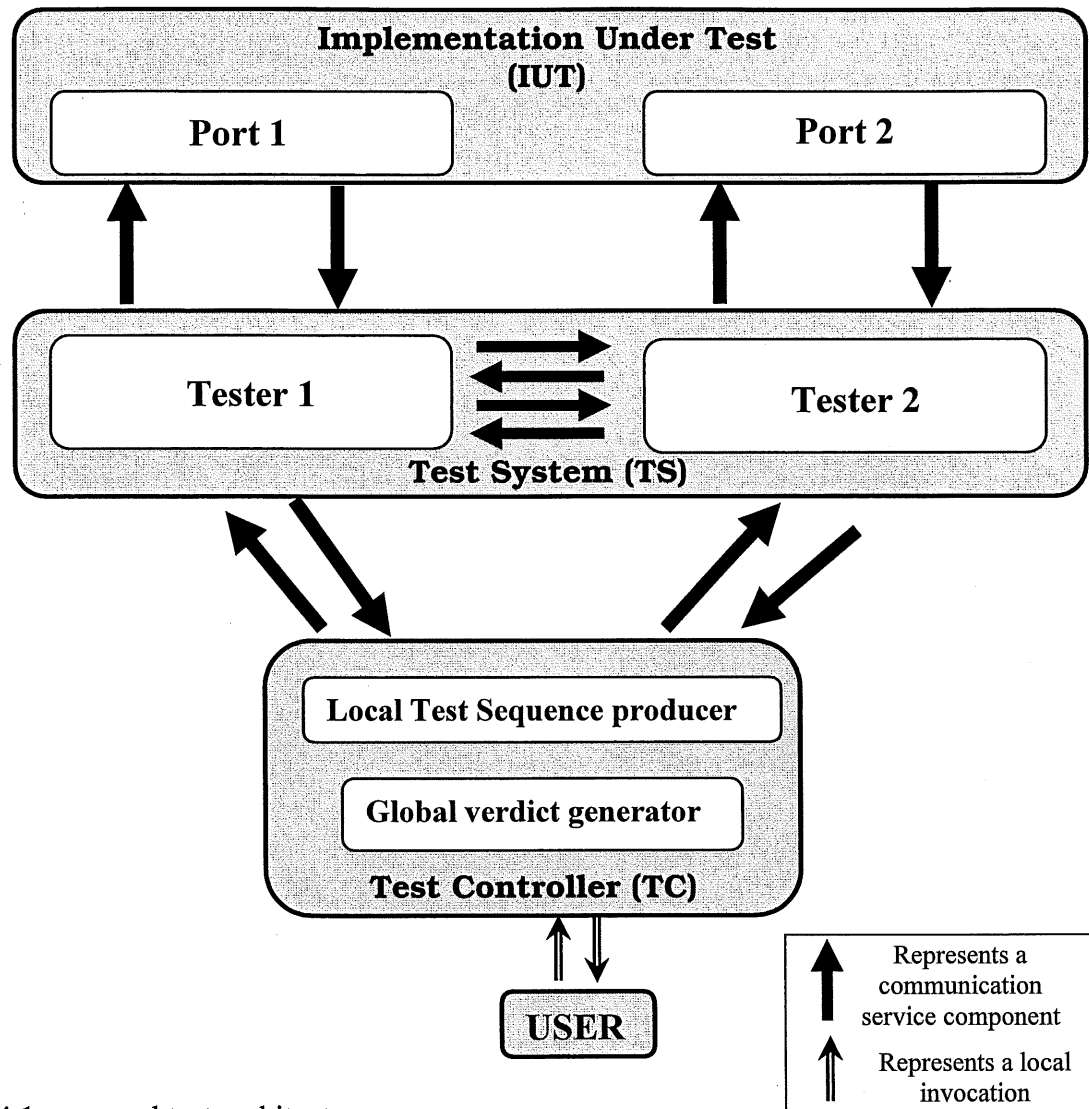


Figure 4.1 proposed test architecture

4.1.2 Adding an interaction system

As you can see, each tester is directly communicating with its corresponding port of IUT.

The aim is to test the IUT which is given, and the communication service used by IUT to interact with its environment may not be compatible to the one used by TS to interact with IUT.

For example, IUT uses Java RMI, and TS uses CORBA. In such a case, we will not be able to establish a connection between TS and IUT, in order to send inputs to IUT and receive outputs from IUT, to test correctness of IUT.

So, in order to solve this problem, we improve the architecture by adding an Interaction System (IS) between IUT and TS (Figure 4.2). The IS is responsible for the communication between ports of IUT and their corresponding testers in TS.

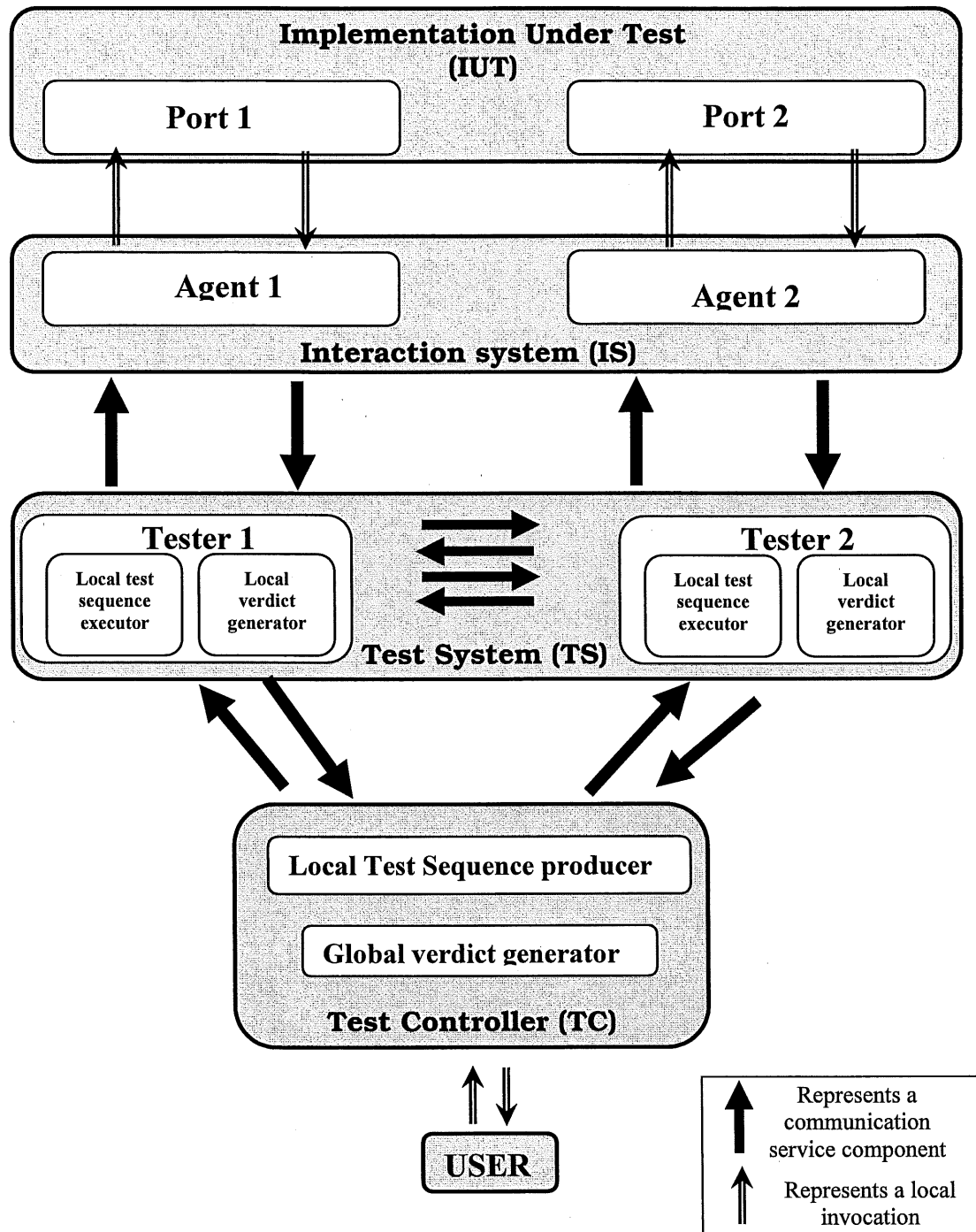


Figure 4.2 Improved proposed test architecture

Regardless what kind of communication services TS and IUT are using, IS receives inputs from TS through communication service, converts them to be compatible to IUT, and then it provides IUT with these inputs. IS also receives outputs from IUT, converts and then sends them to TS

through communication service. So for every set of IUTs using a given communication service, we only have to change the IS, to be converter and adapter between our TS and the IUTs. For example, we assume that IUT receives and sends inputs and outputs, respectively, from and to a text file. In this case, the interaction system is illustrated in figure 4.3.

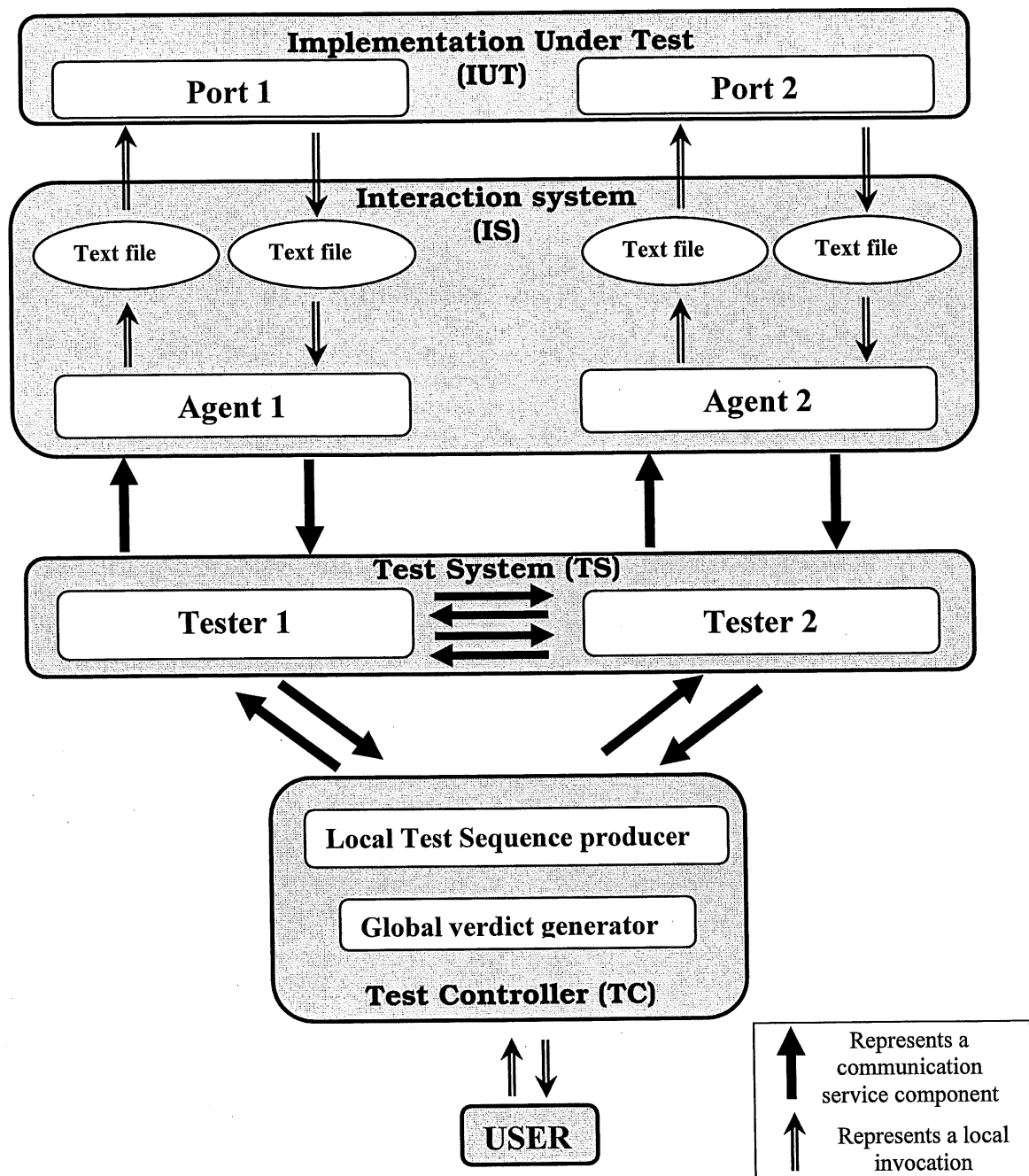


Figure 4.3 Architecture of our example

- For this example, agents are input/output converters and communication service adapters, since
- They receive inputs from TS and convert and insert them in the text files and reads outputs of ports from text files and convert and send them to testers in TS (since the nature of inputs sent by TS might not be compatible for IUT, and outputs sent by IUT may not be compatible for TS, so the conversion is a necessary task of agents),
 - And they are intermediators to adapt the communication service between TS and IUT.

4.1.3 Constructing an IUT

Normally we test existing IUTs, but in order to validate TS by applying it to several IUT, we have developed a program that construct an IUT. We take the following steps to construct an IUT:

- We design a np-FSM that models the IUT,
- Based on this FSM, we create a global text file, which contains all inputs received by IUT and all outputs sent by IUT considering all ports. Each line of this text file is the following form:

$(Q_n/T:x:I)=(\langle T:x:O, \dots, T:x:O \rangle; Q_m)$;

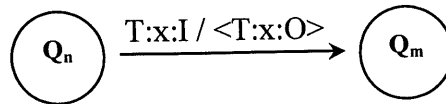
Current state: Q_n is the current state, where we currently are on the IUT's FSM.

Input(s): $T:x:I$ where 'T' denotes "tester", 'x' is the tester's number, 'I' is the input sent by tester x and received by port x of IUT.

Output(s): $T:x:O$ where 'T' denotes "tester", 'x' is the number of tester, 'O' is the output received by tester.

Next state: $Q_m \rightarrow$ the next state where we should go on the FSM of IUT.

That is each line of this text file is: two states and the edge between them, on the IUT's FSM, which represents a transition as follows:



- We create a local text file for each port in IUT by projecting the global text file to the ports. Each of these files contain just the inputs, outputs and states related to a port. Each line of this file is the following form:

$(Q_n/D:x:I)=(\langle S:x:O, \dots, S:x:O \rangle; Q_m)$;

Current state: Q_n is the current state, where we currently are on the port's FSM (port's FSM is explained in example 4.1 step c)

Input(s): $D:x:I$ where 'T' denotes either "tester" or another "port" of IUT, 'x' is the tester's or port's number, 'I' is the input sent by tester x or an IUT internal message from other ports of IUT.

Output(s) and IUT internal message(s): $S:x:O$ represents an output or an IUT internal message where 'S' denotes "tester" or "port", 'x' is the number of tester or the number of port which receives the message, 'O' is the output sent to tester or the IUT internal message sent to another port of IUT.

Next state: $Q_m \rightarrow$ the next state where we should go on the FSM of port.

- d- Then we run the program (realization) and create the port tables for each port. They are created from local text files. Each port table is a hash table that contains the exact information of its corresponding local text file.
- e- In this step, ports and the communication service between ports are created and IUT gets ready to interact with environment.

Here, steps a, b and c are done manually by the one who is constructing the IUT, but as a future work step c can be done automatically and steps d and e are done automatically after running the program.

Example 4.1

To clarify these steps we apply them on the example 3.1.

Step a: in this step we design a FSM from the IUT's specification. We obtain the 3n-FSM introduced in Fig. 3.1.

In this example we have 3 ports. The inputs are: a input for port 1, b input for port 2, c input for port 3. The outputs are: x, w outputs for port 1, y output for port 2 and z output for port 3.

Step b: in this step we create the general text file, which contain all information of above FSM.

Figure 4.4 represents this text file.

```

(Q0/T:1:a) = (<T:1:x, T:2:y>; Q1);
(Q1/T:2:b) = (<T:1:x, T:2:y>; Q2);
(Q1/T:3:c) = (<T:1:x, T:2:y>; Q2);
(Q2/T:2:b) = (<T:3:z>; Q3);
(Q2/T:3:c) = (<T:3:z>; Q3);
(Q3/T:1:a) = (<T:1:x, T:3:z>; Q4);
(Q4/T:3:c) = (<T:1:w, T:3:z>; Q4);
(Q4/T:2:b) = (<T:1:x, T:2:y>; Q4);
(Q4/T:1:a) = (<T:1:x, T:3:z>; Q5);
(Q5/T:3:c) = (< T:2:y, T:3:z>; Q0);

```

Figure 4.4 The general text file

Each line of this file corresponds to a transition. We explain the first line of this text file.

$(Q0/T:1:a) = (<T:1:x, T:2:y>; Q11) ;$

when we are in state Q_0 and port 1 receives a , then port 1 sends x and port 2 sends y and then we go to state Q_1 .

Before presenting step c, let us first present two concepts: service and protocol. Service describes what the user sees, and protocol describes how the service is realized (i.e., implemented). Here we bring a simple example, a FSM, which has two states.

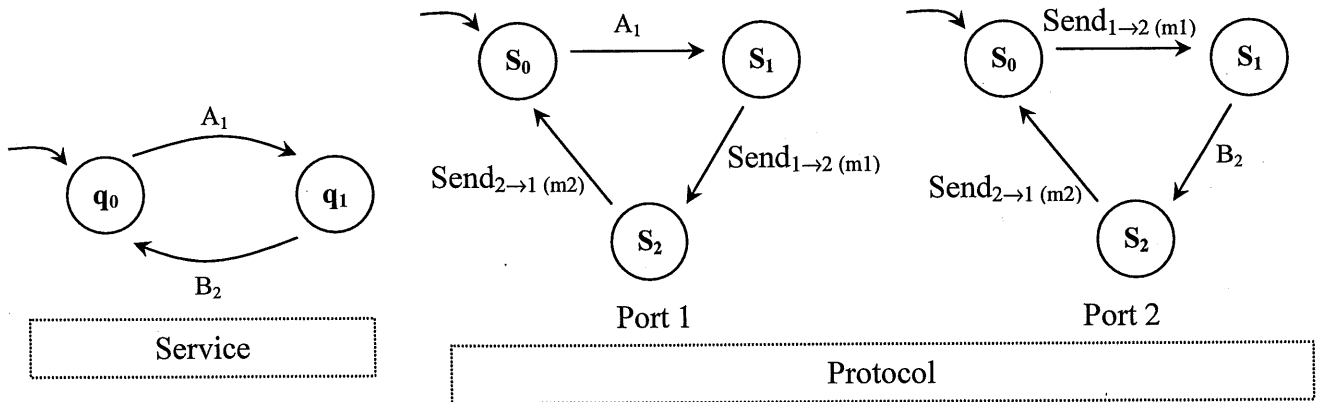


Figure 4.5 service and protocol

Service: we have a loop $A_1 B_2 A_1 B_2 \dots$ first port 1 is needed by user to execute action A_1 then port 2 is needed by user to execute action B_2 and ...

Protocol:

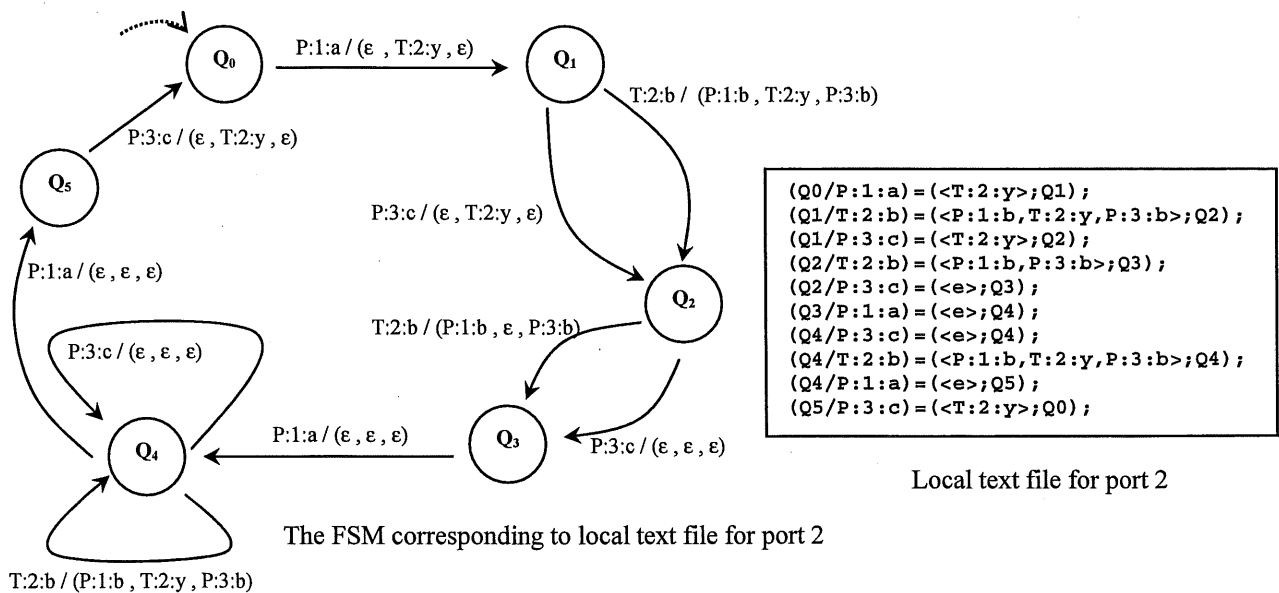
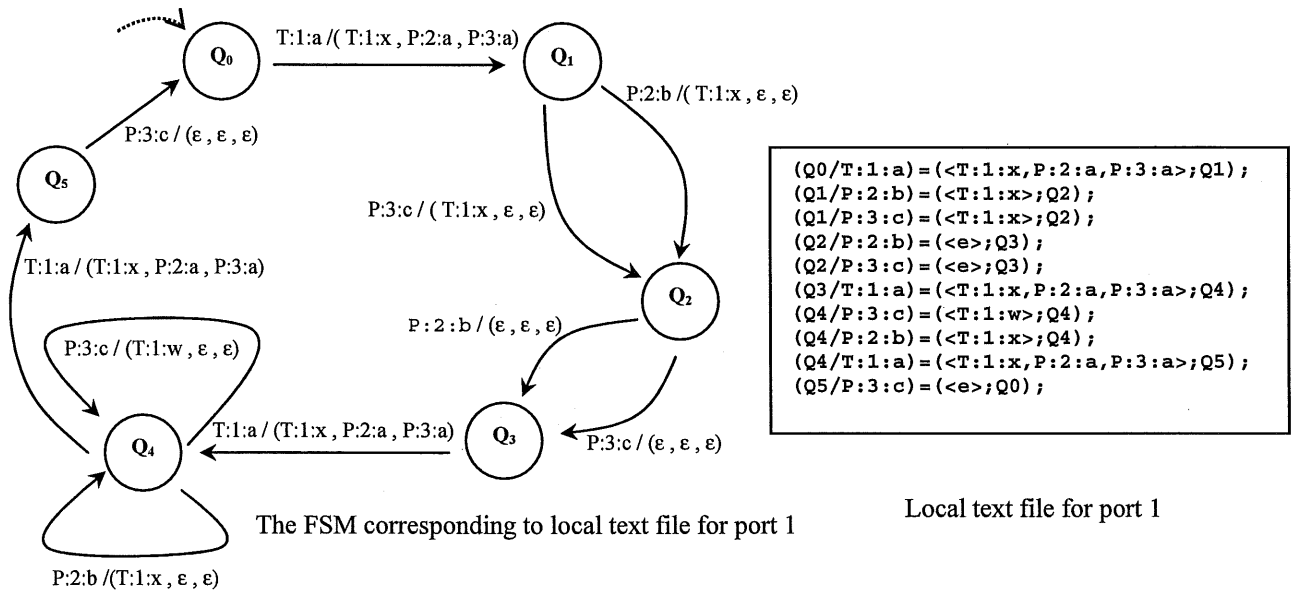
Port 1: we are in state S_0 , port 1 executes action A , we go to state S_1 , port 1 sends a message to port 2 to inform it that it (port 1) has executed its task, we go to state S_2 then port 1 receives a

message from port 2 in order to be informed that port 2 has executed action B, then we go back to state S_0 .

Port 2: we are in state S_0 , port 2 receives a message from port 1 which means that port 1 has done its task, we go to state S_1 , then port 2 executes action B and we go to state S_2 then port 2 sends a message to port 1 to inform port 1 that it has done its task, we go back to state S_0 .

Step c: The aim of step c is to obtain a protocol description from the service description.

Now we create the local text files for each port. Figure 4.6 represents these text files. To clarify we have also added a FSM corresponding to each local text file.



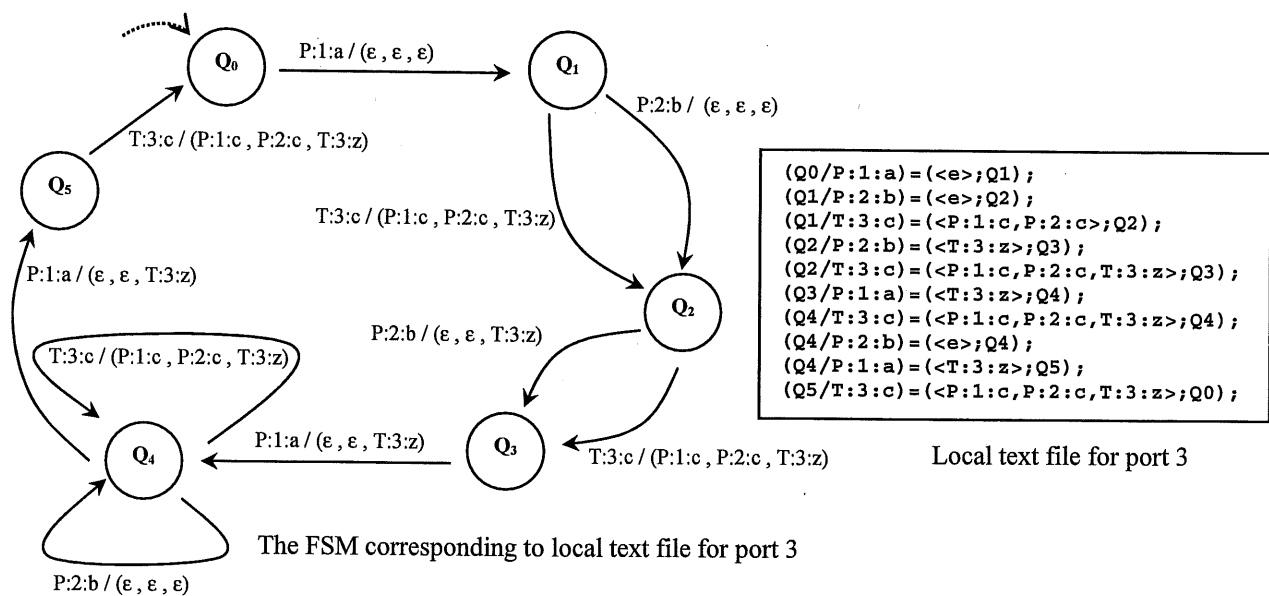


Figure 4.6 The local text files

We explain the fourth line of local text file for port 1 which is $(Q2/P:2:b) = (<e>;Q3)$;
 When we are in state $Q2$, port 1 receives IUT internal message b from port 2, then it sends nothing (e denotes null) as output, then we go to state $Q3$.
 As you can see a local text file contains all information corresponding to a port. For example the local text file for port 1 contains all inputs, outputs and IUT internal messages sent and received by port 1.

Step d: now we run the program. The program uses the local text files and creates the port tables automatically. A port table is a Direct-Access (hash table) as data structure, and contains the exact data of local text file. The reason to choose the hash table is that, it can be used to implement the *insert* and *find* operation in constant average time. Figure 4.7 represents the port table for port 1.

Input	Output	Next State
Q0/T:1:a	T:1:x, P:2:a, P:3:a	Q1
Q1/P:2:b	T:1:x	Q2
Q1/P:3:c	T:1:x	Q2
Q2/P:2:b	e	Q3
Q2/P:3:c	e	Q3
Q3/T:1:a	T:1:x, P:2:a, P:3:a	Q4
Q4/P:3:c	T:1:w	Q4
Q4/P:2:b	T:1:x	Q4
Q4/T:1:a	T:1:x, P:2:a, P:3:a	Q5
Q5/P:3:c3	e	Q0

Figure 4.7 The port table for port 1

Once port receives an input from corresponding tester in TS, it uses the hash table to find the output corresponding to the received input and also to find the next state.

Step e: in this step the IUT is constructed and get ready to interact with environment. The constructed IUT is represented in figure 4.8.

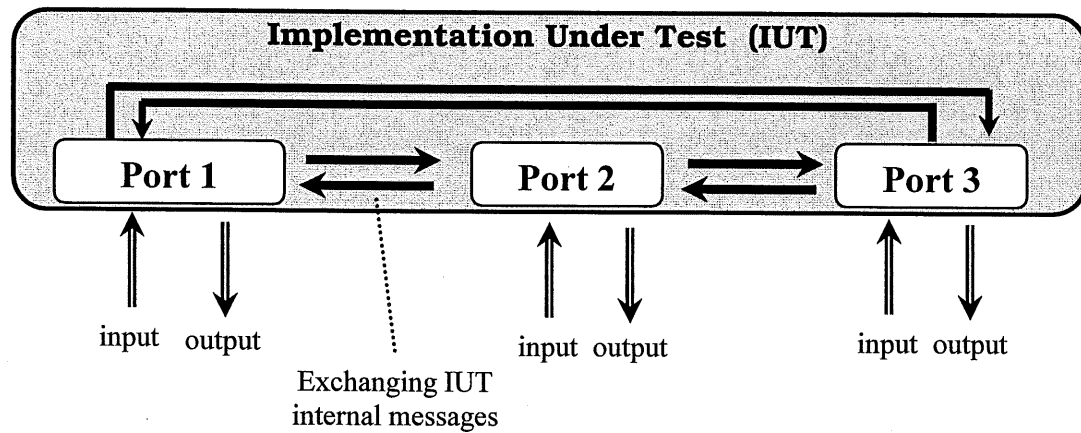


Figure 4.8 Constructed IUT after taking the steps a to e

4.2 Components

Now we go through the architecture's components and discuss its different parts in detail.

It should be mentioned that there are some arrows (links) like \longrightarrow in the figures where we explain the components of the test architecture. These arrows indicate that the origin component invokes a local method on the destination component to provide the latter with a data (the flow of the data).

4.2.1 Test Controller (TC)

Here are the components of TC:

- 1- An array of threads (LocalVerdictReceiver[]): TC receives the verdicts from testers in TS through these threads.
- 2- Global verdict generator: this component generates the global verdict using the local verdicts received from testers.
- 3- LTS producer: this component produces local test sequences (LTS) from the global test sequence (GTS) provided by user.

These components are represented in figure 4.9.

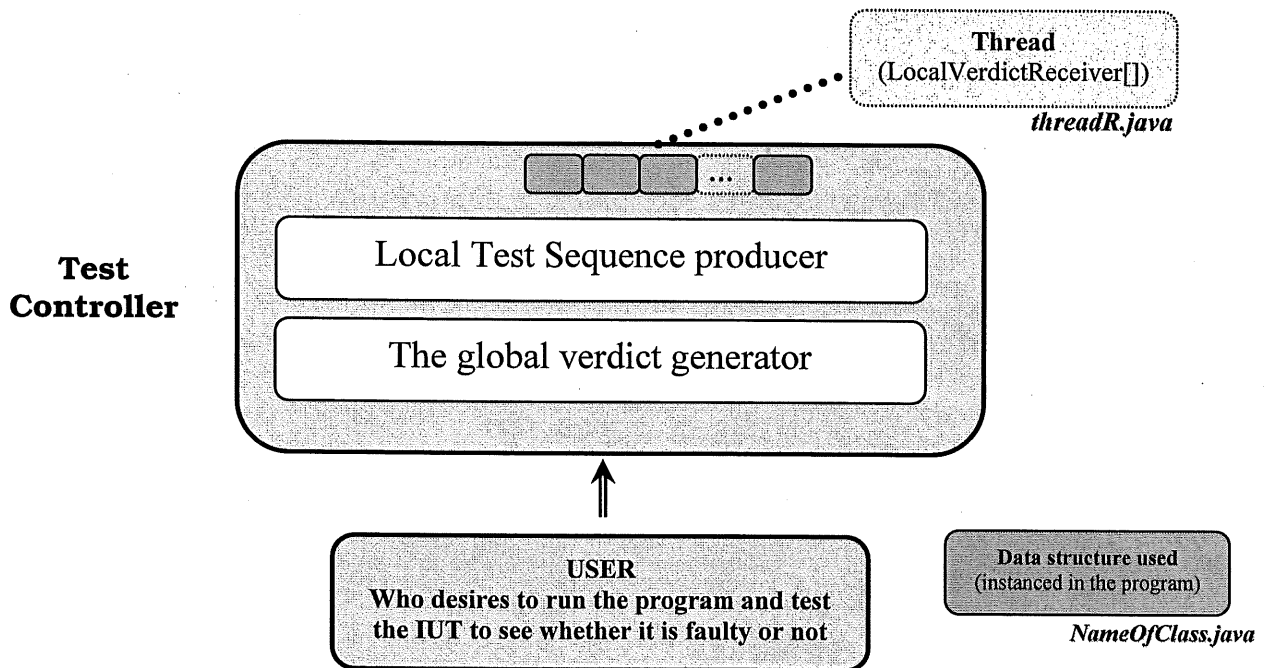


Figure 4.9 The components of Test Controller

TC receives the name of a text file from user, which contains the GTS. Then based on GTS, LTS producer creates the LTSs. TC also receives the name of IUT and the number of ports existing in IUT from the user, then TC sends the following to each tester:

- a- LTS corresponding to the tester,
- b- The name of IUT,
- c- The total number of ports running on the IUT.

Then TC waits to receive a verdict from each tester. If all received verdicts are “pass” then TC determines that IUT as correct. Otherwise if it receives at least one “fail” verdict from one of the testers, then it determines that IUT is faulty.

4.2.2 Test System (TS)

Here are the components of Test System:

- 1- Testers
- 2- Communication service

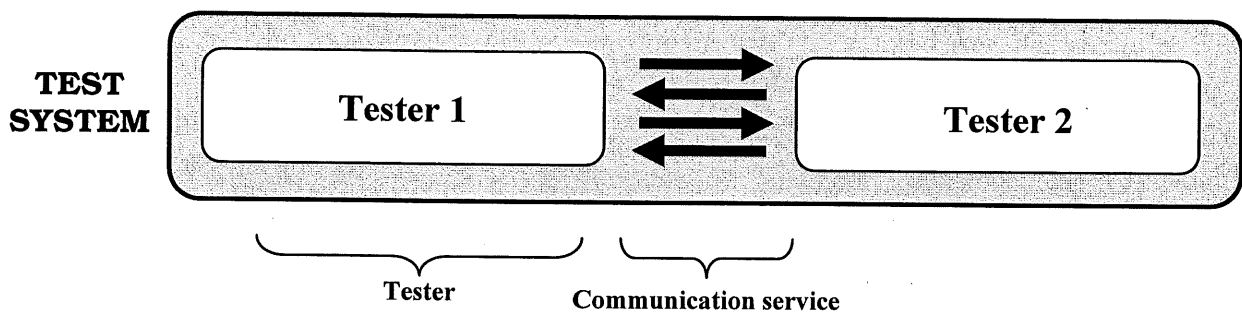


Figure 4.10 Test System

Figure 4.10 represents the test system, consisting of several (2, in the figure) testers.

Each Tester consists of these components:

- 1- Thread (InputSender): this thread sends inputs to the corresponding agent in interaction system (IS).
- 2- Thread (OutputReceiver): this thread receives outputs of port from the corresponding agent in IS.
- 3- An array of threads (VerdictSender[]): when a tester determines that its corresponding port in IUT is faulty this array of threads receives a “fail” verdict from Manage and then it sends it to all other testers, to stop running them,.
- 4- An array of threads (VerdictReceiver[]): this array of threads receives “fail” verdict from a tester in TS when the tester determines that its corresponding port is faulty and it inserts this verdict in LocalVerdictSender to be sent to TC.
- 5- An array of threads (CoordinationMSender[]): when Manage reaches a coordination message in LTS, it inserts that message in this array of threads to be sent to other testers in TS.

- 6- An array of threads (CoordinationMReceiver[]): this array of threads receives the coordination messages from other testers in TS and it inserts them in QueueOfOutput.
- 7- Queue (QueueOfOutput): outputs from OutputReceiver thread and CoordinationMReceiver[] thread are inserted in this queue, then it inserts them in Manage.
- 8- Thread (LocalVerdictSender): this thread sends the local verdict of tester to TC.
- 9- A loop (Manage): this component receives data (LTS, name of IUT and the number of ports running in IUT) from TC. And it inserts inputs in InputSender thread to be sent to the corresponding agent. And when it reaches the end of its LTS, it inserts the local verdict in LocalVerdictSender thread.

Figure 4.11 represents these components. The manage component in tester receives the followings from Test Controller:

- a- The corresponding local test sequence,
- b- The name of IUT,
- c- The number of ports, existing in IUT.

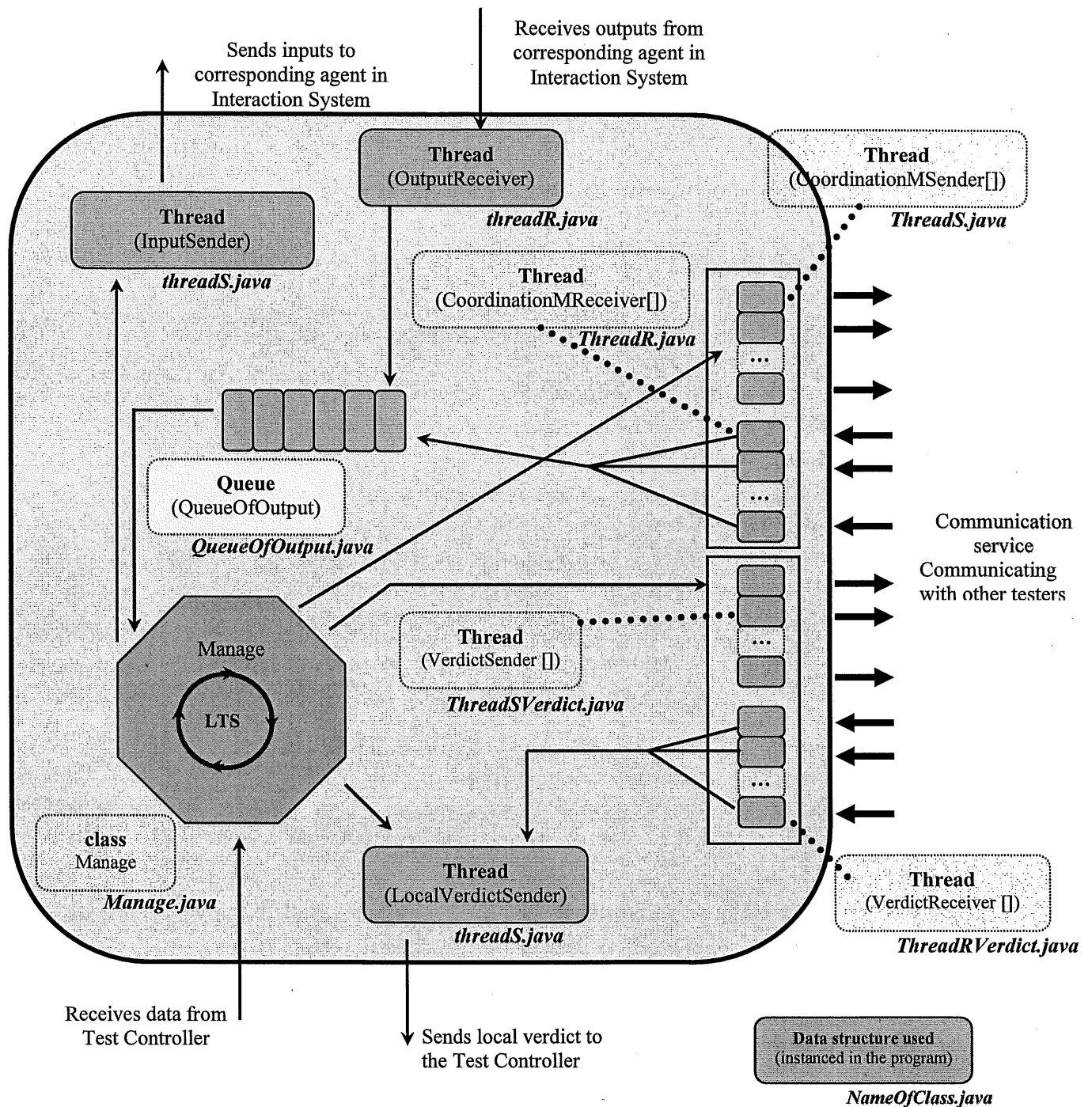


Figure 4.11 Components of a tester

Based on the number of ports existing in IUT, tester determines how many testers are running in the TS. For example if the number of ports, received by tester is X, then tester determines that

there are $X-1$ other testers (excluding itself) in TS. This information is necessary because it has to establish a connection with each of them.

As discussed before, there are four kinds of elements in the local test sequence:

- $!x$, sending of message x to the IUT,
- $?y$, receiving of message y from the IUT,
- $-C_{h_1, \dots, h_r}$, sending of coordination message C to testers h_1, \dots, h_r ,
- $+C_h$, receiving of coordination message C from tester h .

In the loop of Manage, the tester checks every single element of LTS, one by one, respecting the order, starting from the beginning of LTS.

If the element is $!x$, then it is an input for the IUT, so tester puts it in the InputSender thread, and this thread sends the element to the corresponding agent of IS.

If the element is $?y$, then it is an output from IS, so tester constantly checks the QueueOfOutput, until it is able to read the output.

If the element is $-C_h$ (i.e., coordination message to send), then tester puts it in CoordinationMSender[], and the thread sends the coordination message to tester h .

If the element is $+C_h$ (i.e., incoming coordination message from tester h), so tester constantly waits and checks the QueueOfOutput, until it is able to read the output.

Once a tester reaches the end of its LTS, which is being processed in the loop, and if it does not receive any incorrect output neither from its corresponding port in IUT nor from other testers, then it sends a “pass” verdict to the Test Controller.

As indicated in the figure 4.11, there are two arrays of threads, VerdictSender[] and VerdictReceiver[]. Here is the reason to create them:

Assume tester x has received a wrong output, due to a faulty IUT, from corresponding port in IUT, which is port x , and tester y is waiting to receive an output from IUT. Then tester x sends the verdict to Test Controller that IUT is found faulty, but we never receive any verdict from tester y . Tester x terminates running, but tester y is still running (and may other testers). So in this case tester x sends a message through communication service between testers, by VerdictSender[] to all other testers existing in the TS to inform them that IUT has been found faulty, then all other testers receive the message through communication service, by VerdictReceiver[] and then terminate their tasks, and send ‘fault’ verdict to TC. Besides, TC has to have all verdicts from all

testers to be able to decide the correctness of IUT, so in TC we must have a verdict from every single tester.

4.2.3 Interaction system (IS)

As we explained in the beginning of this chapter, we create this system to be an intermediary between TS and IUT. This system receives the inputs from test system and provides the IUT with these inputs, and receives the outputs from IUT, then provides testers with these outputs. This system is running locally with IUT, for example agent1 and port1 are running on the same machine. To explain the components of IS we consider the example where IUT receives from and sends to text files (see Fig. 4.3). Here are the components of IS:

1- Text file (fileOfInput.txt): this text file contains the input for port, written by agent of IS and read by port of IUT.

2- Text file (fileOfOutput.txt): this text file contains the output of port, written by port of IUT and read by agent of IS.

FileOfInput.txt is created by agent, and is accessed by both agent and port.

Agent writes inputs in this file and port reads inputs from this file. The latter is opened as write-able for agent and as readable for port.

FileOfOutput.txt is created by agent and is accessed by both port and agent.

Port writes outputs in this file and agent reads the outputs from this file. The latter is opened as write-able for port and readable for agent.

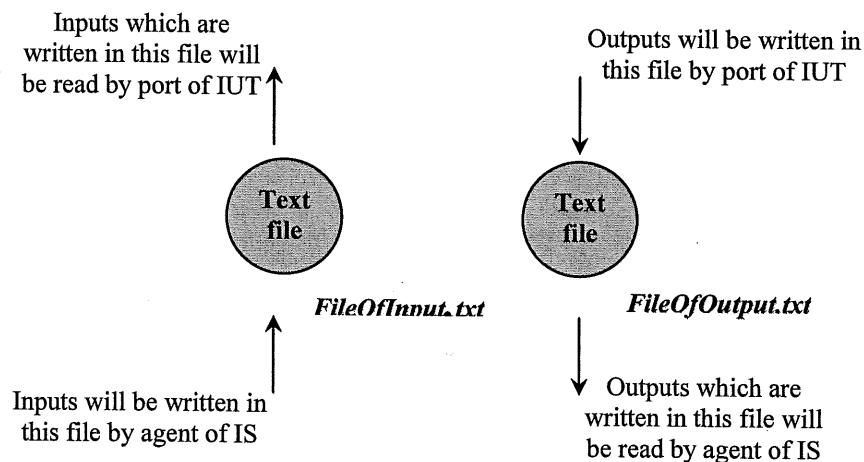


Figure 4.12 Text files between an agent and a port

3- Agent

The task of an agent is:

- a- Receiving input from tester and insert it in the FileOfInput.txt
- b- Reading the output of port from FileOfOutput.txt and send it to corresponding tester.

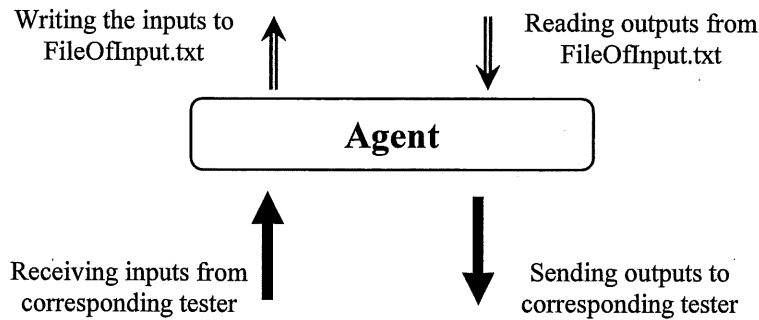


Figure 4.13 An agent

Each agent consists of these components:

- 1- Thread (InputReceiverFromTester): this thread receives the input from corresponding tester in Test System.
- 2- Thread (OutputSenderToTester): this thread sends the output of its corresponding port to its corresponding tester.
- 3- Thread (InputWriterToFile): this thread writes the input received from tester to FileOfInput.txt.
- 4- Thread (OutputReaderFromFile): this thread reads the output of corresponding port, from FileOfOutput.txt.
- 5- Queue (QueueOfInput): inputs from tester are inserted in this queue to be proceeded by component number 3 (InputWriterToFile)

These components are presented in figure 4.14.

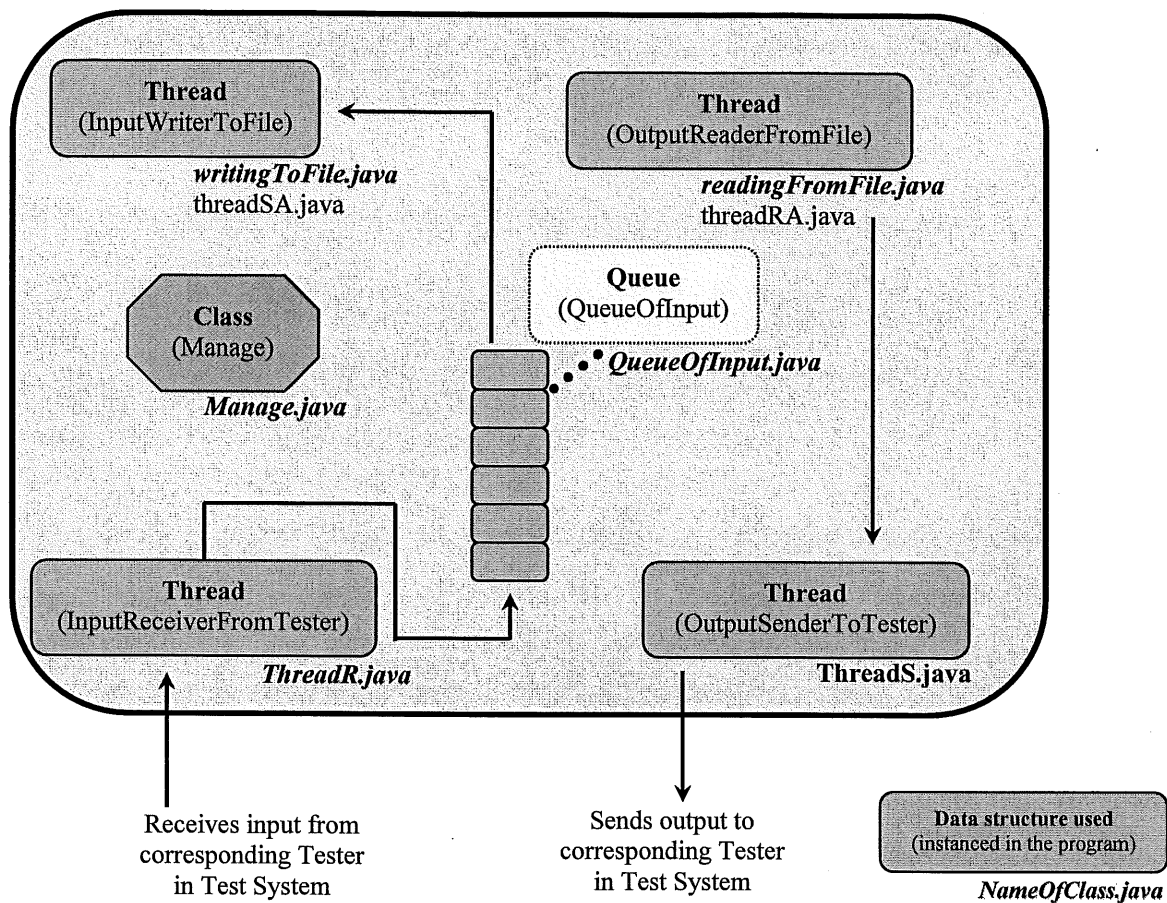


Figure 4.14 The components of an Agent

Thread 'InputReceiverFromTester' receives inputs through the communication service from tester, and inserts them in the 'QueueOfInput'. Thread of 'InputWriterToFile' dequeues inputs from 'QueueOfInput' and writes them in 'FileOfInput.txt'. Thread 'OutputReaderFromFile' reads outputs from 'FileOfOutput.txt' and inserts them in the thread 'OutputSenderToTester' and then thread 'OutputSenderToTester' sends the output to Tester through communication service.

4.2.4 IUT

As we mentioned before, normally the IUT exists and is given to be tested, and we do not know its internal structure. But here, the aim is to show how to create an IUT in order to apply and validate the TS. In our constructed IUT, we may insert some faults in order to check the ability of TS to detect these faults.

Each port in IUT is described by the followings:

- 1- Text file (PortX.atm, X represents the port number, for example for port number 1, this file is named 'Port1.atm')

This text file (Local text file) contains information to create a hash table, which is called PortTable. Step *d* of *constructing an IUT* (see Section 4.1.3) uses this file to create PortTable. To clarify, we consider the example 3.1. Figure 4.15 represents this text file for port number 1.

```
(Q10/T:1:a1)=(<T:1:x1,P:2:a1,P:3:a1>;Q11);
(Q11/P:2:b2)=(<T:1:x1>;Q12);
(Q11/P:3:c3)=(<T:1:x1>;Q12);
(Q12/P:2:b2)=(<ε>;Q13);
(Q12/P:3:c3)=(<ε>;Q13);
(Q13/T:1:a1)=(<T:1:x1,P:2:a1,P:3:a1>;Q14);
(Q14/P:3:c3)=(<T:1:w1>;Q14);
(Q14/P:2:b2)=(<T:1:x1>;Q14);
(Q14/T:1:a1)=(<T:1:x1,P:2:a1,P:3:a1>;Q15);
(Q15/P:3:c3)=(<ε>;Q10);
```

Figure 4.15 Represents text file (port1.atm) for port number 1 in

- 2- Hashtable (PortTable)

This table is created from the information existing in the portX.atm file.

Input	Output	Next State
Q10/T:1:a1	T:1:x1,P:2:a1,P:3:a1	Q11
Q11/P:2:b2	T:1:x1	Q12
Q11/P:3:c3	T:1:x1	Q12
Q12/P:2:b2	ε	Q13
Q12/P:3:c3	ε	Q13
Q13/T:1:a1	T:1:x1,P:2:a1,P:3:a1	Q14
Q14/P:3:c3	T:1:w1	Q14
Q14/P:2:b2	T:1:x1	Q14
Q14/T:1:a1	T:1:x1,P:2:a1,P:3:a1	Q15
Q15/P:3:c3	ε	Q10

Figure 4.16 Represents PortTable for port 1 in example 3.1

Let us explain the syntax of PortTable for the first line in Fig. 4.16:

Q10/T:1:a1	T:1:x1,P:2:a1,P:3:a1	Q11
------------	----------------------	-----

Current state:

Q10: is the current state,

Input: **T:1:a1**: is the input, read from text file

Output(s):

T:1:x1,P:2:a1,P:3:a1: are the outputs of the port, 'T' means that the output must be sent to tester and 'P' means that the output must be sent to the other port in IUT. (messages "P" will be called IUT internal messages).

T:1:x1 shows that, output x1 must be sent to the tester number 1,

P:2:a1, internal message a1 must be sent from port 1 to port number 2 in IUT,

P:3:a1, internal message a1 must be sent from port 1 to port number 3 in IUT.

Next state:

Q11: is the next state

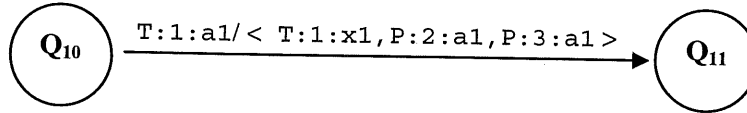


Figure 4.17 displays the two states and the transition in the first line of table

Port 1 is in the state Q_{10} and receives input a1 from tester 1, then sends output x1 to tester 1 and sends output a1 to two other ports in IUT, to inform them and let them know to decide what they should do in this state, and then goes to state Q_{11} .

- 3- Thread (InputReaderFromFile): this thread reads fileOfInput.txt in IS to pick the input written by agent and inserts it in QueueOfInput.
- 4- Thread (OutputWriterToFile): this thread writes the output of port in FileOfOutput.txt in interaction system.
- 5- An array of threads (OutputSenderToPorts[]): this array of threads sends IUT internal messages to other ports in IUT. IUT internal messages are messages, which are exchanged between ports in IUT.
- 6- An array of threads (OutputReceiverFromPorts[]): this array of threads receives IUT internal messages from other testers in IUT.
- 7- Thread (Manage): this thread is responsible to pick up the inputs from QueueOfInput and find the output by invoking a method on PortTable. This output might be one of these cases:
 - a- Output starts by 'T': this output must be sent to the corresponding tester in test system, in this case Manage inserts it in OutputWriterToFile thread.

b- Output starts by 'P': this output is an IUT internal message and must be sent to another port in IUT, in this case, Manage inserts it in appropriate thread in OutputSenderToPorts[].

These components are represented in Figure 4.18.

In a port, first, Manage thread creates PortTable from the Text file. Then InputReaderFromFile reads the input from FileOfInput.txt and inserts it into QueueOfInput. Then Manage removes the input from QueueOfInput and invokes a method on PortTable to find the output. Then based on the output, determines to send it either to other port (IUT internal message) or corresponding tester.

If output must be sent to other port, Manage inserts it in OutputSenderToPort[], and if it must be sent to corresponding tester, Manage inserts it in OutputWriterToFile to be written in FileOfOutput.txt.

Let us see what happens in practice. For the example 4.1, when we are in state Q_0 , port 1 is supposed to receive a from corresponding tester. InputReaderFromFile constantly checks FileOfInput.txt until it is able to read input a , then InputReaderFromFile inserts this input in QueueOfInput then Manage removes this input and invokes a method on PortTable and finds these outputs: T:1:x,P:2:a,P:3:a (see Fig. 4.7). As you see, there are three outputs:

T:1:x is an output for corresponding tester to port 1, so Manage inserts output x in OutputWriterToFile to be written in FileOfOutput.txt,

P:2:a is an IUT internal message to port 2, so Manage inserts IUT internal message a in OutputSenderToPorts to be sent to port 2 of IUT.

P:3:a is also an IUT internal message to port 3 and Manage does the same thing as P:2:a.

CHAPTER 5

CORBA AND ITS EVENT SERVICE

In this chapter, we introduce CORBA (Common Object Request Broker Architecture) and its properties, but before we get through CORBA, it is necessary to bring a short explanation about middleware. In section 5.2 we explain CORBA, in section 5.3 we introduce the event service of CORBA, and finally in section 5.4 we indicate the event services used in our architecture.

5.1 What is middleware? [11]

Traditionally, computing environments are heterogeneous, consisting of a mix of different hardware, operating systems, networking technologies, and programming languages. The reasons for this are usually historical: whatever was considered to be important or "best" at one time ended up being added to the computing environment. For example, a few years ago, Java did not exist, yet, there are hardly any computing environments left today without at least some code written in Java.

Heterogeneous environments will be with us for the foreseeable future; today's hot technology is yesterday's old hat, and companies are forever forced to embrace new technologies, such as Java, HTTP, XML, or ATM.

Conflicting with heterogeneous environments is the need for application integration. Different parts of the environment must be able to seamlessly communicate with each other, despite the fact that the different technologies are usually not designed to be integrated. The recent explosion of Java, e-commerce, and the Web made it abundantly clear that it is very difficult to be competitive unless a corporation's computing components are seamlessly integrated at all levels from the customer's web browser to the back-end stock control systems.

Application integration in a heterogeneous environment is extremely difficult. The wide variety of programming languages, networking technologies, operating systems, and other factors pose formidable technical problems, and the complexity and cost of developing custom integration solutions is commercially infeasible for all but the largest corporations.

Middleware addresses the integration challenge by providing a common communications substrate. Rather than an end in itself, middleware makes it possible for applications to easily communicate with each other and to be integrated into a coherent whole. In effect, middleware acts as "integration glue" and permits developers to focus on application logic instead of building communications infrastructure.

5.2 What is CORBA? [11]

CORBA (Common **O**bject **R**quest **B**roker **A**rchitecture) is the world's dominant middleware technology. It is available from many vendors for a vast number of hardware platforms, programming languages, and operating systems, and it is used for mission-critical applications in industries as diverse as manufacturing, telecommunications, banking, entertainment, and health care.

CORBA is an open platform standard, defined and made available free of charge by the Object Management Group (OMG). With over 800 members, the OMG is the world's largest software consortium and has just entered its eleventh year of operation. Unlike competing technologies, such as Microsoft's COM+, DCE, or Java RMI, CORBA is not tied to a specific vendor, platform, or programming language. Technology decisions in the OMG are driven by the needs of its members and not by the interests of one particular industry segment.

CORBA permits application integration at a much higher level of abstraction than other middleware technologies and requires little networking knowledge. This enables programmers who are not networking gurus to provide robust, efficient, and scalable integration solutions. CORBA not only enables the choice of programming language and operating system (permitting you to use the most appropriate tool), it also results in shorter development time, lower defect rates, and shorter time-to-market because it removes the chore of network programming from programmers and replaces it with a predictable and stable environment.

CORBA embraces technologies such as transaction processing, security and encryption, asynchronous messaging, Java, XML, metadata, portable devices, real-time systems, audio-visual streams, and fault-tolerance, and so represents the state of the art of distributed computing.

CORBA is in a unique position among middleware: whenever a new technology appears, OMG members move to provide access to it via CORBA. This results in the integration of new technologies into the overall CORBA architecture often long before other competing

technologies have even time to react. And, in contrast with other middleware, the OMG is an organisation without commercial interest in any particular technology or vendor and is therefore free to add even competing approaches (such as Microsoft's COM) to the overall architecture.

The CORBA approach of embracing new technology and lowering the threshold of entry of such technology into distributed computing makes it today's dominant and most widely deployed middleware technology. Its wide commercial availability, as well as Open Source implementations for Linux and other platforms, places it into the prime position to remain the dominant middleware technology for the next decade.

5.3 The OMG Event Service [7]

5.3.1 Introduction

All applications built with other OMG services are based on synchronous request invocations.

With synchronous requests, a client actively invokes requests on passive servers; after sending a request, the client blocks waiting for the response. Clients are aware of the destinations of requests because they hold object references to the target objects, and each request has a single destination denoted by the object reference used to invoke it. If the target object no longer exists or for some reason is unreachable, the invoking client receives an exception.

Many distributed applications find the synchronous request invocation model too restrictive despite its obvious utility. These applications generally require a means of decoupling the suppliers of information from the consumers interested in it. For example, in our climate control system we might want to have the thermometers send alarm messages if the temperature falls below or rises above a specified range, or we might want to be notified if a thermostat is set too high or too low. Making the thermometer and thermostat objects responsible for disseminating these messages to all interested parties unnecessarily complicates their implementations, and it scales poorly as the number of interested consumers rises.

The OMG Event Service provides support for decoupled communications between objects. It allows suppliers to send messages to one or more consumers with a single call. In fact, suppliers using an implementation of the Event Service need not be aware of any of the consumers of its messages; the Event Service acts as a mediator that decouples suppliers from consumers. An Event Service implementation also shields suppliers from exceptions resulting from any of the consumer objects being unreachable or poorly behaved.

5.3.2 Event Service Basics

In the OMG Event Service model, **suppliers** produce event and **consumers** receive them. Both suppliers and consumers connect to an **event channel**. An event channel conveys events from suppliers to consumers without requiring suppliers to know about consumers or vice versa. The event channel plays the central role in the Event Service. It is responsible for suppliers and consumer registration, timely and reliable event delivery to all registered consumers, and then handling of errors associated with unresponsive consumers.

The OMG Event Service provides two models for event delivery: the **push** model and the **pull** model. With the push model, suppliers push events to the event channel, and the event channel pushes events to consumers.

Figure 5.1 illustrates the push style of event delivery. Note that the arrows indicate the client and server roles and point from client to server.

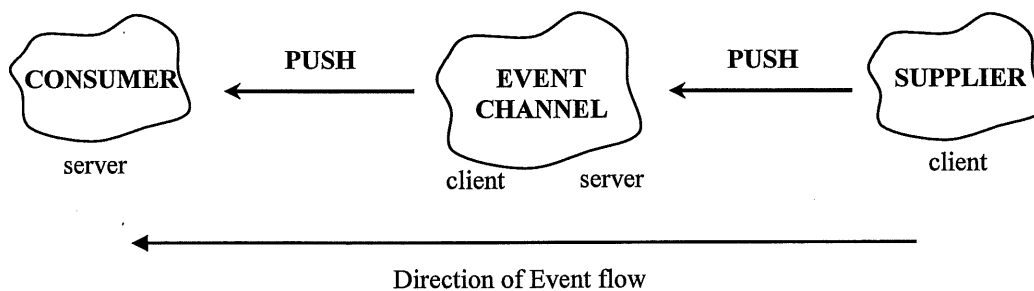


Figure 5.1 Push-style event delivery model

For the pull model, the actions that cause event flow occur in the opposite direction: consumers pull events from the event channel, and the event channel pulls events from suppliers. The pull model is shown in figure 5.2.

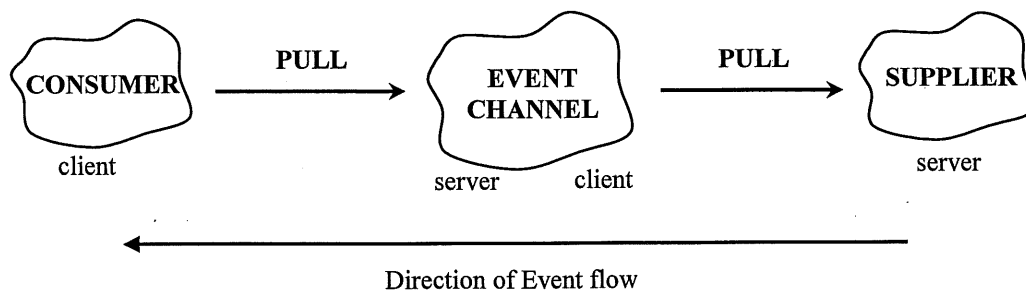


Figure 5.2 Pull-style event delivery model

Event channels allow multiple suppliers and consumers to be connected to them. Because some of them will want to use the push model, and others will want to use the pull model, event channels support four different models for event delivery:

- The *canonical push* model
- The *canonical pull* model
- The *hybrid push/pull* model
- The *hybrid pull/push* model

These models differ in whether suppliers and consumers are *active* or *passive* (that is, act as client or server).

Canonical Push Model

In this model, suppliers push events to the event channel, which in turn pushes them to all registered consumers.

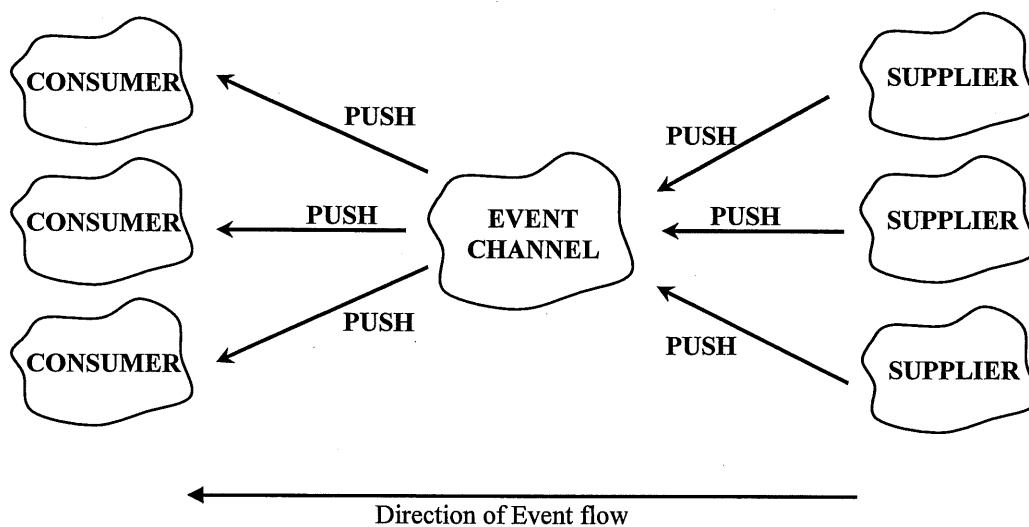


Figure 5.3 canonical push model

As illustrated in Figure 5.3, suppliers are thus the active initiators of events, whereas consumers passively wait to receive them.

Canonical Pull Model

In this model, consumers pull events from the event channel, which in turn pulls them from suppliers. As illustrated in Figure 5.4, consumers are the active initiators of events, and suppliers passively wait until events are pulled from them.

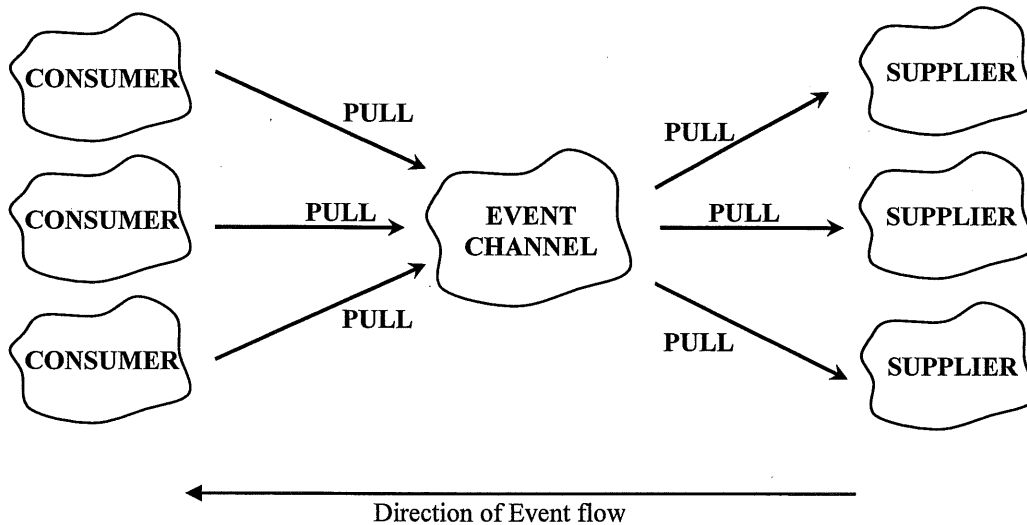


Figure 5.4 canonical pull model

Hybrid Push/Pull Model

In this model, suppliers push events to the event channel, where they are pulled by consumers. Thus, both suppliers and consumers are active in this model. The event channel plays the role of queue because it merely stores event data pushed by suppliers until it has been pulled by consumers.

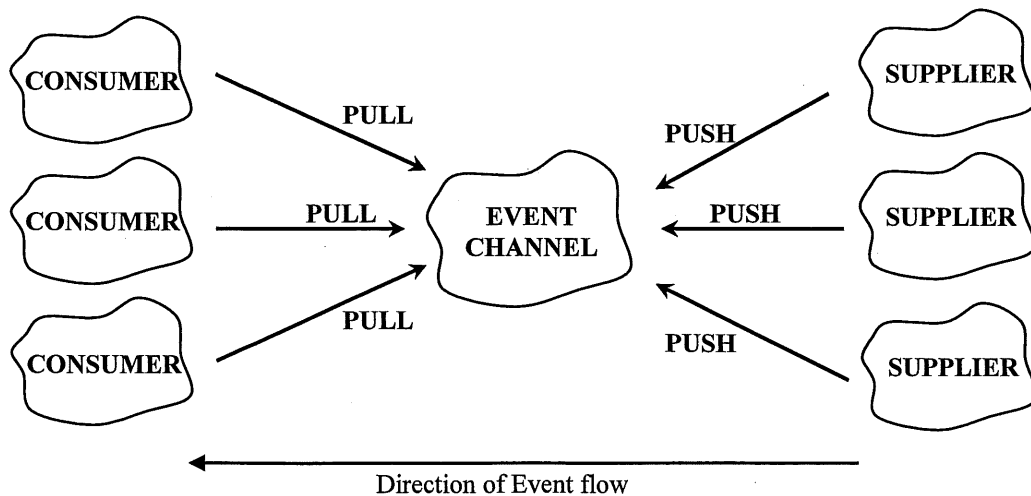


Figure 5.5 Hybrid push/pull model

Hybrid Pull/Push Model

In this model, event channels pull events from suppliers and push them to consumers. Both suppliers and consumers are passive in this model. The event channel plays the role of intelligent agent. The role is so named because the event channel must be capable of initiating the movement of all events in the system.

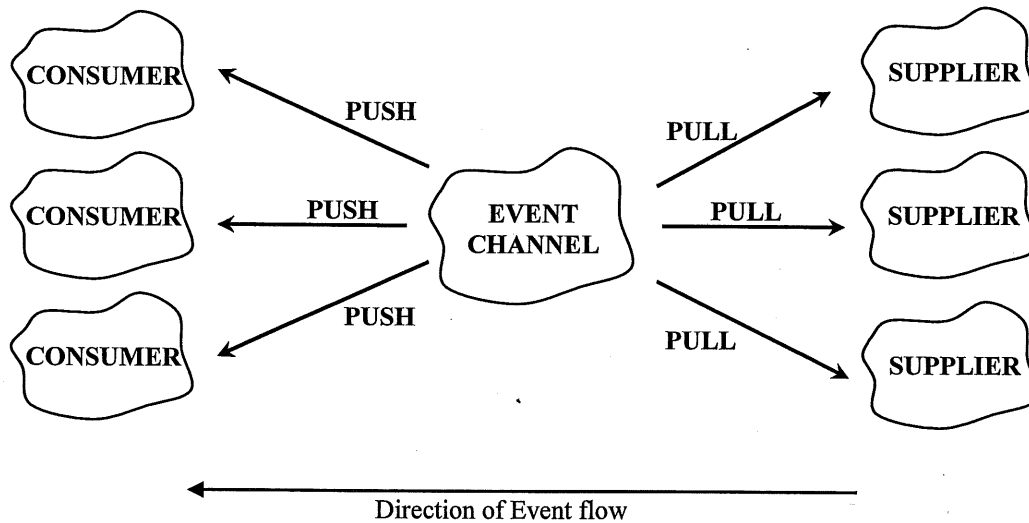


Figure 5.6 Hybrid pull/push model

Mixing Event Models

A single event channel can support all four models simultaneously, as shown in Figure 5.7. Here, a single event channel has attached to it two passive suppliers and one active supplier as well as a passive consumer and an active consumer. All four event delivery models are represented here.

- The relationship between the top consumer and the top supplier represents the canonical pull model.
- The relationship between the top consumer and the middle supplier represents the hybrid push/pull model.
- The relationship between the bottom consumer and the middle supplier represents the canonical push model.
- The relationship between the bottom consumer and the bottom supplier represents the hybrid pull/push model.

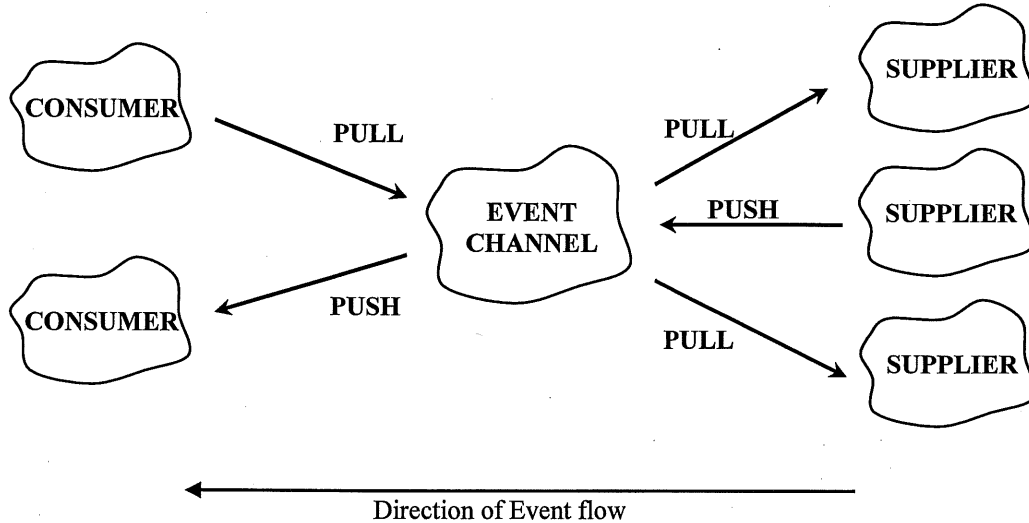


Figure 5.7 Mixing event delivery model

5.3.3 Event Service Limitations

Multiple Suppliers

Because multiple suppliers can connect to an event channel, consumers may end up receiving far more events than they are interested in. This is because event channels deliver all events to all consumers; each consumer receives all events from all suppliers connected to the same event channel.

Lack of Reliability

It is extremely important to keep in mind that event channels are fundamentally unreliable. Their lack of reliability stems from the difficulty of providing end-to-end guaranteed delivery in a service in which the channel has no way to throttle the supplier. If a supplier pushes so many events that the event channel cannot keep up with delivering all of them to its consumers, the event channel has no choice except to drop some of the events.

Lack of Filtering

Even if an event channel has only a single supplier connected to it, clients may still receive events in which they have no interest. This is because event channels pass events from their suppliers to their consumers without attempting to interpret event data in any way.

Lack of Factory Considerations

Event channels are CORBA objects, and , as with all other objects, you must create one before you can use it. You usually create objects using some sort of factory, either programmatically by invoking the factory from your application or manually by running a command-line program or a GUI-based tool.

The event service does not specify anything having to do with event channel factories. This behaviour allows each vendor that supplies event channel implementation complete freedom as to how it has you create and administer its event channels, but it also prevents you from easily writing portable event channel factories for your applications.

Asynchronous Messaging

In some cases, applications do not require decoupled communications; instead, they require *asynchronous messaging* or *time-independent invocation*. Asynchronous messaging allows an application to issue a request without blocking for the response; later, it receives the response either by a call-back from the ORB or by polling. With time-independent invocation, a client can make a request, disconnect from network, and then reconnect later and get the response. This is useful for application such as those that run on laptops or other portable computers. You can implement limited asynchronous messaging using the Dynamic Invocation Interface, but it is generally too cumbersome to use.

5.4 Event service in realization

In the next page, Figure 5.8 represents the event channels, which are responsible for the communication between different components of the architecture presented in section 4.1.2.

There is an event channel between:

- 1- Test Controller and each tester to send the LTS, name of IUT, and number of ports in IUT.
- 2- Each tester and its corresponding Agent in interaction system to send the inputs.
- 3- Each tester and each one of other testers in the test system to exchange coordination messages.
- 4- Each tester and each one of the other testers in the test system to exchange 'fail' verdict in case when a tester determines its corresponding port is faulty.
- 5- Each agent and its corresponding tester in the test system to send outputs.
- 6- Each port and each one of other ports in IUT to exchange internal messages.

7- Each tester and Test Controller to send the local verdict.

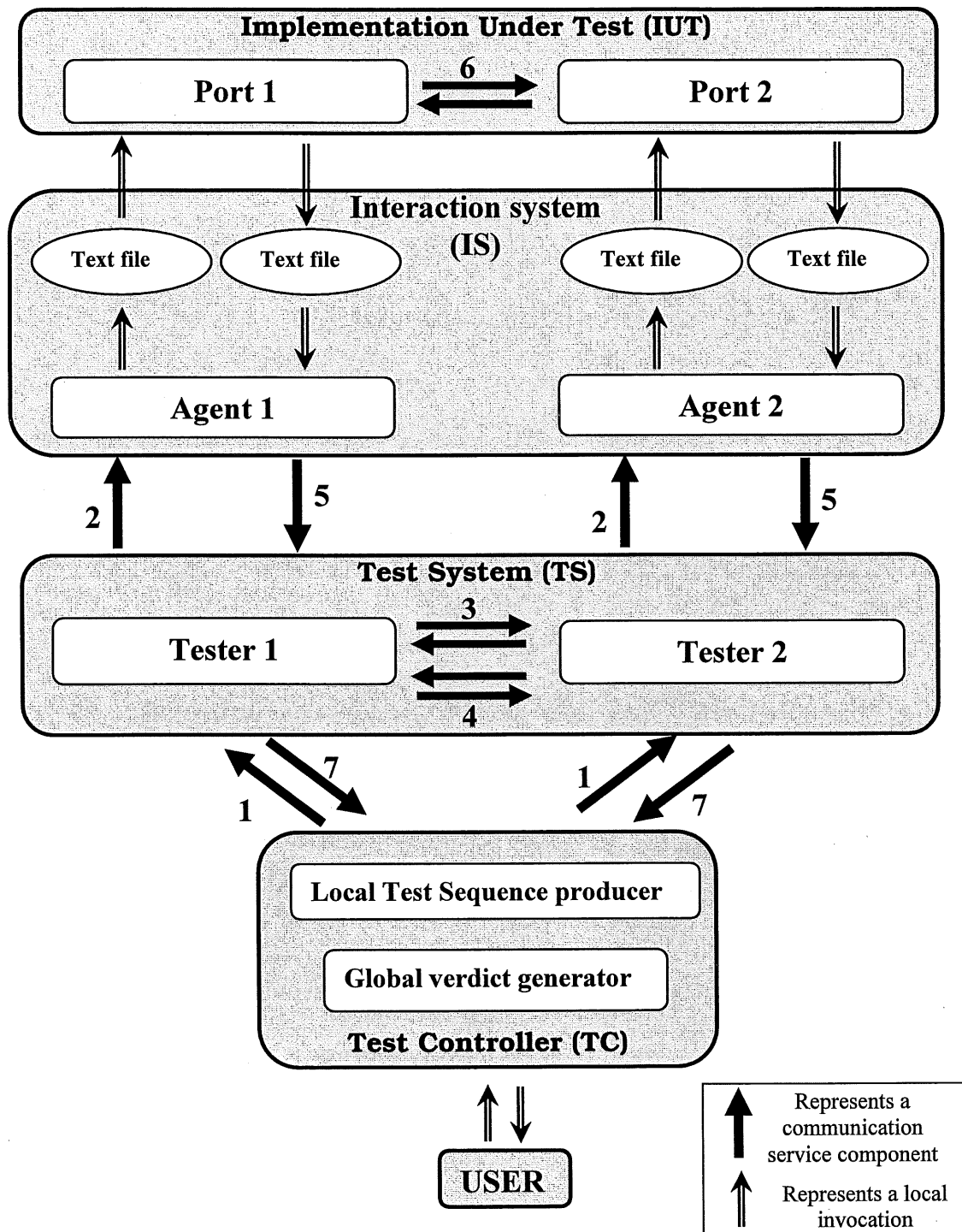


Figure 5.8 Representing the event channels

The number of event channels in a program depends on the number of ports in the IUT. If we have X Ports in the IUT, then we will have $3X^2 + X$ running event channels after the program is

run. For example if there are 2 ports in IUT, then there would be 14 event channels running in the realization. Here is the method to compute the formula introduced above. Let X be the number of ports in the IUT, then:

There are:

2X event channels between TC and TS, since we have 2 event channels between each tester and TC,

$2X^2 - 2X$ between testers in TS, since we have 4 event channels between each two testers,

2X between TS and IS, since we have two event channels between each tester and corresponding agent in IS,

$X^2 - X$ between ports in IUT, since we have 2 event channels between each two testers in IUT, which totally comes to $3X^2 + X$ event channels in the whole architecture.

For example if we have 4 ports in IUT, then we have:

8 event channels between TC and TS, 24 event channels between testers in TS, 8 event channels between TS and IS and 12 event channels between ports in IUT, which totally comes to 52 event channels in whole architecture.

All event channels used in the realization are created using canonical push model, so suppliers push events to the event channel, which in turn pushes them to all registered consumers.

In the following we show the code for creating an event channel:

```
Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");

// the code for supplier
try
{
    //
    // Create ORB.
    //
    ORB orb = ORB.init(args, props);

    //
    // Get event channel
    //
    EventChannel e = GetEventChannel(orb, channelName);

    //
    // Get ProxyPushConsumer.
    //
    SupplierAdmin supplierAdmin = e.for_suppliers();
```

```

ProxyPushConsumer consumer = supplierAdmin.obtain_push_consumer();

//
// Since callbacks are not needed a PushSupplier_impl is not
// created
//
try
{
    consumer.connect_push_supplier(null);
}
catch(AlreadyConnected ex)
{
    ex.printStackTrace();
    System.exit(1);
}
}

// the code for consumer
try
{
    //
    // Create ORB.
    //
    ORB orb = ORB.init(args, props);

    //
    // Get event channel
    //
    EventChannel e = GetEventChannel(orb, channelName);

    //
    // Get ProxyPullSupplier.
    //
    ConsumerAdmin consumerAdmin = e.for_consumers();
    ProxyPullSupplier supplier = consumerAdmin.obtain_pull_supplier();

    //
    // Since callbacks are not needed a PushSupplier_impl is
    // not created.
    //
    try
    {
        supplier.connect_pull_consumer(null);
    }
    catch(AlreadyConnected ex)
    {
        ex.printStackTrace();
        System.exit(1);
    }
}

// this is the method to obtain the event channel
GetEventChannel(org.omg.CORBA.ORB orb, String name)
{
    EventChannel e = null;
    if(name == null)
    {
        //
        // Get event channel from the initial reference.
        //
        try

```



```

    {
        org.omg.CORBA.Object obj =
            orb.resolve_initial_references("EventService");
        e = EventChannelHelper.narrow(obj);
    }
    catch(org.omg.CORBA.ORBPackage.InvalidName ex)
    {
        System.err.println("can't resolve `EventService'");
        System.exit(1);
    }

    if(e == null)
    {
        System.err.println("'EventService' is not an EventChannel " +
            " object reference");
        System.exit(1);
    }
}
else
{
    //
    // Get event channel factory. Note that this uses ORBacus
    // proprietary methods.
    //
    com.ooc.OBEventChannelFactory.EventChannelFactory f = null;
    try
    {
        org.omg.CORBA.Object obj =
            orb.resolve_initial_references("EventChannelFactory");
        f = com.ooc.OBEventChannelFactory.EventChannelFactoryHelper.
            narrow(obj);
    }
    catch(org.omg.CORBA.ORBPackage.InvalidName ex)
    {
        System.err.println("Can't resolve `EventChannelFactory'");
        System.exit(1);
    }

    if(f == null)
    {
        System.err.println("'EventChannelFactory' is not an " +
            "EventChannelFactory object reference");
        System.exit(1);
    }

    //
    // Get event channel.
    //
    try
    {
        e = f.get_channel_by_id(name);
    }
    catch(com.ooc.OBEventChannelFactory.ChannelNotAvailable ex)
    {
        try
        {
            e = f.create_channel(name);
            System.err.println("Channel " + name + " created");
        }
        catch(com.ooc.OBEventChannelFactory.ChannelAlreadyExists ex1)
        {
            System.err.println("Channel exists: try again");
        }
    }
}

```

```
        System.exit(1);
    }
}
return e;
}
```

CHAPTER 6

CLASSES AND INTERFACES USED IN THE CORBA-BASED REALIZATION OF THE PROPOSED TEST ARCHITECTURE

In this chapter, we introduce the functions of the different interfaces and classes used in the realisation program. The classes and interfaces are organized in three folders as follows:

- 1- TestController
- 2- TestSystem
- 3- IUT & IS

The structure of the folders and files is introduced in the figure 6.1.

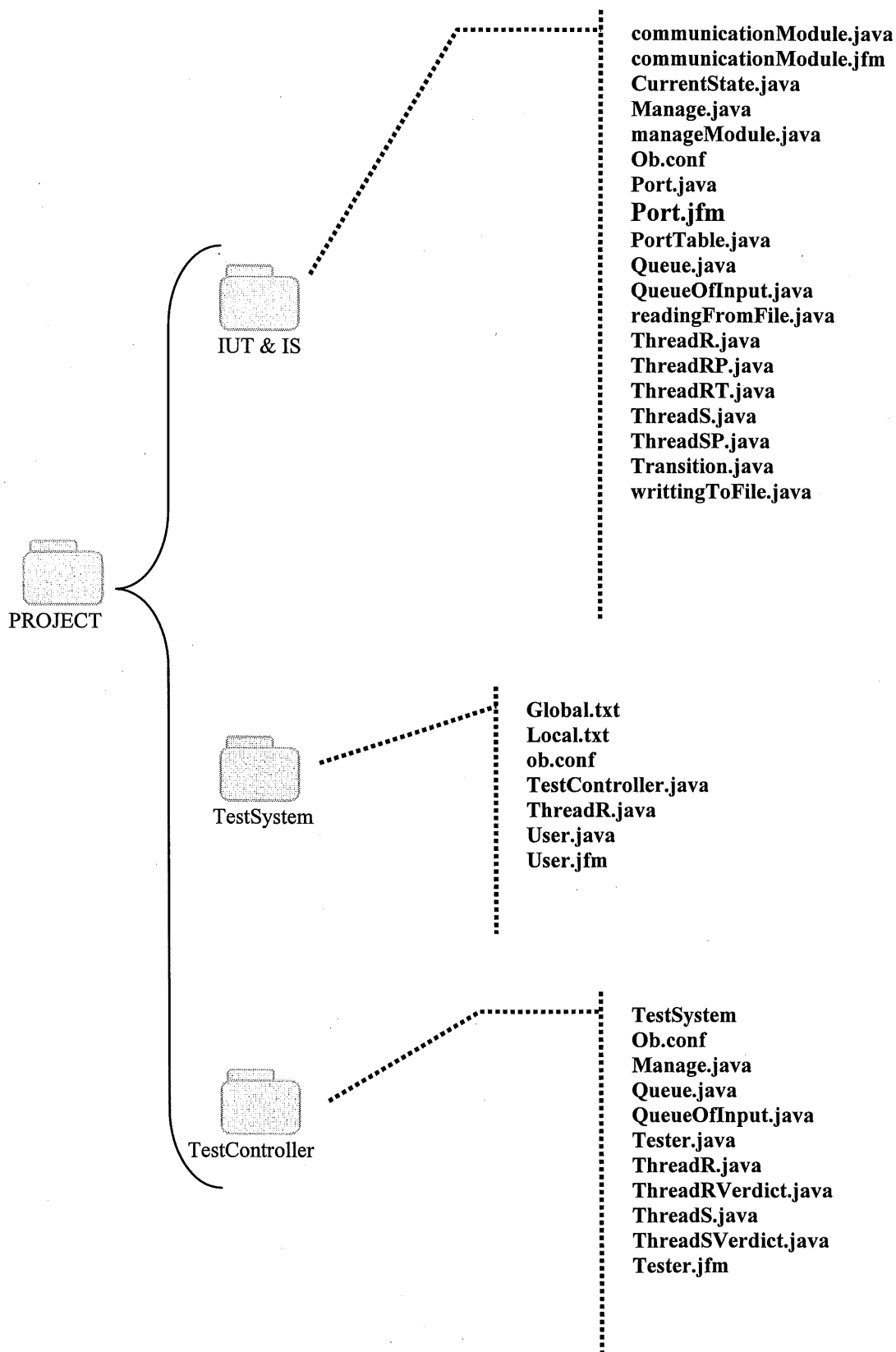


Figure 6.1 The structure of folders and files of realisation program

6.1 Classes of the Test Controller

User.java : /* This method implements the user interface introduced in Fig. 6.1 */

public class User extends Frame

```
{  
    public User(String [] args)  
    {  
        /* Constructor */  
    }  
    public void InitialPositionSet()  
    {  
        /* this method initializes the position of user interface of Fig. 6.2 */  
    }  
    public boolean handleEvent(Event evt)  
    {  
        /* this method handles the function of buttons on the user interface of Fig. 6.2, by clicking  
        on 'Quit' or 'start' an event is created which is argument of this method */  
    }  
    public static void main(String args[])  
    {  
        /* this is the main method and makes the user interface of Fig. 6.2 */  
    }  
    public void User_WindowDestroy(Object target)  
    {  
        /* this method is invoked by 'handleEvent()' method and will disappear the user interface  
        of Fig. 6.2 if the user clicks on "Quit" button on the interface */  
    }  
    public void Start_Action(Object target)  
    {  
        /* this method is invoked by 'handleEvent()' method and handles the process which must  
        be done by clicking on "start" button on the user interface of Fig. 6.2 */  
    }  
}
```

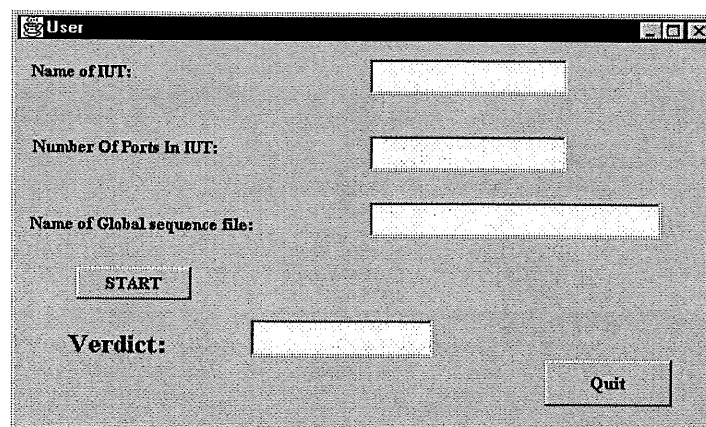


Figure 6.2 User Interface displayed by running User.java

```

TestController.java : /* this method contains all methods to produce the LTSs from GTS, and to produce the
public class TestController                                final verdict */
{
    public void handle(String args[])
    {
        /* this method creates the event channels between TC and TS and also sends the
        following data to testers: name of IUT, number of ports in IUT, and LTSs */
    }
    public void getVerdicts()
    {
        /* this method invokes the “readVerdict” of threadR to read the Local Verdicts */
    }
    public String finalVerdict()
    {
        /* This method corresponds to “Global Verdict Generator” in figure 4.9. it generates the
        Global Verdict from Local Verdicts */
    }
    public void produce()
    {
        /* This method corresponds to the “Local Test Sequence Producer” in figure 4.9. It
        produces the LTSs from GTS.
    }
    public static int[] port( String []array, int i )
    {
        /* This method is part of procedure of producing LTSs. For example, this method returns
        1 for this input: !X : 1, which means port number 1 */
    }
    public static int[] difference (int [] array1, int [] array2)
    {
        /* This method is part of procedure of producing LTSs. This methods accepts two integer
        arrays, and return the difference between them ( $C=A-B$ ), for example, for these two arrays,
         $A=\{1,2,3\}$ ,  $B=\{1,2\}$ , it returns  $C=\{3\}$  */
    }
    public static int[] merge (int [] array1, int [] array2)
    {
        /* This method is part of procedure of producing LTSs. This method accepts two integer
        arrays, merges them ( $C = A \cup B$ ) and returns it. For example, for these two integer arrays,
         $A=\{1,2,3,4\}$ ,  $B=\{1,3,5,6\}$ , it returns this array  $C=\{1,2,3,4,5,6\}$  */
    }
    public static int[] symmetricalDefference ( int [] array1,int[] array2)
    {
        /* This method is part of procedure of producing LTSs. This method accepts two integer
        arrays and returns the symmetrical difference between them ( $C = (A-B) \cup (B-A)$ ) */
    }
}

```

```

public static String[] devideOutput (String [] output, int index)
{
    /* This method is part of procedure of producing LTSs. This method accepts an array of
    string, and an integer, this integer is an index for the string array. For example if this is the input
    for this method "{x:1,y:2}", this would be the output "?", "x:1", "y:2".
    }
public static String[] produceLocalSequences
                                (String oneLine, int numberOfTesters)
{
    /* This method accepts the global test sequence and number of testers as arguments, and
    then produces the local test sequences by using the methods introduced above */
}
static private EventChannel GetEventChannel
                                (org.omg.CORBA.ORB orb, String name)
{
    /* This method obtains an event channel. This method accepts two values as arguments, 1-
    orb that is obtained from orb class of CORBA, 2- and a string, which is a name for the event
    channel */
}
}

```

```

ThreadR.java : /* This thread receives the local verdicts from testers */
public class ThreadR extends Thread
{
    public ThreadR()
    {
        /* Constructor */
    }
    public String readVerdict()
    {
        /* This method returns the local verdict received by this thread */
    }
    public void run()
    {
        /* This method is invoked once the thread is started */
    }
}

```

6.2 Classes of the Test System

Tester.java : /* This method implements the user interface introduced in Fig. 6.2*/

public class Tester extends Frame

```
{  
    public Tester(String [] args)  
    {  
        /* Constructor */  
    }  
    public void InitialPositionSet()  
    {  
        /* This method initializes the position of user interface of Fig. 6.3 */  
    }  
    public boolean handleEvent(Event evt)  
    {  
        /* This method handles the function of buttons on the user interface of Fig. 6.3, by  
        clicking on 'Quit' or 'start' an event is created which is argument of this method */  
    }  
    public static void main(String args[])  
    {  
        /* This is the main method and makes the user interface of Fig. 6.3 */  
    }  
    public void Tester_WindowDestroy(Object target)  
    {  
        /* This method is invoked by 'handleEvent()' method and will disappear the user interface  
        of Fig. 6.3 if the user clicks on "Quit" button on the interface */  
    }  
    public void Start_Action(Object target)  
    {  
        /* This method is invoked by 'handleEvent()' method and handles the process which must  
        be done by clicking on "start" button on the user interface of Fig. 6.3 */  
    }  
}
```

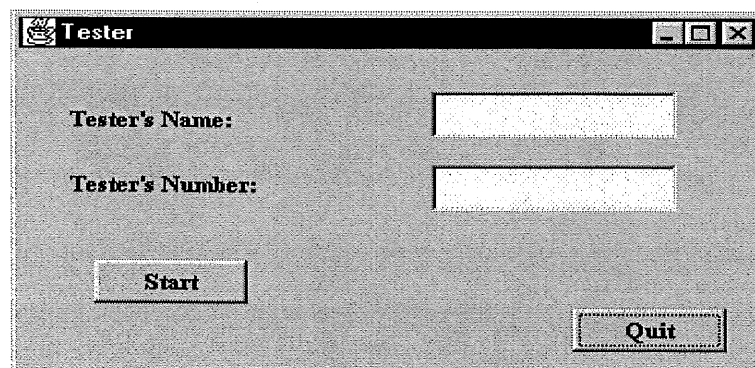


Figure 6.3 User Interface displayed by running Tester.java

Manage.java : /* This class contains the methods to create event channels between TS and TC, and between TS and IS */

```
public class Manage
{
    public Manage ()
    {
        /* Constructor */
    }
    public void general(String [] args)
    {
        /* This method created the event channels between test system (TS) and test controller (TC), and between TS and IS, and also receives data like "name of IUT", "number of ports in IUT" and LTS from TC */
    }
    public void startThreads()
    {
        /* This method starts all threads in a tester */
    }
    public void handle()
    {
        /* This method processes the LTS, element by element and decides what to do with each element */
    }
    static private EventChannel GetEventChannel
        (org.omg.CORBA.ORB orb, String name)
    {
        /* This method obtains an event channel. This method accepts two values as arguments, 1- orb that is obtained from orb class of CORBA, 2- and a string, which is a name for the event channel */
    }
}
```

Queue.java : /* standard queue interface */

```
public interface Queue
{
    /* Standard queue interface */
}
```

QueueOfInput.java : /* This class contains the methods to create a queue for inputs and methods to handle the queue */

```
public class QueueOfInput implements Queue
{
    public QueueOfInput()
    {
        /* Constructor */
    }
    public synchronized void enqueue( Object objectToQueue )
    {
        /* This method inserts an element in queue, this method is synchronized which means it
        doesn't let any other method to access the queue while it is manipulating it. */
    }
    public synchronized String dequeue()
    {
        /* This method removes an element from queue, this method is synchronized which
        means it doesn't let any other method to access the queue while it is manipulating it. */
    }
}
```

ThreadR.java : /* This method receives the coordination messages from other testers in TS */

public class ThreadR extends Thread

```
{
    public ThreadR()
    {
        /* Constructor */
    }
    public void run()
    {
        /* This method is invoked once the thread is started */
    }
}
```

ThreadRVerdict.java : /* This method extends Thread. It receives 'fail' verdict from a tester, which has its corresponding port in IUT is faulty. */

```
public class ThreadRVerdict extends Thread
{
    public ThreadRVerdict()
    {
        /* Constructor */
    }
    public void run()
    {
        /* This method is invoked once the thread is started */
    }
}
```

ThreadS.java : /* This method sends the coordination messages to other testers in TS *

public class ThreadS extends Thread

```
{
    public ThreadS()
    {
        /* Constructor */
    }
    public void writeInput(String )
    {
        /* This method is invoked by 'Manage' to insert a coordination message, which must be
        sent by this thread */
    }
    public void run()
    {
        /* This method is invoked once the thread is started */
    }
}
```

ThreadSVerdict.java : /* This method extends Thread. It sends 'fail' verdict to all other testers in TS to stop them from running, when it detects its corresponding port in IUT is faulty. */

public class ThreadSVerdict extends Thread

```
{
    public ThreadSVerdict()
    {
        /* Constructor */
    }
    public void writeInput(String in)
    {
        /* This method is invoked to insert the 'fail' verdict, which must be sent by this thread to
        all other testers */
    }
    public void run()
    {
        /* This method is invoked once the thread is started */
    }
}
```

6.3 Classes of Interaction System (IS) and IUT

Agent.java: /* This method implements the user interface of Fig. 6.4*/

public class Agent extends Frame

{

public Agent(String [] args)

{

/* Constructor */

}

public void InitialPositionSet()

{

/* This method initializes the position of user interface of Fig. 6.4 */

}

public boolean handleEvent(Event evt)

{

/* This method handles the function of buttons on the user interface of Fig. 6.4, by clicking on 'Quit' or 'start' an event is created which is argument of this method */

}

public static void main(String args[])

{

/* This is the main method and makes the user interface of Fig. 6.4 */

}

public void Agent_WindowDestroy(Object target)

{

/* This method is invoked by 'handleEvent()' method and will disappear the user interface of Fig. 6.4 if the user clicks on "Quit" button on the user interface */

}

public void Start_Action(Object target)

{

/* This method is invoked by 'handleEvent()' method and handles the process which must be done by clicking on "start" button on the user interface of Fig. 6.4 */

}

}

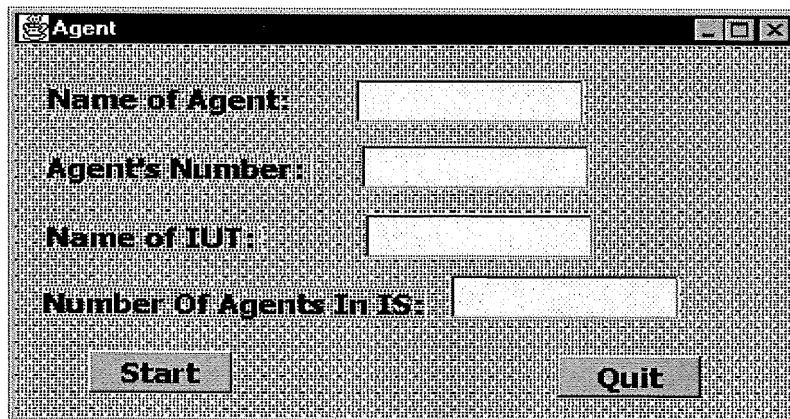


Figure 6.4 User Interface displayed by running Agent.java

ManageA.java : /* This class contains methods to create threads in IS and to create the event channel between IS and TS, */

```
public class ManageA
{
    public ManageA ()
    {
        /* Constructor */
    }
    public void general(String [] args)
    {
        /* this method creates all threads in IS and also the event channels between interaction
        system (IS) and test system (TS) */
    }
    public void startThreads()
    {
        /* this method starts all threads in IS */
    }
    static private EventChannel GetEventChannel
                                   (org.omg.CORBA.ORB orb, String name)
    {
        /* This method obtains an event channel. This method accepts two values as arguments, 1-
        orb that is obtained from orb class of CORBA, 2- and a string which is a name for the event
        channel */
    }
}
```

ThreadRA.java : /* This method reads inputs from FileOfOutput.txt and inserts them in threadS to be sent to the corresponding tester of TS */

```
public class ThreadRA extends Thread
{
    public ThreadRA(String numberOfPort)
    {
        /* Constructor */
    }
    public void run()
    {
        /* This method is invoked once thread is run */
    }
}
```

ThreadSA.java : /* This thread removes inputs from QueueOfInput and writes them in FileOfInput.txt */

public class ThreadSA extends Thread

```
{
    public ThreadSA()
    {
        /* Constructor */
    }
    public void run()
    {
        /* This method is invoked once thread is run */
    }
}
```

Port.java : /* This method implements the user interface of Fig. 6.4 */

public class Port extends Frame

```
{
    public Port(String [] args)
    {
        /* Constructor */
    }
    public void InitialPositionSet()
    {
        /* This method initializes the position of user interface introduced in fig. 6.4 */
    }
    public boolean handleEvent(Event evt)
    {
        /* This method handles the function of buttons on the user interface of Fig. 6.5, by
        clicking on 'Quit' or 'start' an event is created which is argument of this method */
    }
    public static void main(String args[])
    {
        /* This is the main method and makes the user interface of Fig. 6.5 */
    }
    public void Port_WindowDestroy(Object target)
    {
        /* This method is invoked by 'handleEvent()' method and will disappear the user interface
        of Fig. 6.5 if the user clicks on "Quit" button on the user interface */
    }
    public void Start_Action(Object target)
    {
        /* This method is invoked by 'handleEvent()' method and handles the process which must
        be done by clicking on "start" button on the user interface of Fig. 6.5 */
    }
}
```

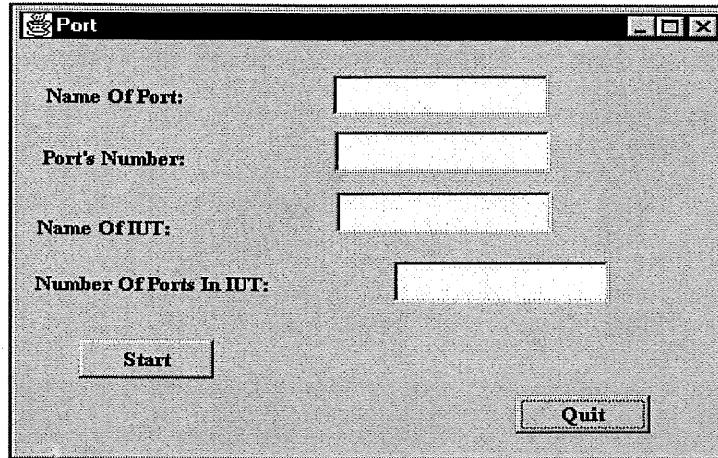


Figure 6.5 User interface by running Port.java

ManageP.java : /* This class contains methods to create PortTable in a port */

public class ManageP extends Thread

```
{
    public ManageP ()
    {
        /* Constructor */
    }
    public void general(String [] args)
    {
        /* This method creates the PortTable */
    }
    public void run()
    {
        /* This method is invoked once the Manage thread is run */
    }
}
```

CurrentState.java : /* This class contains the methods to set up the initial state and update the current state */

public class CurrentState

```
{
    public CurrentState()
    {
        /* Constructor */
    }
    protected void changeCurrentState( String newCurrentState )
    {
        /* This method changes the current state */
    }
}
```

PortTable.java : /* This class contains methods to read the open the general text file and fill out the PortTable */

```
public class PortTable extends Transition
{
    public PortTable(String numberOfPort )
    {
        /* Constructor */
    }
    private BufferedReader openFileAutomator()
    {
        /* This method opens the general text file (see fig. 4.5) */
    }
    public int fillPortTable ()
    {
        /* This method fills the PortTable, which is hash table */
    }
    String findNextState( String key )
    {
        /* This method finds the next state in PortTable */
    }
    String findOutputs( String key )
    {
        /* This method finds the output in PortTable */
    }
}
```

Queue.java : /* standard queue interface */

```
public interface Queue
{
    /* Standard queue interface */
}
```

QueueOfInput.java : /* This calls contains the methods to create a queue for inputs and methods to handle the queue */

```
public class QueueOfInput implements Queue
{
    public QueueOfInput()
    { /* Constructor */ }
    public synchronized void enqueue( Object objectToQueue )
    {
        /* This method inserts an element in queue, this method is synchronized which means it
        doesn't let any other method to access the queue while it is manipulating it. */
    }
    public synchronized String dequeue()
    {
        /* This method removes an element from queue, this method is synchronized which
        means it doesn't let any other method to access the queue while it is manipulating it. */
    }
}
```


ThreadR.java : /* This method receives internal message from other ports in IUT */

public class ThreadR extends Thread

```
{  
    public ThreadR()  
    {  
        /* Constructor */  
    }  
    public void run()  
    {  
        /* This method is invoked once thread is run */  
    }  
}
```

ThreadRP.java : /* This thread reads inputs from FileOfInput and inserts them in QueueOfInput */

public class ThreadRP extends Thread

```
{  
    public ThreadRP(String numberOfPort)  
    {  
        /* Constructor */  
    }  
    public void run()  
    {  
        /* This method is invoked once thread is run */  
    }  
}
```

ThreadS.java : /* This method sends internal messages to other ports in IUT */

public class ThreadS extends Thread

```
{  
    public ThreadS()  
    {  
        /* Constructor */  
    }  
    public void writeInput(String in)  
    {  
        /* This method is invoked by Manage to write the internal message into this thread */  
    }  
    public void run()  
    {  
        /* This method is invoked once thread is run */  
    }  
}
```

ThreadSP.java : /* This method extends Thread. It inserts the outputs of a port in FileOfOutput.txt in IS */

public class ThreadSP extends Thread

```
{
    public ThreadSP(String numberOfPort)
    {
        /* Constructor, this constructor receives numberOfPort which is the port number */
    }
    public void write(String output)
    {
        /* This method is invoked by Manage to write the output in this thread */
    }
    public void run()
    {
        /* This method is invoked once thread is run */
    }
}
```

Transition.java : /* This method implements a transition when a PortTable is being constructed */

public class Transition

```
{
    public Transition()
    {
        /* Constructor */
    }
    protected void writeOutputs( String outputs )
    {
        /* This method permits an output to be written in the transition */
    }
    protected void writeNextState( String nextState )
    {
        /* This method permits the next state of a transition to be written */
    }
}
```

We run the event service of CORBA either on a separate machine than the ones, which TC, testers, ports and agents are running on, or one of these machines. Then we obtain the IP address of this machine.

In addition to classes introduced above, each of the three folders contains some other files as follows.

TestController:

User.jfm : this file contains information for the user interface introduced in Figure 6.2.

ob.conf: this file contains the following two lines to obtain an event channel of the event service of CORBA:

```
ooc.service.EventService=iioploc://hostname:portnumber/DefaultEventChannel  
ooc.service.EventChannelFactory=iioploc://hostname:portnumber/DefaultEventChannelFactory
```

hostname is the IP address of the machine which even service is running on and portnumber is an arbitrary port which is not engaged by any program.

TestSystem:

Tester.jam: this file contains graphical information for the user interface introduced in Figure 6.3.

Ob.conf: the same thing as above.

IUT & IS:

Agent.jfm: this file contains graphical information for interface introduced in figure 6.4.

Port.jfm: this file contains graphical information for interface introduced in figure 6.5.

Ob.conf: the same thing as above.

CHAPTER 7

EXAMPLE OF APPLICATION

In this chapter, we introduce an example of application of our test architecture.

7.1 Simplified X.25

7.1.1 Introduction to X.25 protocol [8]

X.25 is a communication protocol consisting of the lowest three levels of the OSI reference model. Services are offered to the user through the network layer. Globally, the X.25 protocol allows two sites, Site_i and Site_j of the network to communicate. After the two sites have established a connection, they can exchange data. The communication between them is stopped when one of the two sites initiates a disconnection. A site can send a message at any moment without waiting for an acknowledgement. Two kinds of data are supported: normal and express data. Each of the two kinds of data are transmitted according to a FIFO discipline. But the FIFO discipline is not respected between the two kinds of data since express data may be received before normal data, which were sent before.

In order to give the possibility to both sites to establish a connection, we have used a mechanism of tokens to realize a distributed choice.

7.1.2 Simplified service provided by X.25 protocol

To simplify the X.25 protocol, we assume that: (i) a new message cannot be sent before the last one is received; and (ii) express data are not supported.

We remind the meaning of service and protocol. A service describes what is observed by the user and the corresponding protocol describes a realization of the service. And service primitives are events seen by user, and they are used to describe the service provided to the user.

Now we present the primitives used in the different phases of the simplified X.25.

Let U_1 and U_2 be two users of the network who are located in Site_1 and Site_2 , respectively.

The following service primitives are defined:

- *Connection* : A connection may be established between U_1 and U_2 if one of them, for instance U_1 , sends a Connect request (CN.req) to U_2 and then U_2 receives a Connect indication (CN.ind). When the latter receives a Connect indication, it may answer either by a Disconnect request (DC.req) to reject the Connection request, or by a Connect response (CN.rsp). In the first case, U_1 receives a Disconnect indication (DC.ind), while in the second case U_1 receives a Connect confirm (CN.cnf).
- *Disconnection* : A disconnection primitive can be used either to reject a Connect request, or to terminate an existing connection. For instance, U_1 may send a Disconnect request (DC.req) and then U_2 will receive a Disconnect indication (DC.ind).
- *Data Transfer* : this primitive allows to transfer data in both directions between two sites linked by a connection. To simplify the example, we assume that only the party which has initiated the connection can send data. The sending of a message is generated by a Data request (DT.req) and its reception by a Data indication (DT.ind).
- *Re-initialization* : The re-initialization procedure allows to restore the synchronization between two parties. When a Re-Initialization request (RI.req) is generated, for instance by U_1 , then all the data being transmitted in the medium are removed. The next element to be received by U_2 is a Re-Initialization indication (RI.ind). U_2 answers by a Re-Initialization response (RI.rsp) and then U_1 will receive a Re-Initialization confirm (RI.cnf). We assume that the party, which requests the re-initialization, is the sender of data.

7.1.3 Formal specification of the simplified X.25 service

The formal specification of the simplified X.25 service (i.e., the service provided by the simplified X.25 protocol) contains principally two blocs $S_{1,2}$ and $S_{2,1}$, where $S_{i,j}$ models the service when Site_{*i*} and Site_{*j*} are the sender and the receiver, respectively. The event Token_{*i*}^{*j*} means that "Site_{*i*} gives to Site_{*j*} the possibility to establish a connection".

Figure 7.1 represents the specification of a bloc $S_{i,j}$, i.e., specification of the simplified X.25 service when site *i* is the sender and site *j* is the receiver.

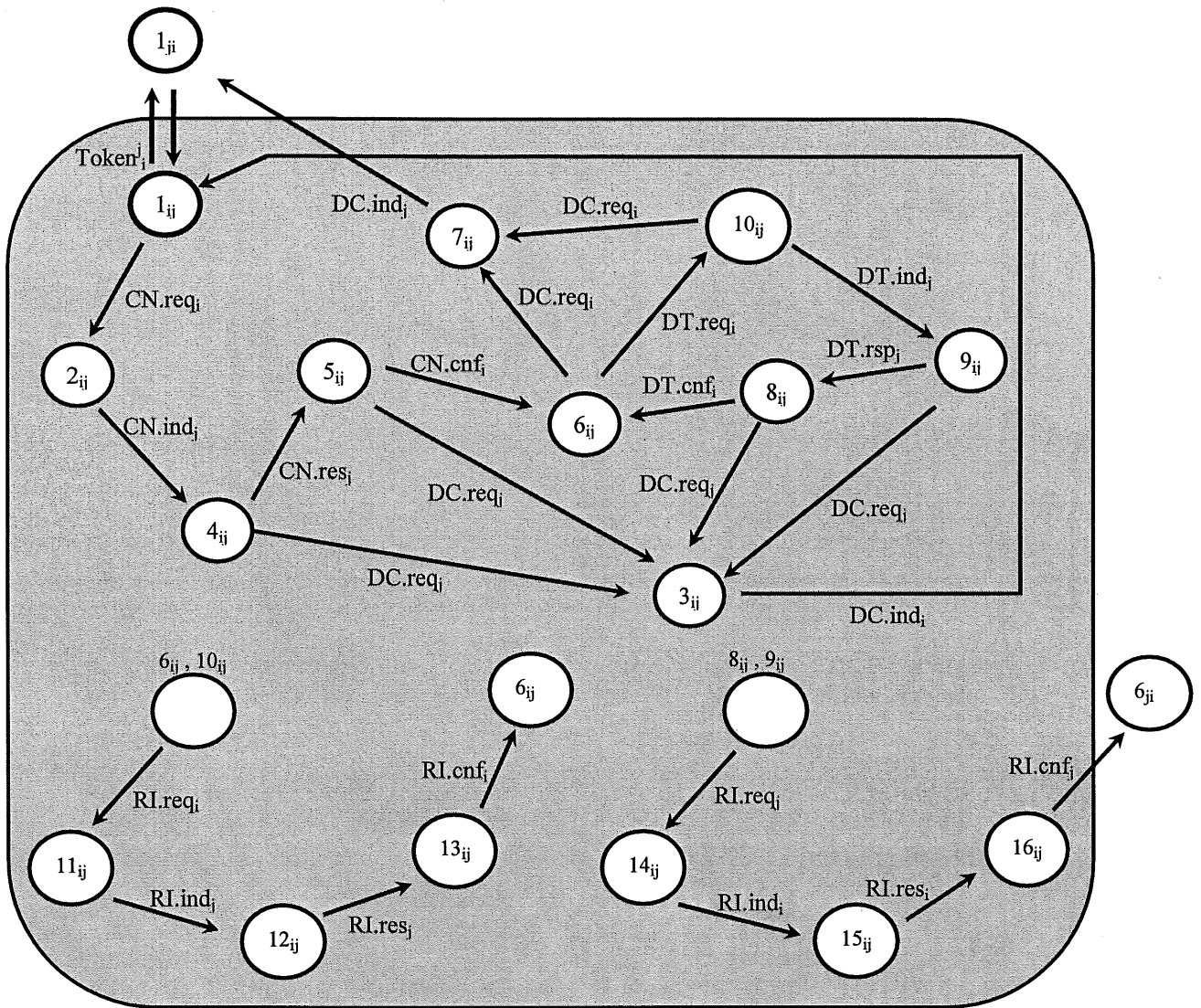


Figure 7.1 Specification of S_{ij} , i.e., service specification of the simplified X.25, where Site $_i$ is the sender and Site $_j$ is the receiver

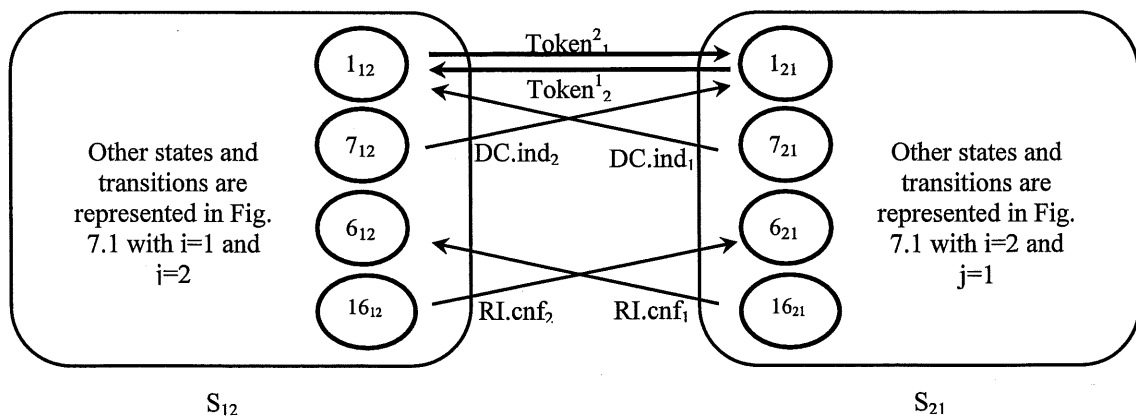


Figure 7.2 Global service specification of X.25

Figure 7.2 represents the global specification and shows the transitions between the two blocks $S_{i,j}$ and $S_{j,i}$.

In figure 7.2 the initial state is 1_{12} , which means $Site_1$ has the possibility to establish a connection and also can give this possibility to $Site_2$ by sending $Token^2_1$. In the latter case we switch from 1_{12} (the initial state of S_{12}) to 1_{21} (initial state of S_{21}).

In state $7_{i,j}$, $Site_i$ requests for disconnection and then we go to state $1_{j,i}$, the initial state of $S_{j,i}$.

In state $16_{i,j}$, $Site_i$ requests for re-initialization and then we go to state $6_{j,i}$. In this state, $Site_j$ has the possibility to request for either data by sending by sending $DT.req_2$ or disconnection by sending $DC.req_2$ to $Site_j$.

Transition $7_{i,j} \longrightarrow 1_{j,i}$ and $16_{i,j} \longrightarrow 6_{j,i}$ mean that when a site requests a disconnection or a re-initialization, then the token is implicitly given to the other site.

7.2 IUT Construction

We will illustrate the procedure of construction of an IUT (see section 4.1.3) with the example of Fig. 7.3.

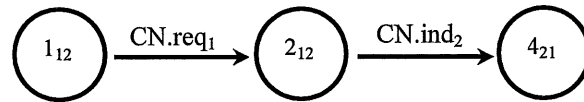
Step a: in this step we are about to design a FSM. As we can see, the specification presented in figure 7.2 is not the same as input/output FSM we introduced in chapter 2 (see definition and example 2.1), the np-FSM does not support multi-input transition, but in this example we have to consider this as an exception in order to be able to describe X.25 protocol specification by np-FSM. So we design an input/output FSM for the specification introduced in figure 7.2. This FSM is represented in figure 7.3. It is designed to show the inputs and outputs for both sites in every state. q_{10} is the initial state, in this state $Site_1$ has the possibility to establish a connection or give this possibility to $Site_2$. Transitions in this FSM are described as follows:

(input for $Site_1$, input for $Site_2$) / (output for $Site_1$, output for $Site_2$)

To clarify this format, we explain the transition between q_{10} and q_{11} of fig. 7.3, this is the transition: $(CN.req_1, \epsilon) / (\epsilon, CN.ind_2)$. In this transition $Site_1$ (port 1) receives $CN.req_1$ from tester 1 and $Site_2$ (port 2) receives nothing from tester 2, then as outputs, port 1 sends nothing to tester 1 and port 2 sends $CN.ind_2$ to tester 2.

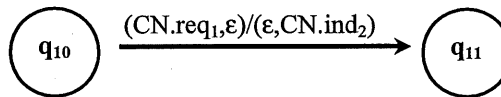
Here, to clarify how the FSM of Figure 7.3 is obtained from service specification (figure 7.1), we choose again the transition, which is explained above.

In figure 7.1, we start from the initial state 1_{12} (i & j are replaced by 1 & 2), where Site₁ has the possibility to establish a connection or give this possibility to Site₂, and choose the two first transition, which are represented below:



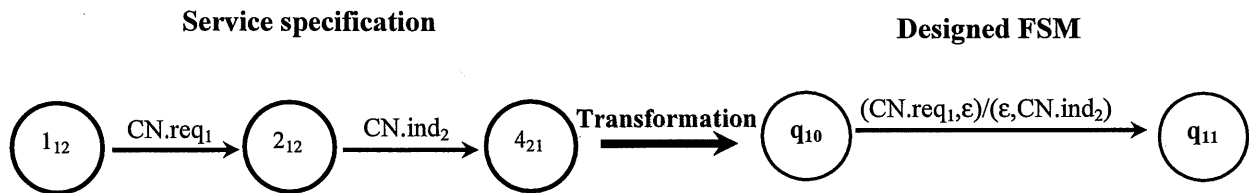
We are in state 1_{12} , Site₁ sends CN.req₁ to Site₂ to request a connection, and then we go to state 2_{12} , then Site₂ indicates that it has received a connection request from Site₁ and we go to state 4_{21} .

These two transitions are transformed to one single transition in the FSM (figure 7.3) as below:

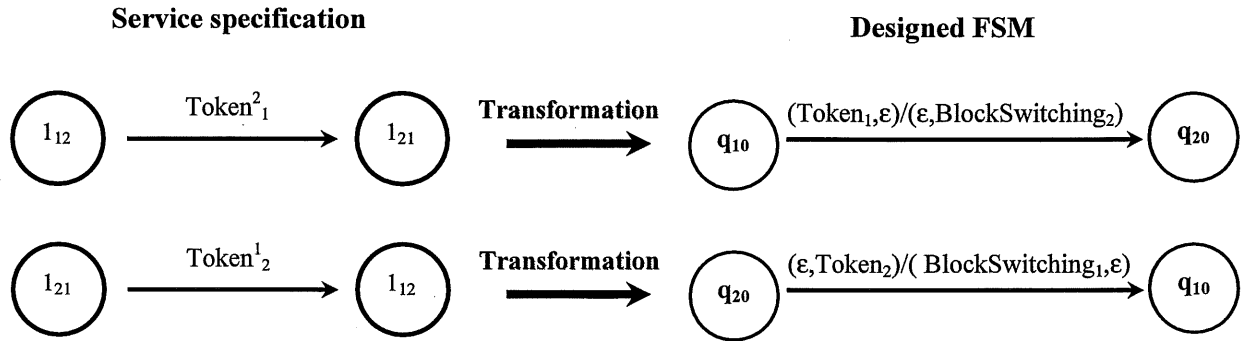


Tester 1 sends CN.req₁ to port 1 (Site₁) while tester 2 sends nothing (ϵ) to port 2 (Site₂), and then tester 2 receives CN.ind₂ from port 2 (Site₂) while tester 1 receives nothing (ϵ) from port 1 (Site₁).

This transformation is shown below.



As we can see, each two transition of specification will be transformed into one transition in the designed FSM, except in one case where a site has the possibility to establish a connection but gives this possibility to other site where we switch the block on the FSM. This case is shown below:



Step b:

We create the global text file that describes the FSM obtained in step a. Figure 7.4 represents this global text file. The syntax format of this text file is explained in 4.1.3.

```
(Q10/T:1:CN.req1)=( < T:2:CN.ind2 >;Q11);
(Q10/T:1:Token1)=( < T:2:BlockSwitching2>;Q20);
(Q11/ T:2:CN.rsp2)=( <T:1:CN.cnf1>;Q12);
(Q11/ T:2:CN.rsp2 & T:2:DC.req2)=( <T:1:DC.ind1>;Q10);
(Q11/T:2:DC.req2)=( <T:1:DC.ind1>;Q10);
(Q12/T:1:DT.req1)=( < T:2:DT.ind2>;Q14);
(Q12/T:1:DT.req1 & T:1:RI.req1)=( < T:2:RI.ind2>;Q13);
(Q12/T:1:RI.req1)=( < T:2:RI.ind2>;Q13);
(Q12/T:1:DT.req1 & T:1:DC.req1)=( < T:2:DC.ind2>;Q20);
(Q12/T:1:DC.req1)=( < T:2:DC.ind2>;Q20);
(Q13/T:2:RI.rsp2)=( <T:1:RI.cnf1>;Q12);
(Q14/T:2:DT.rsp2)=( <T:1:DT.cnf1>;Q12);
(Q14/T:2:DC.req2)=( <T:1:DC.ind1>;Q10);
(Q14/T:2:RI.req2)=( <T:1:RI.ind1>;Q15);
(Q14/T:2:DT.rsp2 & T:2:RI.req2)=( <T:1:RI.ind1>;Q15);
(Q15/T:1:RI.rsp1)=( < T:2:RI.cnf2>;Q22);
(Q20/T:2:Token2)=( < T:1:BlockSwitching1>;Q10);
(Q20/T:2:CN.req2)=( < T:1:CN.ind1>;Q21);
(Q21/ T:1:CN.rsp1)=( <T:2:CN.cnf2>;Q22);
(Q21/ T:1:DC.req1)=( <T:2:DC.ind2>;Q20);
(Q21/T:1:CN.rsp1 & T:1:DC.req1)=( < T:2:DC.ind2>;Q20);
(Q22/T:2:DT.req2)=( < T:1:DT.ind1>;Q24);
(Q22/T:2:RI.req2)=( < T:1:RI.ind1>;Q23);
(Q22/T:2:DT.req2 & T:2:RI.req2)=( < T:1:RI.ind1>;Q23);
(Q22/T:2:DC.req2)=( < T:1:DC.ind1>;Q10);
(Q22/T:2:DT.req2 & T:2:DC.req2)=( < T:1:DC.ind1>;Q10);
(Q23/ T:1:RI.rsp1)=( <T:2:RI.cnf2>;Q22);
(Q24/ T:1:DT.rsp1)=( <T:2:DT.cnf2>;Q22);
(Q24/ T:1:RI.req1)=( <T:2:RI.ind2>;Q25);
(Q24/T:1:DT.rsp1 & T:1:RI.req1)=( <T:2:RI.ind2>;Q25);
(Q24/ T:1:DC.req1)=( <T:2:DC.ind2>;Q20);
(Q25/T:1:DT.req1 & T:1:DC.req1)=( <T:2:DC.ind2>;Q12);
```

Figure 7.4 The global text file

Step c:

We create a local text file for each port; each local text file describes the protocol in the corresponding port. Figure 7.5 represents these two local text files for port 1 and port 2. The syntax format of these files is explained in 4.1.3.

```
(Q10/T:1:CN.req1)=(<P:2:CN.req1>;Q11);
(Q10/T:1:Token1)=(<P:2:Token1>;Q20);
(Q11/P:2:CN.rsp2)=(<T:1:CN.cnf1>;Q12);
(Q11/P:2:DC.req2)=(<T:1:DC.ind1>;Q10);
(Q12/T:1:DT.req1)=(<P:2:DT.req1>;Q14);
(Q12/T:1:DT.req1 & T:1:RI.req1)=(<P:2:RI.req1>;Q13);
(Q12/T:1:RI.req1)=(<P:2:RI.req1>;Q13);
(Q12/T:1:DT.req1 & T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q12/T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q13/P:2:RI.rsp2)=(<T:1:RI.cnf1>;Q12);
(Q14/P:2:DT.rsp2)=(<T:1:DT.cnf1>;Q12);
(Q14/P:2:DC.req2)=(<T:1:DC.ind1>;Q10);
(Q14/P:2:RI.req2)=(<T:1:RI.ind1>;Q15);
(Q15/T:1:RI.rsp1)=(<P:2:RI.rsp1>;Q22);
(Q20/P:2:Token2)=(<T:1:BlockSwitching1>;Q10);
(Q20/P:2:CN.req2)=(<T:1:CN.ind1>;Q21);
(Q21/T:1:CN.rsp1)=(<P:2:CN.rsp1>;Q22);
(Q21/T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q21/T:1:CN.rsp1 & T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q22/P:2:DT.req2)=(<T:1:DT.ind1>;Q24);
(Q22/P:2:RI.req2)=(<T:1:RI.ind1>;Q23);
(Q22/P:2:DC.req2)=(<T:1:DC.ind1>;Q10);
(Q23/T:1:RI.rsp1)=(<P:2:RI.rsp1>;Q22);
(Q24/T:1:DT.rsp1)=(<P:2:DT.rsp1>;Q22);
(Q24/T:1:RI.req1)=(<P:2:RI.req1>;Q25);
(Q24/T:1:DT.rsp1 & T:1:RI.req1)=(<P:2:RI.req1>;Q25);
(Q24/T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q24/T:1:DT.rsp1 & T:1:DC.req1)=(<P:2:DC.req1>;Q20);
(Q25/P:2:RI.rsp2)=(<T:1:RI.cnf1>;Q12);
```

The local text file for port 1

```
(Q10/P:1:CN.req1)=(<T:2:CN.ind2>;Q11);
(Q10/P:1:Token1)=(<T:2:BlockSwitching2>;Q20);
(Q11/T:2:CN.rsp2)=(<P:1:CN.rsp2>;Q12);
(Q11/T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q11/T:2:CN.rsp2 & T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q12/P:1:DT.req1)=(<T:2:DT.ind2>;Q14);
(Q12/P:1:RI.req1)=(<T:2:RI.ind2>;Q13);
(Q12/P:1:DC.req1)=(<T:2:DC.ind2>;Q20);
(Q13/T:2:RI.rsp2)=(<P:1:RI.rsp2>;Q12);
(Q14/T:2:DT.rsp2)=(<P:1:DT.rsp2>;Q12);
(Q14/T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q14/T:2:DT.rsp2 & T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q14/T:2:RI.req2)=(<P:1:RI.req2>;Q15);
(Q14/T:2:DT.rsp2 & T:2:RI.req2)=(<P:1:RI.req2>;Q15);
(Q15/P:1:RI.rsp1)=(<T:2:RI.cnf2>;Q22);
(Q20/T:2:Token2)=(<P:1:Token2>;Q10);
(Q20/T:2:CN.req2)=(<P:1:CN.req2>;Q21);
(Q21/P:1:CN.rsp1)=(<T:2:CN.cnf2>;Q22);
(Q21/P:1:DC.req1)=(<T:2:DC.ind2>;Q20);
(Q22/T:2:DT.req2)=(<P:1:DT.req2>;Q24);
(Q22/T:2:RI.req2)=(<P:1:RI.req2>;Q23);
(Q22/T:2:DT.req2 & T:2:RI.req2)=(<P:1:RI.req2>;Q23);
(Q22/T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q22/T:2:DT.req2 & T:2:DC.req2)=(<P:1:DC.req2>;Q10);
(Q23/P:1:RI.rsp1)=(<T:2:RI.cnf2>;Q22);
(Q24/P:1:DT.rsp1)=(<T:2:DT.cnf2>;Q22);
(Q24/P:1:RI.req1)=(<T:2:RI.ind2>;Q25);
(Q24/P:1:DC.req1)=(<T:2:DC.ind2>;Q20);
(Q25/T:2:RI.rsp2)=(<P:1:RI.res2>;Q12);
```

The local text file for port 2

Figure 7.5 The local text files

Step d: we create the port tables for each port, by using the local text files. Figure 7.6 represents these port tables.

The port table for port 1

Input	Output	Next State
Q10/T:1:CN.req1	P:2:CN.req1	Q11
Q10/T:1:Token1	P:2:Token1	Q20
Q11/P:2:CN.rsp2	T:1:CN.cnf1	Q12
Q11/P:2:DC.req2	T:1:DC.ind1	Q10
Q12/T:1:DT.req1	P:2:DT.req1	Q14
Q12/T:1:DT.req1 & T:1:RI.req1	P:2:RI.req1	Q13
Q12/T:1:RI.req1	P:2:RI.req1	Q13
Q12/T:1:DT.req1 & T:1:DC.req1	P:1:b2,T:2:y2:P:3:b2	Q20
Q12/T:1:DC.req1	P:2:DC.req1	Q20
Q13/P:2:RI.rsp2	P:2:DC.req1	Q12
Q14/P:2:DT.rsp2	T:1:RI.cnf1	Q12
Q14/P:2:DC.req2	T:1:DT.cnf1	Q10
Q14/P:2:RI.req2	T:1:DC.ind1	Q15
Q15/T:1:RI.rsp1	T:1:RI.ind1	Q22
Q20/P:2:Token2	P:2:RI.rsp1	Q10
Q20/P:2:CN.req2	T:1:BlockSwitching1	Q21
Q21/T:1:CN.rsp1	T:1:CN.ind1	Q22
Q21/T:1:DC.req1	P:2:CN.rsp1	Q20
Q21/T:1:CN.rsp1 & T:1:DC.req1	P:2:DC.req1	Q20
Q22/P:2:DT.req2	P:2:DC.req1	Q24
Q22/P:2:RI.req2	T:1:DT.ind1	Q23
Q22/P:2:DC.req2	T:1:RI.ind1	Q10
Q23/T:1:RI.rsp1	T:1:DC.ind1	Q22
Q24/T:1:DT.rsp1	P:2:RI.rsp1	Q22
Q24/T:1:RI.req1	P:2:DT.rsp1	Q25
Q24/T:1:DT.rsp1 & T:1:RI.req1	P:2:RI.req1	Q25
Q24/T:1:DC.req1	P:2:RI.req1	Q20
Q24/T:1:DT.rsp1 & T:1:DC.req1	P:2:DC.req1	Q20
Q25/P:2:RI.rsp2	T:1:RI.cnf1	Q12

The port table for port 2

Input	Output	Next State
Q10/P:1:CN.req1	T:2:CN.ind2	Q11
Q10/P:1:Token1	T:2:BlockSwitching2	Q20
Q11/T:2:CN.rsp2	P:1:CN.rsp2	Q12
Q11/T:2:DC.req2	P:1:DC.req2	Q10
Q11/T:2:CN.rsp2 & T:2:DC.req2	P:1:DC.req2	Q10
Q12/P:1:DT.req1	T:2:DT.ind2	Q14
Q12/P:1:RI.req1	T:2:RI.ind2	Q13
Q12/P:1:DC.req1	T:2:DC.ind2	Q20
Q13/T:2:RI.rsp2	P:1:RI.rsp2	Q12
Q14/T:2:DT.rsp2	P:1:DT.rsp2	Q12
Q14/T:2:DC.req2	P:1:DC.req2	Q10
Q14/T:2:DT.rsp2 & T:2:DC.req2	P:1:DC.req2	Q10
Q14/T:2:RI.req2	P:1:RI.req2	Q15
Q14/T:2:DT.rsp2 & T:2:RI.req2	P:1:RI.req2	Q15
Q15/P:1:RI.rsp1	T:2:RI.cnf2	Q22
Q20/T:2:Token2	P:1:Token2	Q10
Q20/T:2:CN.req2	P:1:CN.req2	Q21
Q21/P:1:CN.rsp1	T:2:CN.cnf2	Q22
Q21/P:1:DC.req1	T:2:DC.ind2	Q20
Q22/T:2:DT.req2	P:1:DT.req2	Q24
Q22/T:2:RI.req2	P:1:RI.req2	Q23
Q22/T:2:DT.req2 & T:2:RI.req2	P:1:RI.req2	Q23
Q22/T:2:DC.req2	P:1:DC.req2	Q10
Q22/T:2:DT.req2 & T:2:DC.req2	P:1:DC.req2	Q10
Q23/P:1:RI.rsp1	T:2:RI.cnf2	Q22
Q24/P:1:DT.rsp1	T:2:DT.cnf2	Q22
Q24/P:1:RI.req1	T:2:RI.ind2	Q25
Q24/P:1:DC.req1	T:2:DC.ind2	Q20
Q25/T:2:RI.rsp2	P:1:RI.res2	Q12

Figure 7.6 The port tables obtained in step d

Now, we apply this example to the test architecture proposed in 4.1.2 and see the function of all components in TC, TS, IS and IUT.

First we select a global test sequence (GTS). The latter is a sequence of transitions of the automata of Fig. 7.3. Note that there exist methods to generate test sequences from a given specification [2]. This aspect is not contributed in our study. Rather, our aim is, recall it, to check whether an IUT conforms to a given GTS. Let us, for example, consider the following GTS:

```
!CN.req:1?{CN.ind:2}!CN.rsp:2?{CN.cnf:1}!DT.req:1?{DT.ind:2}!DT.rsp:2?{DT.cnf:
1}!RI.req:1?{RI.ind:2}!RI.rsp:2?{RI.cnf:1}!DT.req:1?#:1!DC.req:1?{DC.ind:2}!To
ken:2?{BlockSwitching:1}!Token:1?{BlockSwitching:2}!CN.req:2?{CN.ind:1}!CN.rsp
:1?{CN.cnf:2}!DT.req:2?{DT.ind:1}!DT.rsp:1?{DT.cnf:2}!DT.req:2?{DT.ind:1}!RI.r
eq:1?{RI.ind:2}!DT.req:1?#:1!DC.req:1?{DC.ind:2}.
```

This GTS is illustrated in Fig. 7.8: its transitions are in bold and numbered in the order they are executed. For example the transition from q_{10} to q_{11} is the first and ninth transition of the contributed GTS.

The order to follow the transitions in the path are numbered and the edges are bold.

The starting (q_{10}) and ending (q_{12}) states of this GTS are distinguished.

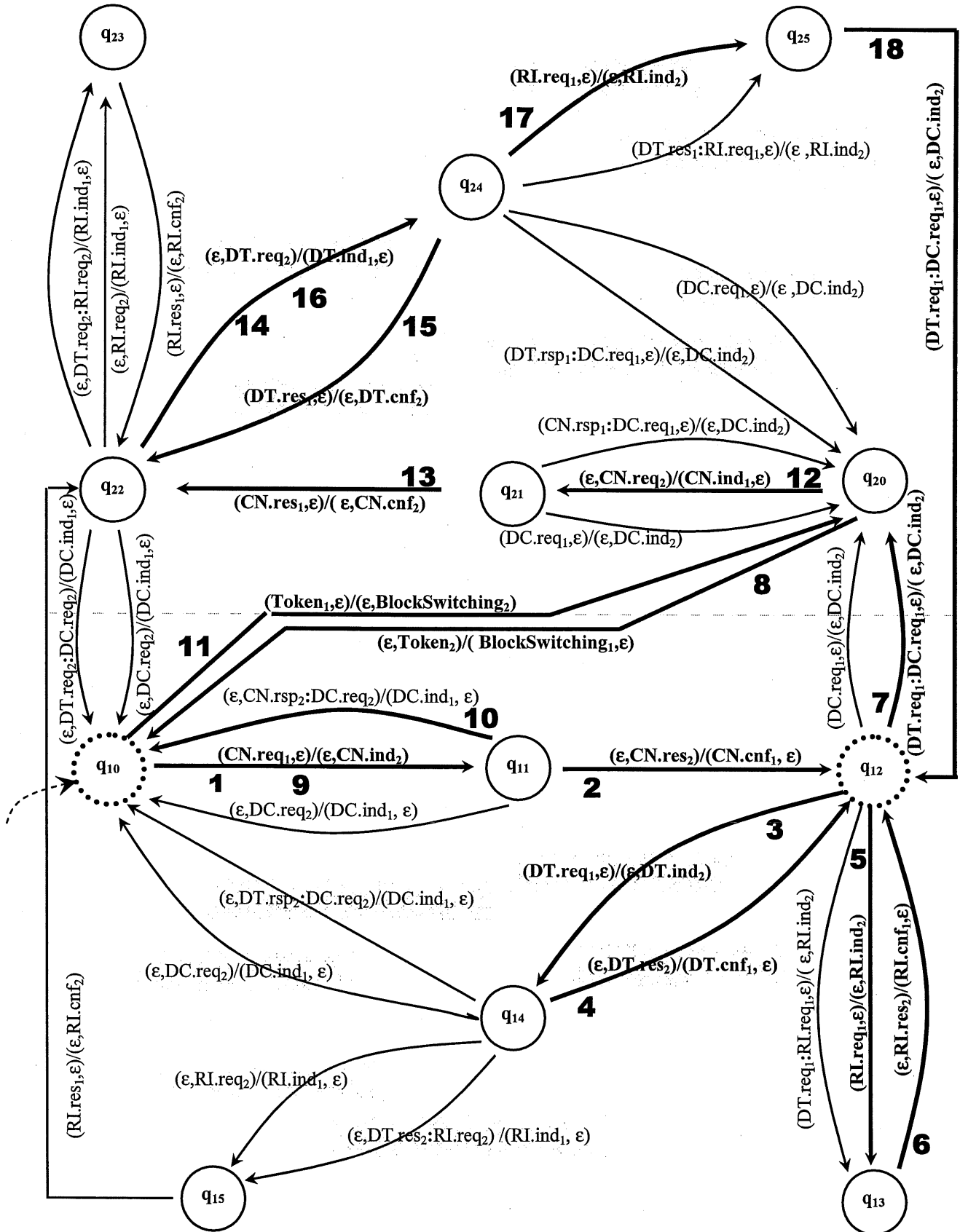


Figure 7.7 Represents the specific transitions in GTS chosen by user

Let us see what happens step by step:

- 1- Test Controller (TC), generates two Local Test Sequences (LTS) from the Global Test Sequence (GTS) because the number of sites is two.

For the selected GTS, we obtain the following LTSs. (The syntax of LTS is globally explained in Definition 2.3. and in detail in Section 3.3)

LTS for tester 1: (LTS1)

!CN.req:1 +O:2 ?CN.cnf:1 -O:2 !DT.req:1 +O:2 ?DT.cnf:1 -O:2 !RI.req:1 +O:2 ?RI.cnf:1 !DT.req:1 -O:2 !DC.req:1 +O:2 ?BlockSwitching:1 -O:2 !Token:1 +O:2 ?CN.ind:1 -O:2 !CN.rsp:1 +O:2 ?DT.ind:1 -O:2 !DT.rsp:1 +O:2 ?DT.ind:1 -O:2 !RI.req:1 !DT.req:1 !DC.req:1

LTS for tester 2: (LTS2)

?CN.ind:2 -O:1 !CN.rsp:2 +O:1 ?DT.ind:2 -O:1 !DT.rsp:2 +O:1 ?RI.ind:2 -O:1 !RI.rsp:2 +O:1 ?DC.ind:2 -O:1 !Token:2 +O:1 ?BlockSwitching:2 -O:1 !CN.req:2 +O:1 ?CN.cnf:2 -O:1 !DT.req:2 +O:1 ?DT.cnf:2 -O:1 !DT.req:2 +O:1 ?RI.ind:2 ?DC.ind:2

Testers start as soon as they receive their LTS. Each tester will process its LTS as follows:

- it starts at the first element of its LTS,
- if the element is an input for IUT (starting with ?) which has to be sent to the corresponding port, then the tester sends it to IS.
- if the element is an output from IUT (starting with !), then the tester waits until it receives it from IS.
- if the element is an outgoing coordinated message (-C and -O), tester sends it to the appropriate other tester in TS.
- if the element is an incoming coordinated message (+C and +O), then the tester waits to receive it from another tester.

- 2- TC sends these LTS1 and LTS2 to tester 1 and tester 2, respectively.
- 3- tester 1 starts scanning its LTS, and then the first element is and as we explained before (section 3.3) this message has to be sent to port 1 of IUT, so tester 1 sends this message to agent 1.
- 4- agent 1 writes the input in FileOfInput.txt.
- 5- in port 1, InputReaderFromFile reads the input from FileOfInput.txt and inserts it in QueueOfInput.

- 6- Manage picks up this input from QueueOfInput and finds the output from PortTable1, the output for this input is P:2:CN.req1 (see figure 7.6) and this is an IUT internal message for port 2 of IUT.
- 7- So Manage inserts this IUT internal message in OutputSenderToPorts[] to be sent to port 2.
- 8- Port 2 receives this IUT internal message via OutputReceiverFromPorts[] and inserts it in QueueOfInput.
- 9- Manage in port 2 removes this input from QueueOfInput and finds the output corresponding to it, which is T:2:CN.ind2 and this an output to be sent to tester 2, so Manage inserts this output to OutputWriterToFile.
- 10- OutputWriterToFile writes this output in FileOfOutput.txt.
- 11- then OutputReaderFromFile of agent 2 reads this output from FileOfOutput.txt and inserts it in OutputSenderToTester.
- 12- OutputSenderToTester sends this output to tester 2.
- 13- OutputReceiver in tester 2 receives this output and inserts it in QueueOfOutput.
- 14- then Manage of tester 2 removes this output from QueueOfOutput and compares this output with the expected outputs of its local test sequence,
 - a- if the two outputs are equal then tester 2 moves on to the next element in its local test sequence and the process will continue until both tester reach the end of their local test sequences and if testers do not find their corresponding ports faulty, then Manage in each tester inserts a 'pass' verdict in LocalVerdictSender, and latter sends the 'pass' verdict to TC and LocalVerdictReceiver in TC, receives the 'pass' verdict from two testers then the global verdict generator in TC, generates the 'pass' verdict and reports to the user that IUT is correct.
 - b- if the two compared outputs are different, then Manage in port 2 stops processing and inserts a 'fail' verdict in LocalVerdictSender and this thread sends the 'fail' verdict to TC, and Manage also inserts a 'fail' message in VerdictSender to stop port 1 from running, VerdictSender sends this message to port 1 and VerdictReceiver in port 1 receives this message and inserts the 'fail' message in VerdictSender and the latter sends the 'fail' verdict to TC, then LocalVerdictReceiver in TC receives two 'fail' verdicts from tester 1 and tester 2, then the global verdict generator in TC, generates the global verdict which is 'fail' and TC reports to the user that IUT has been found faulty.

In Figure 7.8 you can see the flow of the data.

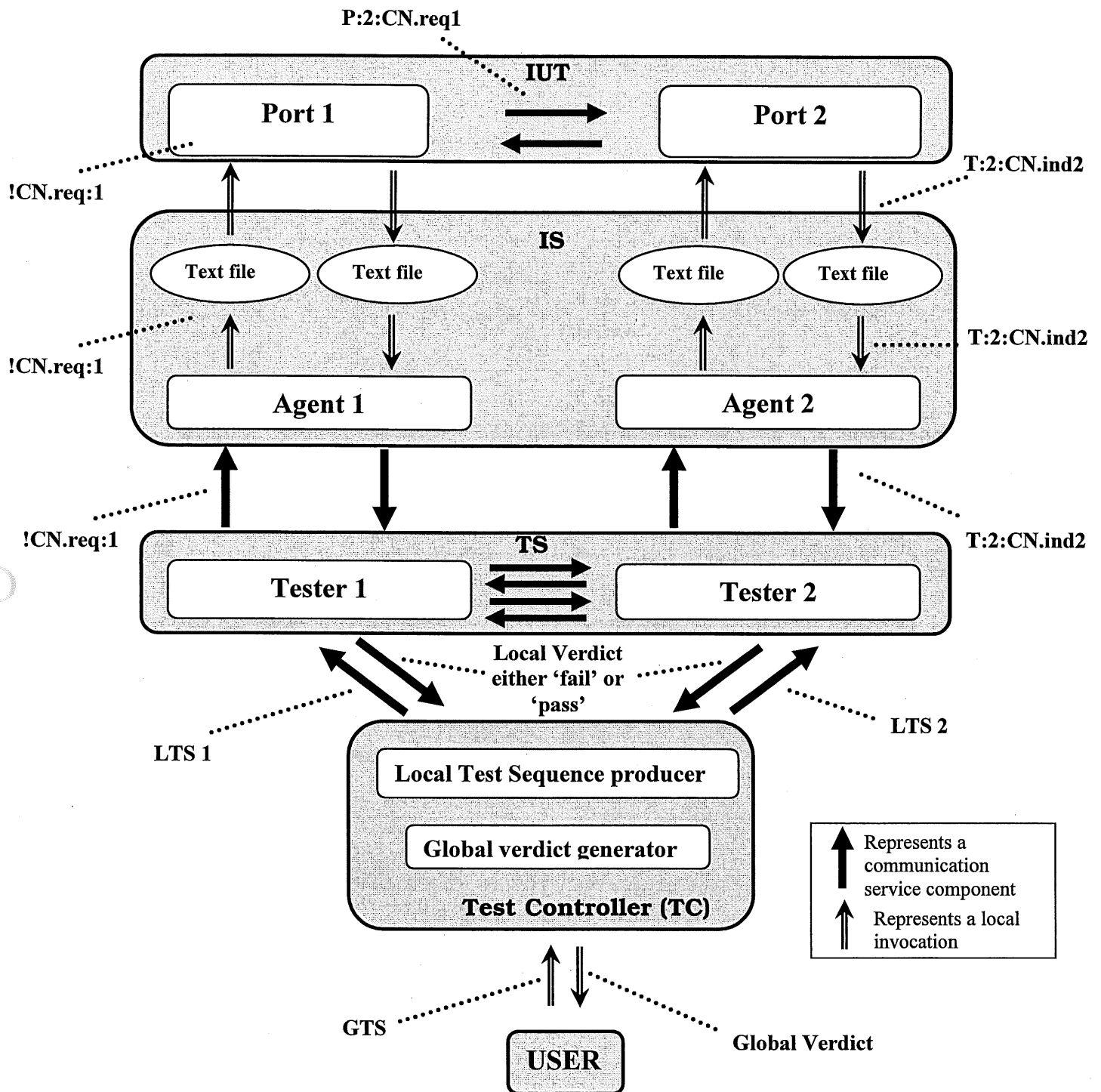


Figure 7.8 The flow of data

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this chapter we conclude our work, introduce some restriction and limitation in the realized program and the future work.

8.1 conclusion

In this paper, we discussed about testing in general, then we introduced different standard test architectures, and then we discussed about two problems, which raise with distributed test architecture. We then introduced a coordinated method, which solves those two problems.

We proposed an architecture and design a distributed Test System, which is able to test a real distributed Implementation Under Test. Finally, we implemented our method using Java.

8.2 Future work

In the realized test architecture, there is a restriction that we introduce below:

Restriction

In the Test System, when a tester wants to send an input to IUT, a restriction has to be respected. This restriction happens because of a limitation in Event Service of CORBA, which is discussed in chapter 5.3.3, this limitation is Lack of Reliability.

Let us precise this limitation.

It is extremely important to keep in mind that event channels of Event Service are fundamentally unreliable. Their lack of reliability stems from the difficulty of providing end-to-end guaranteed delivery in a service in which the channel has no way to throttle the supplier. If a supplier pushes so many events that the event channel cannot keep up with delivering all of them to its consumers, the event channel has no choice except to drop some of the events.

Let us consider the example from chapter 7. Let us assume the following is the LTS for tester 1:

$LST_1 = \dots !DT.req:1\#:1!DC.req:1\{DC.ind:2\} \dots$

And we are at the state q_{21} .

As you can see in the transition of figure 8.1 in q_{21} tester 1 sends $CN.rsp_1$ as an input and then, right after this input, it sends $DC.req_1$ in order to request for a disconnection, while tester 2 sends nothing and then, in response to this input, tester 2 receives $DC.ind_2$.

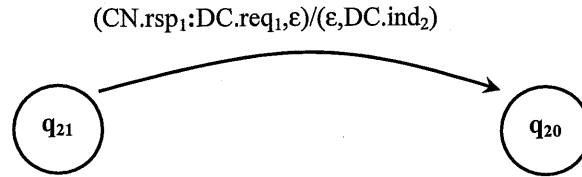


Figure 8.1 Transition

If tester 1, after sending $CN.rsp_1$, immediately sends the second input which is $DC.req_1$ so, because of Lack of Reliability limitation of event service, the event service might not be able to deliver the second message, and it might drop it. So in this case we will not have a reliable tester, which is testing the IUT, because tester 2 will receive the wrong output from IUT.

Let us see what happens if the event channel drops the second message.

We are currently in state q_{21} and then tester 1 sends the first message followed by the second message. Now, if event channel does not drop the second message, tester 2 receives $DC.ind_2$, which is the right output from IUT and we go to q_{20} . If event channel drops the second message of tester 1, then tester 2 receives $CN.cnf_2$, which is the wrong output from IUT, and then we go to state q_{22} .

This is represented in figure 8.2.

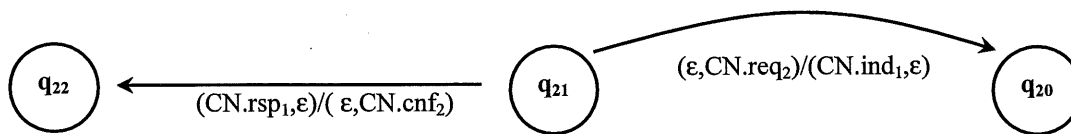


Figure 8.2 Transition

Then tester 2 will detect IUT as faulty, and this is a fault of Test System, not IUT.

To solve this problem, after sending the first input, tester 1 must wait for a *certain time*, and then sends the second input, to ensure that event channel will not drop the second message.

Unfortunately, this period of time that tester must wait, is not constant, and varies from machine to machine, and from network to network on which the testers are running. Solving this problem can save the performance time of Test System, therefore the total time of execution of program to test the IUT and also it can decrease the complexity of program's code, and also program's execution procedure, since the user has to execute the program several times to determine that *certain time* and then set it up in the code. So we consider it as a restriction for this program, and finding a solution could be a future work.

Another future work can be testing real-time IUT [14], i.e., which must respect timing constraints. In this case, we have to check:

- The order of events,
- The instant of time when events occur.

References

- [1] http://www.dot.state.nc.us/it/DMV_Business_Systems/DMVApplicationDevelopment/irp/SoftwareDevelopmentPlan/default.html.
- [2] Dssouli, R., Saleh, K., Aboulhamid, E., En-Nouaary, A., Bourhfir, C. (1999) *Test development for communication protocols: towards automation*. Computer Networks, 31, p 1835-1872.
- [3] Cacciari, L., Rafiq, O. (1999) *Controllability and observability in distributed testing*, Information and software technology, 41, p. 767-780.
- [4] Huey-Der Chu, (1997) *An Evaluation Scheme of Software Testing Techniques*. Department of Computing Science, University of Newcastle.
- [5] Khoumsi, A. (November 2000) *Timing issues in testing non real-time distributed system*, in *IASTED Intern*, Conference on Software Engineering and Applications (SEA), Las Vegas, USA.
- [6] Benattou, M., Cacciari, L., Pasini, R., Rafiq, O. (September 1999) *Principles and tools for testing open distributed systems*. In 12th Intern. Workshop. On Testing of Communicating Systems (IWTCS), Budapest, Hungary, Kluwer Academic Publishers. p. 77-92.
- [7] Henning, M., Vinoski, S. (1999) *Advanced CORBA programming with C++*, published by Addison Wesley Longman, Inc. ISBN: 0-201-37927-9.
- [8] Khoumsi, A. (October 1999) *Protocol synthesis for real-time applications*, in Joint Intern. Conferences on Protocol Specification, Testing and Verification (PSTV) and FORMAL description TECHniques for distributed systems (FORTE), Beijing, China.

- [9] Edmonds, J., Johnson, E.L (1973) *Matching, Euler tours and the Chinese postman*, Math. Programming 5.
- [10] T.Walter, I.Schieferdecker, J.grabowski, *Test Architectures for Distributed Systems- State Of the Art and Beyond*.
- [11] <http://www.ooc.com>.
- [12] Naito, S., Tsunoyama, M. (1981) *Fault detection for sequential machines by transition tours*, Proc. 11th Fault Tolerant Computing Symposium, IEEE, New York, p. 238-243.
- [13] http://www.pogner.demon.co.uk/mil_498.
- [14] Khoumsi A. (2001) *Testing distributed real time systems using a distributed test architecture*. In IEEE Intern.Symposium on Computers and Communications (ISCC), Hammamet, Tunisia.

APPENDIX A

REQUIREMENTS AND INSTRUCTIONS TO RUN THE PROGRAM

In this appendix, we bring the instruction and requirement to run the program.

This chapter includes step-by-step instruction to run all parts of program, including TC, TS, IS and IUT.

1. Requirements

First, the requirements to be able to run the program, you need to have the followings:

1- A Java developer.

This program is written by jdk-1.2.2, you can find this version of Java in the following web site, www.sun.com , all details about the compiler and steps to download and install it on your hard disk.

Let us assume the Java developer is installed on C drive as follows: c:\jdk-1.2.2

2- A CORBA compliant Object Request Broker.

Between many vendors who provides CORBA, we select to use ORBacus that supports the C++ and Java language mappings.

ORBacus for C++ and Java is a robust, full-featured object request broker with a proven track record and enterprise-class features. ORBacus is fully compliant with CORBA specifications, FREE for non-commercial use and available with complete source code.

The official web site of ORBacus is www.orbacus.com . To download the free of charge software, you can go to this address: <http://www.orbacus.com/products/download.html>, there are two version of ORBacus are available at this address, 3 and 4, this program is using version 3, specifically 3.3.2.

To download this software, there are two options,

- 1- The complete ORBacus for Java 3.3.2 source code, this is the file's name: JOB-3.3.2.zip or JOB-3.3.2.tar.qz
- 2- Precompiled JAR files from ORBacus for Java 3.3.2, this is the file's name: JOB-3.3.2.jars.zip or JOB-3.3.2.jars.tar.qz

In the first case you have to install and compile the software to obtain the JAR files, while in the second case you are going to have the JAR files directly.

Let us assume the ORBacus Jar files are ready to use on the C drive as follows: c:\ ooc

This folder will contain these files:

OBEvent.jar, OB.jar, OBNaming.jar, OBProperty.jar, OBTest.jar, OBTime.jar, OBTrading.jar, OBUtil.jar

Next step is to prepare a bath file:

You can create a bath file and name it auto.bath as follows:

```
set path=C:\jdk1.2.2\bin;  
SET CLASSPATH=%CLASSPATH%;C:\ooc\OB.jar;C:\ooc\OBEvent.jar;
```

To run our program we only need two of Jar files, OB.jar and OBEvent.jar, that's the reason we bring only these files in the bath file.

2. Instructions

In this step we show how to run each part of program.

First we have to configure the configuration file for ORBacus.

As we introduced, this file is called, ob.conf which exists in all folders including TestController, TestSystem and IUT (in all ports)

Let us see how this file looks like:

```
ooc.service.EventService=iioploc://HostName:PortNumber/DefaultEventChannel  
ooc.service.EventChannelFactory=iioploc://HostName:PortNumber/DefaultEventChannelFactory
```


For HostName you can insert the IP address of the host that you are going to run the ORBacus on it, and for PortNumber you can choose a port, this port must be free to use.

Let us assume that IUT has two ports, so you will need 7 machines to run the whole program as follows:

- 1- To run the CORBA (ORBacus)
- 2- TestController
- 3- Tester1
- 4- Tester2
- 5- Port1 and Agent 2
- 6- Port2 and Agent 2

It is necessary to be mentioned that you can run the ORBacus either on a separate machine or on one of other machines which you are running the TestController or testers or ports, in this case you will have totally 6 machines that will be running the program.

We go with the first case, which we run the ORBacus on a separate machine.

You must get the IP address of the machine which on you are running the ORBacus, then you insert this IP address in the HostName in ob.conf and you insert a desire port number in the PortNumber, then this file is ready to use.

All ob.conf files on all other machines will be configured exactly the same.

In the following for every dos window you open, you first run the bath file auto.bath to load the Java and ORBacus.

“X:” shows the drive letter that the program is on it.

Step 1:

We run the ORBacus on machine number 1.

cd X:\~\project\TestController

java com.ooc.CosEventServer.Server -ORBconfig ob.conf -ORBthreaded -OAThread_per_request

This will load the event service of ORBacus in the memory.

Step 2:

We run the TestController on machine 2:

Window 1:

```
cd X:\Project\TestController
```

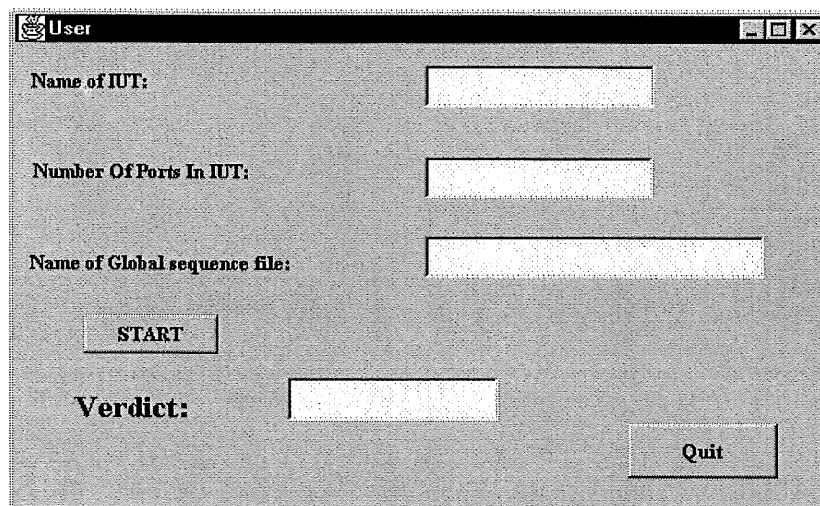
```
java com.ooc.CosEventServer.Server -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

Window 2:

```
cd X:\Project\TestController
```

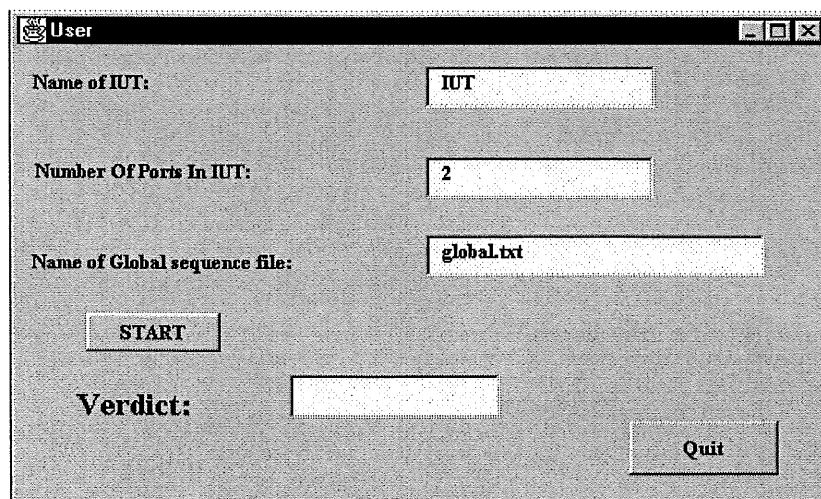
```
java User -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

By running this command you will see this user interface:



The screenshot shows a window titled "User" with a standard Windows-style title bar. Inside the window, there are three text input fields with labels to their left: "Name of IUT:", "Number Of Ports In IUT:", and "Name of Global sequence file:". Below the first two fields is a "START" button. At the bottom left, there is a "Verdict:" label followed by an empty text box. At the bottom right, there is a "Quit" button. All input fields are currently empty.

In the "Name of IUT" you must insert the name of Implementation Under Test (IUT), let it be "IUT", in the next field you insert the number of ports existing in the IUT, which is 2 as we assumed, and in the next field you insert the name of text file which contains the Global Sequence, by doing these, you will have the user interface as follows:



This screenshot shows the same "User" window as before, but with the input fields filled. The "Name of IUT:" field contains the text "IUT". The "Number Of Ports In IUT:" field contains the text "2". The "Name of Global sequence file:" field contains the text "global.txt". The "START" button, "Verdict:" label with its text box, and "Quit" button remain in the same positions as in the previous screenshot.

Step 3:

We run the Tester 1 on machine number 3.

Window 1:

```
cd X:\~\project\TestSystem
```

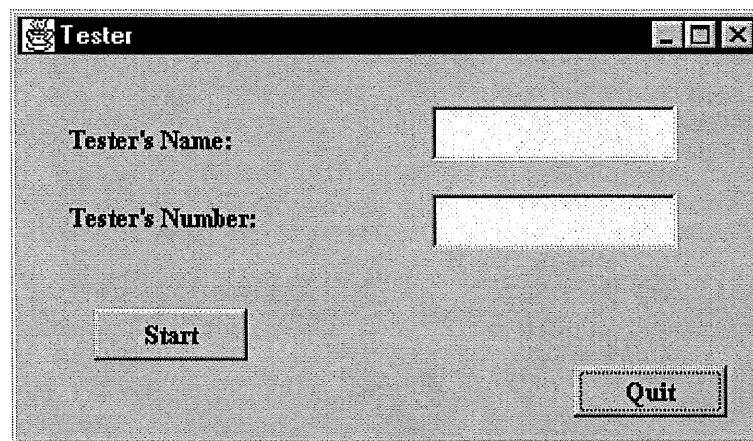
```
java com.ooc.CosEventServer.Server -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

Window 2:

```
cd X:\~\project\TestSystem
```

```
java Tester -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

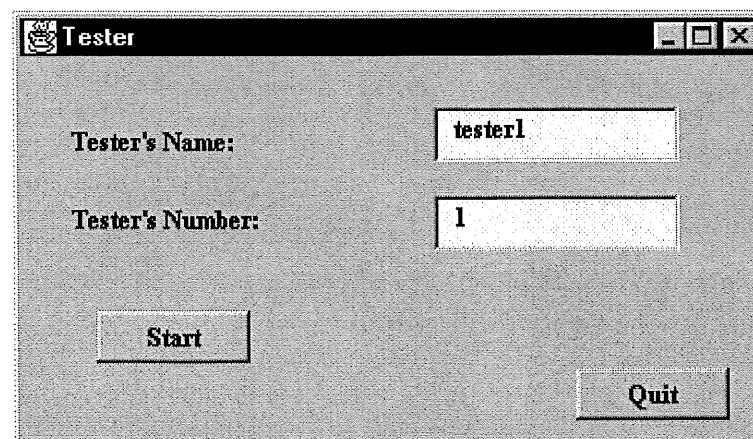
By running this command you will see this interface:



The screenshot shows a window titled "Tester". Inside the window, there are two labels: "Tester's Name:" and "Tester's Number:". To the right of each label is an empty text input field. Below the "Tester's Name:" field is a button labeled "Start". Below the "Tester's Number:" field is a button labeled "Quit".

In the first field you insert the name of tester, let it be "tester1" and in the second field tester's number which is 1.

Then you will have this interface:

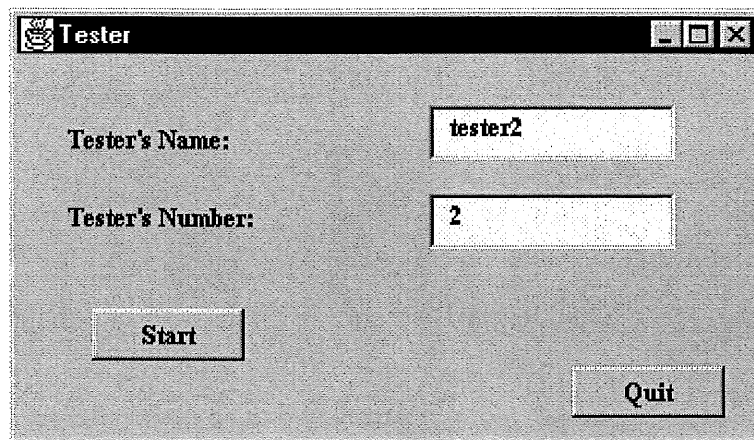


The screenshot shows the same "Tester" window as before, but now the input fields contain text. The "Tester's Name:" field contains the text "tester1" and the "Tester's Number:" field contains the text "1". The "Start" and "Quit" buttons are still present below their respective fields.

Then you click on Start button.

You must run tester 2 on machine number 4 exactly the same way as tester 1.

This is what you will have as user interface for tester 2:



Step 4:

We run port 1 and agent 1 on machine number 5.

Window 1:

```
cd X:\~\project\IUT\Port1
```

```
java com.ooc.CosEventServer.Server -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

Window 2: (to run agent 1)

```
cd X:\~\project\IUT\Port1
```

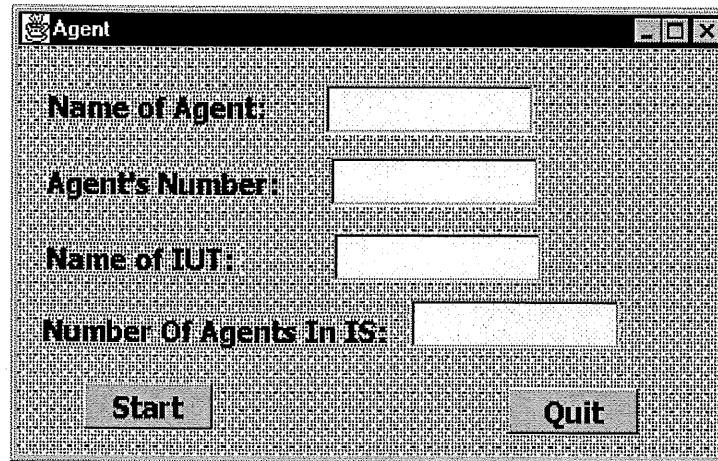
```
java Agent -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

Window 3: (to run port 1)

```
cd X:\~\project\IUT\Port1
```

```
java Port -ORBconfig ob.config -ORBthreaded -OAThread_per_request
```

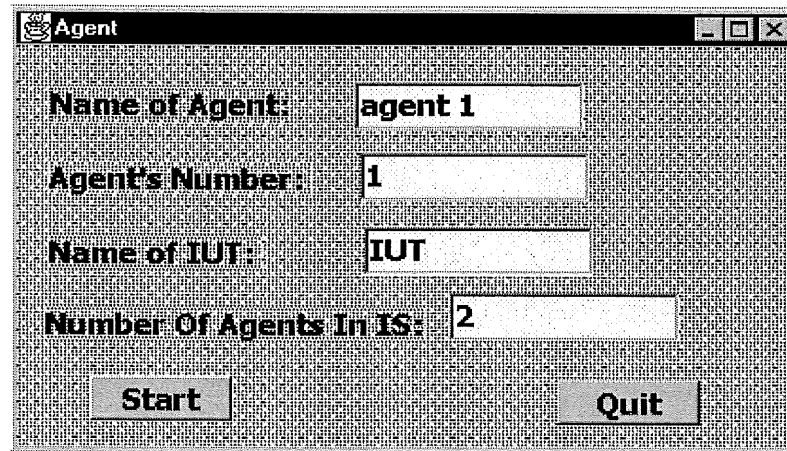
By running window 2, this user interface will be appeared:



The screenshot shows a window titled "Agent" with a standard Windows-style title bar (minimize, maximize, close buttons). Inside the window, there are four labels with corresponding text input fields: "Name of Agent:", "Agent's Number:", "Name of IUT:", and "Number Of Agents In IS:". All four input fields are currently empty. At the bottom of the window, there are two buttons: "Start" on the left and "Quit" on the right.

In the first field you insert the name of agent, let it be "agent1", in the second field the agent's number, which is 1 and in the next one, the name of IUT, which is "IUT" and in the next one the number of agents in IS, which is 2 (as we assumed).

By inserting this information you will have this interface:



This screenshot shows the same "Agent" window as before, but now the input fields contain text. The "Name of Agent:" field contains "agent 1", the "Agent's Number:" field contains "1", the "Name of IUT:" field contains "IUT", and the "Number Of Agents In IS:" field contains "2". The "Start" and "Quit" buttons remain at the bottom.

Then you click on "Start" button.

By running window 3, you will have the same interface for port; the completed one is like following:

Port

Name Of Port:

Port's Number:

Name Of IUT:

Number Of Ports In IUT:

Then you click on “Start” button.

Then you must run agent 2 and port 2 on machine number 6 the same way as explained for agent 1 and port 1.

Step 5:

You click on “Start” button on the test controller interface.

User

Name of IUT:

Number Of Ports In IUT:

Name of Global sequence file:

Verdict:

Step 6:

You wait to see the verdict in the “Verdict” field on the TestController user interface.

This can be either “fail” or “pass”. If you receive “pass” it means the IUT is not detected as faulty, and if you receive “fail” it means the IUT is detected as faulty.

APPENDIX B

ALGORITHM TO OBTAIN COORDINATED LOCAL TEST SEQUENCES FROM GLOBAL TEST SEQUENCE

In this appendix, we introduce an algorithm, which produces coordinated local test sequences from global test sequence.

begin

for $k=1, \dots, n$ **do** $\omega_k \leftarrow \varepsilon$

for $i=1, \dots, t$ **do**

$k \leftarrow \text{port}(x_i)$

To add the observation messages

if $i > 1$ **then**

1: $\text{send_To} \leftarrow (\text{ports}(y_i) \Delta \text{ports}(y_{i-1})) \setminus \{k\}$

2: **if** $\text{sender} \neq 0$ **then** $\text{send_To} \leftarrow \text{send_To} \setminus \{\text{sender}\}$ **end if**

3: **if** $\text{send_To} \neq \emptyset$ **then**

$\omega_k \leftarrow \omega_k . -O_{\text{send_To}}$

for all $h \in \text{send_To}$ **do** $\omega_h \leftarrow \omega_h . +O_k$ **end for**

end if

To add the observation messages

4: $\omega_h \leftarrow \omega_h . !x_i$

5: **for all** $a \in y_i$ **do** $\omega_{\text{port}(a)} \leftarrow \omega_{\text{port}(a)} . ?a$

To add the control messages

if $i < t$ **then**

$h \leftarrow \text{port}(x_{i+1})$

$\text{sender} \leftarrow 0$

6: **if** $h \notin \text{ports}(y_i) \cup \{k\}$ **then**

select

7: **case** $y_i = \emptyset$:

$\omega_k \leftarrow \omega_k . -C_h$

$\omega_h \leftarrow \omega_h . +C_h$

$\text{sender} \leftarrow k$

8: **case** $\text{ports}(y_i) \setminus \text{ports}(y_{i+1}) \neq \emptyset$:

choose $l \in \text{ports}(y_i) \setminus \text{ports}(y_{i+1})$

$\omega_l \leftarrow \omega_l . -C_h$

$\omega_h \leftarrow \omega_h . +C_l$

$\text{sender} \leftarrow l$

9: **otherwise** :

choose $l \in \text{ports}(y_i)$

$\omega_l \leftarrow \omega_l . -C_h$

$\omega_h \leftarrow \omega_h . +C_l$

$\text{sender} \leftarrow l$

return($\omega_1, \dots, \omega_n$)

End Algorithm

This algorithm uses two kinds of coordination messages:

C coordination messages for guaranteeing Controllability

O coordination messages for guaranteeing Observability

If we apply this algorithm to the test sequence (1) (chapter 3), we obtain the following local test sequences:

$$\omega_1 = !a ?x ?x + O_3 + C_3 !a ?x ?x ?w !a ?x + O_3$$

$$\omega_2 = ?y !b ?y -C_3 + C_3 !b ?y -C_3 + O_3 ?y$$

$$\omega_3 = +C_2 -O_1 !c ?z -C_1 ?z -C_2 + C_2 !c ?z ?z -O_{\{1,2\}} !c ?z$$

Now we can follow the algorithm step by step to generate the Local Test Sequences:

$$W = !a?\{x,y\} !b?\{x,y\} !c?\{z\} !a?\{x,z\} !b?\{x,y\} !c?\{w,z\} !a?\{x,z\} !c?\{y,z\}$$

For i = 1, in !a?{x,y}:

$$K = \text{Port}(x_i) = \text{Port}(x_1) = 1$$

$$I = 1$$

$$(4:) \omega_1 = \omega_1 . ! x_1 = \varepsilon . !a$$

$$(5:) y_1 = \{x,y\}$$

$$(5:) \omega_{\text{port}(x)} = \omega_1 = \omega_1 . ?x = !a?x$$

$$(5:) \omega_{\text{port}(y)} = \omega_2 = \omega_2 . ?y = \varepsilon . ?y = ?y$$

$$h = \text{Port}(x_{i+1}) = \text{Port}(x_2) = 2$$

$$\text{sender} = 0$$

$$\text{Ports}(y_i) = \text{Ports}(y_1) = \{1,2\}$$

$$(6:) h \in \{1,2\} \cup \{1\}$$

$$\Rightarrow \omega_1 = !a?x$$

$$\Rightarrow \omega_2 = ?y$$

$$\Rightarrow \omega_3 = \varepsilon$$

For i = 2, in !b?{x,y}:

$$K = \text{Port}(x_i) = \text{Port}(x_2) = 2$$

$$\text{Ports}(y_i) = \text{Ports}(y_2) = \{1,2\}; \text{Ports}(y_{i-1}) = \text{Ports}(y_1) = \{1,2\}$$

$$\text{Ports}(y_i) \Delta \text{Ports}(y_{i-1}) = \{1,2\} \Delta \{1,2\} = \emptyset$$

$$(1:) \text{Send_To} = \text{Ports}(y_i) \Delta \text{Ports}(y_{i-1}) \setminus \{k\} = \emptyset \setminus \{2\} = \emptyset$$

$$(2:) \text{sender} = 0$$

$$(3:) \text{Send_To} = \emptyset$$

$$(4:) \omega_2 = \omega_2 . ! x_2 = ?y!b$$

$$(5:) y_2 = \{x,y\}$$

$$(5:) \omega_{\text{port}(x)} = \omega_1 = \omega_1 . ?x = !a?x?x$$

$$(5:) \omega_{\text{port}(y)} = \omega_2 = \omega_2 . ?y = ?y!b?y$$

$$h = \text{Port}(x_{i+1}) = \text{Port}(x_3) = 3$$

$$\text{sender} = 0$$

$$\text{Ports}(y_i) = \text{Ports}(y_2) = \{1,2\}$$

$$(6:) h \notin \{1,2\} \cup \{2\}$$

$$(8:) \text{Ports}(y_i) \setminus \text{Ports}(y_{i-1}) = \text{Ports}(y_2) \setminus \text{Ports}(y_1) = \{1,2\} \setminus \{1,2\} = \emptyset$$

$$(9:) 1 = 2 \in \text{Ports}(y_i)$$

$$\omega_1 = \omega_1 . - C_h = \omega_2 . - C_3 = ?y!b?y - C_3$$

$$\omega_h = \omega_h . + C_1 = \omega_3 . + C_2 = +C_2$$

$$\text{sender} = 2$$

$$\Rightarrow \omega_1 = !a?x?x$$

$$\Rightarrow \omega_2 = ?y!b?y - C_3$$

$$\Rightarrow \omega_3 = +C_2$$

For i = 3 in !c?{z}:

$$\Rightarrow \omega_1 = !a ?x ?x + O_3 + C_3$$

$$\Rightarrow \omega_2 = ?y !b ?y - C_3$$

$$\Rightarrow \omega_3 = +C_2 - O_1 !c ?z - C_1$$

For i = 4 in !a?{x,z}:

$$\Rightarrow \omega_1 = !a ?x ?x + O_3 + C_3 !a ?x$$

$$\Rightarrow \omega_2 = ?y !b ?y - C_3 + C_3$$

$$\Rightarrow \omega_3 = +C_2 - O_1 !c ?z - C_1 ?z - C_2$$

For i = 5 in !b?{x,y}:

$$\Rightarrow \omega_1 = !a ?x ?x + O_3 + C_3 !a ?x ?x$$

$$\Rightarrow \omega_2 = ?y !b ?y - C_3 + C_3 !b ?y - C_3$$

$$\Rightarrow \omega_3 = +C_2 - O_1 !c ?z - C_1 ?z - C_2 + C_2$$

For i = 6 in !c?{w,z}:

$$\Rightarrow \omega_1 = !a ?x ?x + O_3 + C_3 !a ?x ?x ?w$$

$$\Rightarrow \omega_2 = ?y !b ?y - C_3 + C_3 !b ?y - C_3$$

$$\Rightarrow \omega_3 = +C_2 - O_1 !c ?z - C_1 ?z - C_2 + C_2 !c ?z$$

For i = 7 in !a?{x,z}:

$$\Rightarrow \omega_1 = !a ?x ?x + O_3 + C_3 !a ?x ?x ?w !a ?x$$

$$\Rightarrow \omega_2 = ?y !b ?y - C_3 + C_3 !b ?y - C_3$$

$$\Rightarrow \omega_3 = +C_2 - O_1 !c ?z - C_1 ?z - C_2 + C_2 !c ?z ?z$$

For i = 8 in !c?{y,z}:

$$\omega_1 = !a ?x ?x + O_3 + C_3 !a ?x ?x ?w !a ?x + O_3$$

$$\omega_2 = ?y !b ?y - C_3 + C_3 !b ?y - C_3 + O_3 ?y$$

$$\omega_3 = +C_2 - O_1 !c ?z - C_1 ?z - C_2 + C_2 !c ?z ?z - O_{\{1,2\}} !c ?z$$