Timed Test Case Execution for Distributed Real-Time Systems

Shoukat Hayat Siddiquee

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

November 2005

Your file Votre référence
ISBN: 0-494-14282-0
Our file Notre référence
ISBN: 0-494-14282-0

ABSTRACT

Timed Test Case Execution for Distributed Real-Time Systems

Shoukat Hayat Siddiquee

A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment. In real-time applications, the timing requirements are the main constraints and their mastering is the predominant factor for assessing the quality of service.

We proposed two methods for timed test case execution for distributed real time systems modeled as TIOA.

Ensuring the correctness of real time systems before the development and guaranteeing that it functions correctly within the specified time constraints are very critical to avoid catastrophic consequences. A number of design issues affect the testing strategies and the testability of the system. Existing methods of distributed test systems still have some drawbacks. Firstly, there is lack of coordination among local testers in a distributed system. Secondly, the centralized test method may result in performance problems, due to the fact that all *TS-IUT* interactions are achieved through the network. With an aim to alleviate the shortcomings, we developed a Synchronizer in centralized test system, which would control and coordinate the local testers in the test environment. We implemented these methods and tested them with a benchmark of real life automata model by using CORBA distributed architecture. As a result, the performance problem has been reduced in both methodologies. We distinguished between two methodologies

and compare their performances in terms of functionalities, based on our implementation and illustrate their approaches using Train-Gate-Controller case study.

Keywords: Timed input-output automata, Real-time systems, Distributed test system, Centralized test system, CORBA, Conformance testing.

ACKNOWLEDGEMENTS

On an academic level, I would like to express my sincere gratitude to my supervisor, Dr. Abdeslam En-Nouaary. Without his guidance, encouragement, support and kindness, this thesis would not have been possible. I really appreciate his invaluable suggestion and inspiring discussions and comments.

I would also like to convey my gratefulness to the examiners Dr. Olga Ormandjieva and Dr. Otmane Ait Mohamed for spending their valuable time and also my earnest gratitude to the chairman Dr. Amir G. Aghdam.

On a professional level, I would like to thank all my colleagues at Electrical and Computer Engineering department as well as its staffs for their sharing views and ideas on different technological issues.

Finally, I dearly thank my wonderful wife, my parents and my sister.

To my Father

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATION

IUT – Implementation Under Test.

TTCN – Tree and Tabular Combined Notation.

PCO- Point of Control and Observation

LS – Local Single Layer.

DS –Distributed Single Layer

RS – Remote Single Layer

CS – Coordination Single Layer

TT – Transition tour

DS – Distinguishing Sequence method

UIO – Unique Input Output

CORBA – Common Object Request Broker Architecture

IDL – Interface Definition Language

OMG – Object Management Group

ORB – Object Request Broker

RT-CORBA – Real Time CORBA

POA – Portable Object Adapter

TIOA – Timed Input Output Automata

TS – Test System

ATS – Abstract Test Suite

DII – Dynamic Invocation Interface

RPC – Remote Procedure Call

ASP – Abstract Service Primitive

Chapter 1

Introduction

Computer applications are being increasingly used as the core of safety-critical real-time systems, such as patient monitoring systems, plant control systems, air traffic control systems and telecommunication systems. For such systems, a functional misbehavior or a deviation from the specified time constraints may have catastrophic consequences. In real-time applications, the timing requirements are the main constraints and their mastering is the predominant factor for assessing the quality of service. Therefore, ensuring the correctness of real-time systems become necessary. Two different techniques are usually used to cope with the correctness of a software system prior to its deployment, namely - verification and testing.

Testing is an important activity, which aims to ensure the quality of the implementation. Testing procedure consists of generating test suite and applying them to the implementation, which is referred to as an *implementation under test* (*IUT*). There exist mainly three testing strategies: *white box testing, black box testing, and gray box testing*. In *white box testing*, the structure of the implementation is known and the test suite is generated from the implemented structures. In *black box testing*, however, the structure of the implementation is not known; we use the specification of the required functionality at defined interfaces for test generation, execution, and evaluation. Finally, in *gray-box testing*, we assume that modular structure of the implementation is known but not the details of the programs within each component.

The PDCS (Predictably Dependable Computer Systems) (Randell et al.., 1995) project defines RTS as a *system* that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by environment. To illustrate the various ways in, which 'real-time' systems are defined, one further definition is given. Young (1982) defines a real-time system to be: Any information processing activity or system, which has to respond to externally generated input stimuli within a finite and specified period. Real time systems are classified into two ways: *Hard real- time systems* and *Soft real- time systems.*

The classic definition is that *hard real-time* application fails if the system's timing requirement is not met. Good examples are digital fly-by-wire system of aircraft and missile self cruise system.

The *soft real-time* applications can tolerate some degree of latency in what they are required by the system. The time constraint violation is not fatal, but the system performance may degrade. Examples are vendor machine and printer controller.

1.1 Testing Real-Time Software Systems

Real time applications are harder to test because they require not only the correct output signals, but also the signals occurring at the correct time. To ensure the correctness of the implementation, the output signals and the time should both be checked upon testing.

The most efficient way to describe the complicated system behaviors is to use formal models. There are different models such as Finite State Machine (FSM), Petri-nets and Input-Output Automata. Most of proposed formal models for real-time systems are

time enrichment of those traditional models, with time constraint and additional time labels, such as Timed Finite State Machine (TFSM) and Timed Input Output Automata (TIOA)([30][31]. To test real-time systems described by formal models, the obvious means for lowering test cost is deriving test suite from the models automatically. The observed output traces from the *IUT* should also be analyzed against the models.

1.2 Contribution of the Thesis

Although there are already some methods for centralized test system, the problems of those methods still exist. First of all, there is lack of coordination of local testers in distributed system, because all local testers are independent of *IUT*. Secondly, the centralized test method may result in performance problems, due to the fact that all interactions *TS-IUT* are achieved through the network. Thirdly, a quantitative comparative analysis between the centralized and distributed methods is very relevant.

In this thesis, we propose two methods of timed test case execution based on specification by formal model of TIOA. We use n-port TIOA to describe the specification and also consider that test case generation will be applied by transition tour method on the TIOA specification. We apply these two methods for test case execution on real life automata model like Train-Gate-Controller distributed system.

The main contributions of our thesis work are the followings:

- Generating test cases from n-TIOA by Transition Tour method.
- Developing an algorithm for test case execution on centralized test system.
- Developing an algorithm for test case execution on distributed test system.
- Implementation and testing of centralized test system through CORBA with Java.

- Implementation and testing of distributed test system through CORBA with Java.
- Implementation of "Train Gate Controller" - a benchmark example, which is real-time reactive system for *IUT* through CORBA with Java.
- Comparison between centralized and distributed test systems and case study.

1.3 Organization of the Thesis

The structure of this thesis is as follows:

Chapter 2 presents the overview of the Conformance testing activities. The well known test methodology is introduced here, as well as the framework of conformance testing and test methods. We also discuss the conformance relation between reference specification and *IUT*. The chapter also introduces the concepts of test hypotheses as well as fault coverage and fault model.

Chapter 3 represents CORBA to introduce this industrial based architecture for implementing distributed system and also analyze the Real-Time CORBA to ensure the distributed system is robust and efficient. The chapter also introduces the CORBA implementation model for Borland's visibroker. This software has been used for implementation by CORBA architecture with Java.

Chapter 4 presents the overview of two proposed methodologies, definition of n-TIOA for centralized and distributed test systems. It also provides centralized and distributed test methods with their architectures.

Chapter 5 shows the design details, implementation and case study using an example that demonstrates our implementation methodology and the usefulness of the

tool. We provide several class diagrams, sequence diagrams and collaboration diagrams in UML to specify our design and implementation.

Chapter 6 comes with the summary of this thesis and the future work.

Chapter 2

Overview of Conformance Testing

In this chapter, we introduce the background concepts and knowledge about testing. A standard conformance testing methodology and framework is presented in section 2.1, section 2.2 introduces conformance testing and specification and section 2.3 represents test hypotheses. Section 2.4 discusses FSMs fault model. Section 2.5 provides the FSMs based test suite generation in, which several well-known test suite derivation methods are introduced.

2.1 Conformance Testing and Standards

Conformance is usually defined as testing to see if an implementation faithfully meets the requirements of a standard or specification. There are many types of testing including testing for performance, robustness, behavior, functions and interoperability. Although conformance testing may include some of these kinds of tests, it has one fundamental difference - the requirements or criteria for conformance must be specified in the standard or specification. This is usually in a conformance clause or conformance statement, but sometimes some of the criteria can be found in the body of the specification. Some standards have subsequent documentation for the test methodology and assertions to be tested. If the criteria or requirements for conformance are not specified, there can be no conformance testing.

The distributed real-time systems communicate with each other by exchanging messages, which are governed by the communication protocol. A communication protocol is a set of rules that govern an orderly exchange of messages among communicating entities. Testing plays a major role in the development of communication protocols.

There are many reasons that cause incompatibility of distributed system i.e. fail to interact with each other. First, the developer could make some errors due to complexity of specification. Second, specification could be incomplete, so that different implementation could have different behaviors for the incomplete part of the specification. Third, different implementations of the same specification might result from various interpretations. Finally, the specification could provide a range of choices, which may result in incompatibilities.

Conformance testing is the activity of checking whether a new implementation conforms to a specification or not. However, different test developers could have different principles when deciding if the implementation conforms to its specification. Moreover, the same product could be tested more than once by different test developers. Therefore, a general principle and test procedure for conformance testing are necessary, so that the repeated conformance testing for the same system can be minimized. International Organization for Standardization (ISO) has developed a standard for conformance testing, that is ISO IS-9646, "OSI Conformance Testing Methodology and Framework".

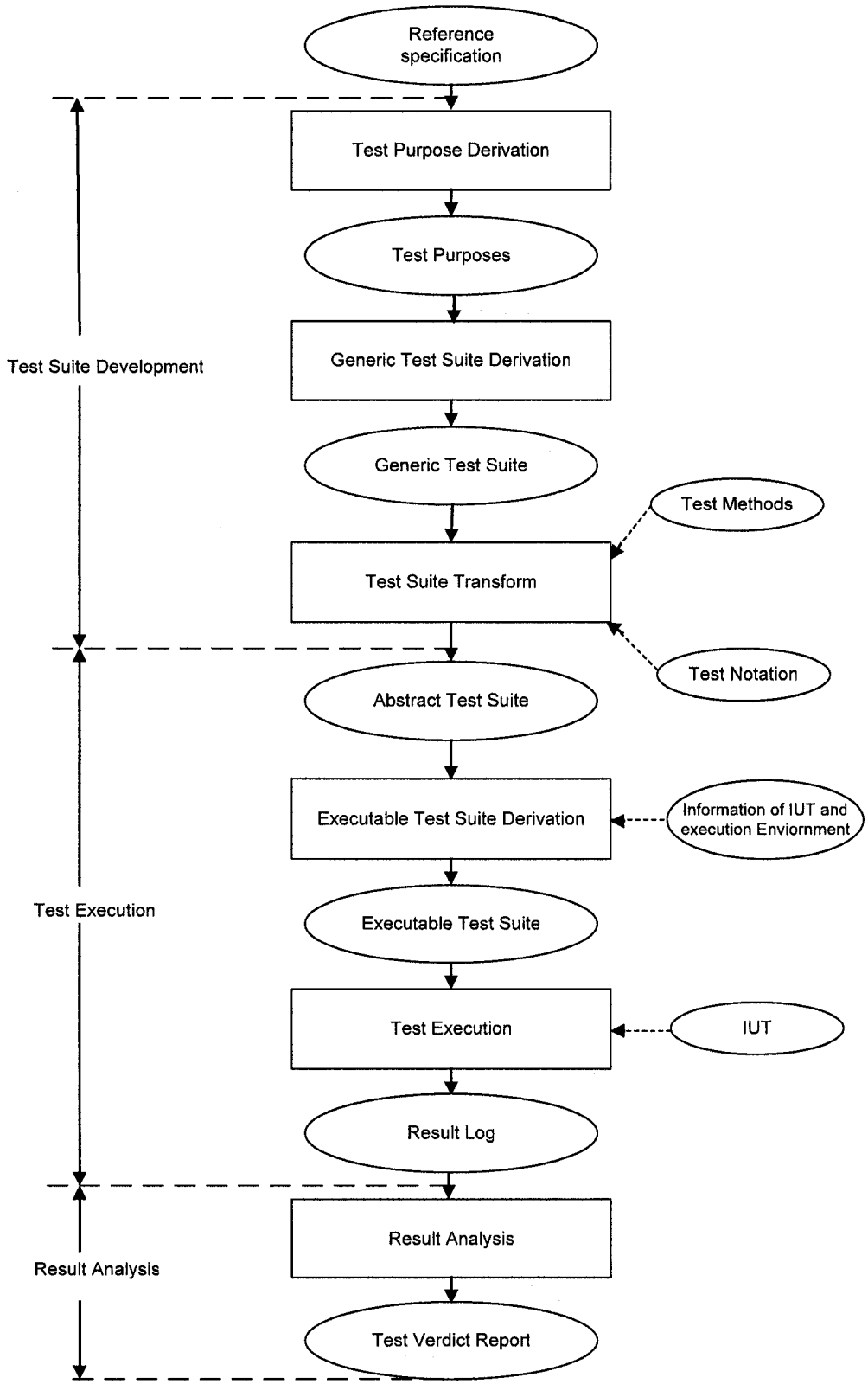

Fig. 2-1 Conformance Testing Framework

2.1.1 Conformance Testing Process

The standard (ISO IS-9646) defines a framework for conformance testing. It specifies the principle and general procedure of test suite generation, test execution, and test result analysis. The representation of test suite and test verdict is also specified in the standard. The standard does not specify tests for specific protocols, but recommends the general procedure of testing. It assumes that the natural language is used for specification.

Fig. 2-1 depicts an overview of ISO IS-9646 recommended conformance testing process. The whole process can be classified into three phases: test suite development, test execution, and result analysis.

Test suite development: A test suite is a set of test cases that represents the test for a specific test purpose. First of all, the test purposes are created from the specification. It represents what to test. It focuses on one or a group of conformance requirements specified in the specification. The conformance requirements represent the functionalities of the system, and they can be divided into groups. The related requirements can be described as a single test purpose to be tested.

Secondly, after test purposes are available, one generic test suite is generated from each specific test purpose. A generic test suite describes the high level test actions to achieve the specific test purpose, without considering any test methods or the execution environment.

Finally, an abstract test suite is derived from each generic test suite. The derivation is made by considering a particular test method and the constraints of the applied test

environment. The resulting abstract test suite is independent of any implementation. The test cases in an abstract test suite are represented in a well-defined test notation.

A semi-formal language to specify abstract test suite is suggested in the standard,, which is TTCN (Tree and Tabular Combined Notation) [23].

Test execution: Since the abstract test suites are independent of any real testing environment and *IUT*, before they can be applied in the real testing devices, the abstract test suite must be transformed into executable test suite. At this point, the implementation of the system under test should be considered. For example, test case could be represented as the parameters of a function, or the payload of a packet unit. To make the transformations, the information about the testing environment and the *IUT* must be supplied. Due to the knowledge that some options provided in the specification may not be implemented, some test cases in the abstract test suite could be irrelevant for the implementation. So, a selection is necessary to choose and refine the relevant test cases.

Once the executable test suites have been created and ready for execution, they are applied to the *IUT*. The observed outputs from the *IUT* are recorded in conformance logs.

Result analysis: The recorded reactions of *IUT* are compared with the reactions specified in the test suite, and a verdict report is created for the certification of the final product. A verdict is assigned to each test case according to the recorded outputs from *IUT*. A verdict is either PASS, INCONCLUSIVE or FAIL. If the outputs indicate that the implementation conforms to the specification and the test purpose, a PASS verdict is concluded. Otherwise, if the implementation fails to conform to the specification, a FAIL

verdict is concluded. INCONCLUSIVE verdict is concluded if the implementation conforms to the specification but the test purpose is not achieved.

2.1.2 Test Methods

A Test Method is an abstract model used to describe how the tester interacts with *IUT*. Test method specifies the accessibility of the *IUT* to the tester; it represents the logical concept of test architecture. The points where the tester can control and observe the *IUT* is called Points of Control and Observation (PCO). At PCO, test suites are applied to the implementation and the results of the test are observed. Several test methods are presented in the framework: the Local Single layer test method (LS-method), the Distributed Single layer test method (DS-method), the Coordinated Single layer test method (CS-method), and the Remote Single layer test method (RS-method).

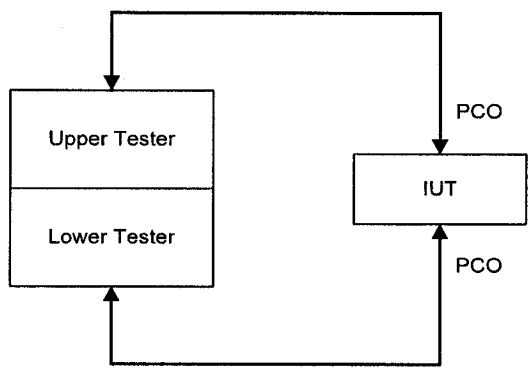

Fig. 2-2: Local Single Layer Test Method

As it is depicted in Fig. 2-2, in LS-method, an Upper Tester (UT) provides the observation and controls on the upper service boundary of *IUT*. A Lower Tester (LT) provides the observation and controls on the lower service boundary of *IUT*. There are

two PCOs in LS-method: the upper PCO is used by UT to send and receive signal to and from the implementation, and the lower PCO is used by LT.

The DS-method is used to test distributed systems as shown in Fig. 2-3. With this architecture, Upper Tester is at the same location as *IUT* and controls testing by upper PCO of *IUT*. Lower Tester is at the remote site and communicates with *IUT* by the PCO of the low layer service provider. UT and LT communicate with each other by Test Coordination Procedures, so that they can coordinate with each other to apply correct test suite to *IUT*, and record corresponding output traces.

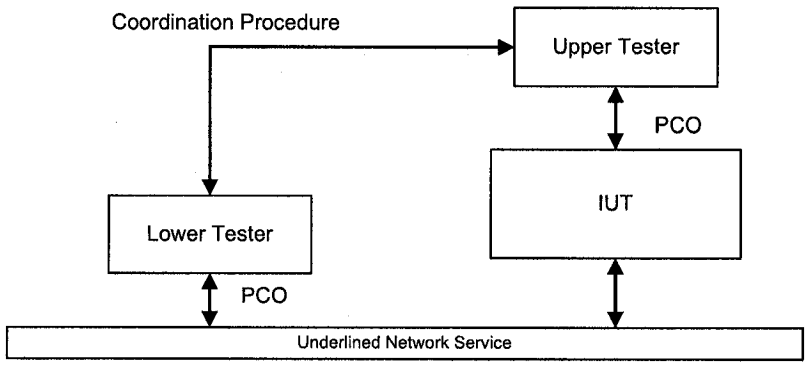

Fig. 2-3: Distributed Single Layer Test Method

Test Coordination Procedures between the Test System, LT and the UT are more rigorous. Explicit control and observation of ASPs (Abstract Service Primitive) is possible. ATS (Abstract Test Suite) makes explicit use of these ASPs. Test Coordination Procedures require either the use of a human operator or the use of a standardized programming language interface. In most cases a human user acts as the interface between the Test System and the UT. The interface is created using a system console and/or by telephone with the remote Test System operator.

Particular difficulties exist while testing with DS-method:

- The independence between UT and LT implies possible fault coverage limitations.
- Because of remote testing, queuing delays could cause remote tester to occupy incorrect time-related information from *IUT*.
- The separated testers could face synchronization problems.
- The external coordination between UT and LT could also face failures since the external environment is not guaranteed to be error free.

In the RS-method (Fig. 2-4), there is no upper tester, and only one PCO is available for the remote tester. A single PCO is located at the LT in the Test System and access to an upper tester is not formally required. Test Coordination Procedures between the Test System LT and the UT (if one exists) are implied or defined in an ad hoc manner. Typically coordination is achieved by a human user intervening at a system console.

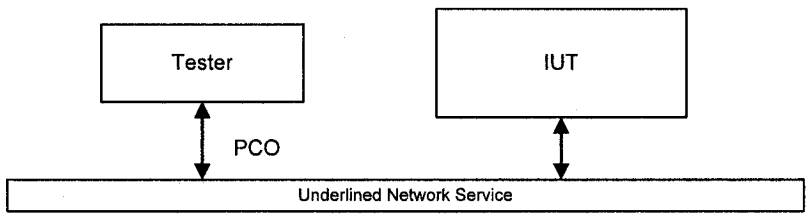

Fig. 2-4: Remote Single Layer Test Method

It may not always be possible to realize the required test coordination between the Test System and the *IUT* at the UT service boundary, however: It may be impossible to initiate some test cases from the LT because of Service Provider restrictions or limitations. It is the simplest ATM and makes no special demands on the *IUT*. It makes no assumptions about the internal design of the *IUT* or the *IUT* within Service Provider

PDUs. It essentially treats the SUT as a black box and the least intrusive of the methods presented.

In CS-method (Fig. 2-5), the UT and LT are separated. They coordinate and communicate with each other by Test Coordination Procedures, as the same in the DS-method. The difference between the two is that in CS-method, UT and LT are at the same location and local to the *IUT*; the Coordination Procedures are also local and could be an internal part of them.

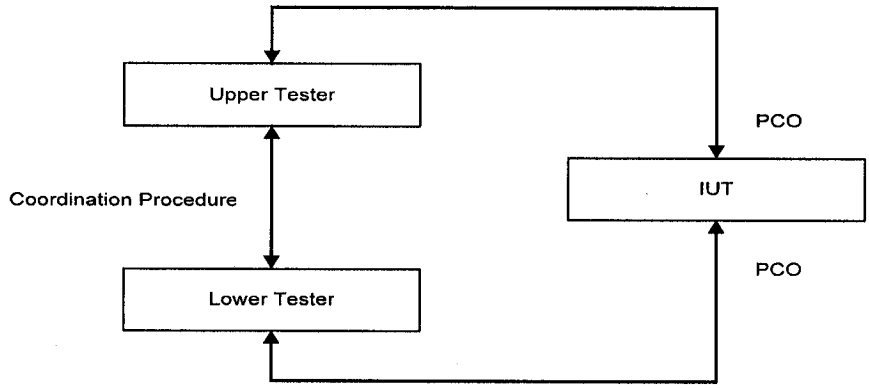

Fig. 2-5: Coordination Single Layer Test Method

These test methods as mentioned are usually used to test one layer *IUT*, but they can also be used to test multi-layer *IUT*. These layers can be tested as a whole, or one layer embedded in the other layers can be tested (embedded testing).

When abstract test suites are generated from test purpose, the test method is considered, so that the generated test suite can be adaptable for the future real test architecture. Again, test method represents the abstract logical concept of real test architecture. Before applying to the real test environment, the test method provides a way for correctly selecting and refining abstract test suites.

2.2 Conformance Testing and Specification

There are two types of testing, so-called black box testing and white box testing. In black-box testing, the tester has no knowledge of the internal structure of the implementation, only the reference specification is known. In this case, the test selection, fault coverage and test result analysis are done with respect to the reference specification.

In white box testing, the internal structure of the implementation is known. The knowledge of the implementation structure and reference specification is combined and used in test selection, fault coverage and test result analysis.

Another type of testing is gray-box testing, only high-level module structure of the implementation is known, but the structure details are not known.

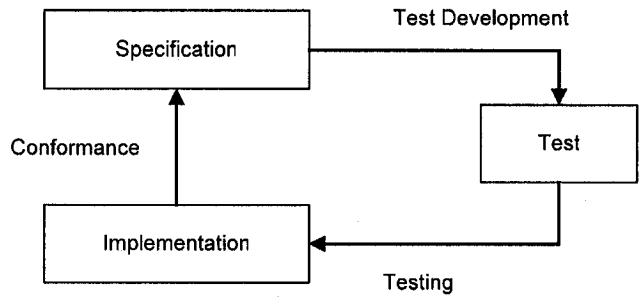

Fig. 2-6: Conformance Relationship

Conformance testing is black box testing. The implementation is required to conform to the reference specification. The test suites are derived from the specification, and then applied to the implementation. In this way, the implementation is tested against the specification, and the conformance between the implementation and the specification is verified (as shown in Fig. 2-6). While the specification is specified by the means of formal models, it is possible to generate test suite automatically from these formal models by some kind of generation algorithms. Compared to the manual test suite generation,

automatic generation algorithm from formal models has obvious advantages, such as no human introducing errors, more fault coverage, and lower test costs.

2.3 Test Hypotheses

To achieve completely error free implementation by testing, we have to do exhaustive testing. It means applying infinite test cases to the implementation to cover all possible faults. It is obvious that infinite test cases are impossible. Due to the real environment limitation and the goal of lower test cost, exhaustive testing can never be realized, even for the white-box testing where all the details of the internal structure are known. However, some assumptions are always true when we have some knowledge about implementation, by, which we can limit and lower the possible test cases. Those assumptions and knowledge about implementation are called Test hypotheses. For example, when we test a system, we really know what the system does, and we know that it does not do something irrelevant. Moreover, when we are testing the implementation with respect to the reference specification given in a formal model, we assume that the system is implemented based on the adopted model. These assumptions make a conformance testing possible.

The purpose to use test hypotheses is to reduce the possible implementations and test cases. One way to achieve that purpose is by using test purpose. Test purpose specifies what functionalities of the implementation to be tested. Single or partial requirements and functions specified in the specification are derived to be tested. Therefore, not all the requirements, but only a finite set of functions are considered. As

we have discussed in section 2.1, ISO conformance testing framework advises that the test suites are created from the test purposes.

2.4 Fault Model

Since we cannot cover all faults by a test, how can we say a test is good or not? It is necessary to define specific criteria that can evaluate the quality of a test. Fault Model does the job. A fault model is used for describing how faults affect the behaviors of the implementation. In a conformance testing, a test suite is said to have a complete fault coverage with respect to a given fault model, when it either satisfies the conformance relation, or there exists a test case in it, which results in a verdict FAIL[6].

A fault model describes the possible high level abstract faults of an implementation. Because a single fault can create various errors in the *IUT*, fault model provides the clear clues where those errors can be form. When the system is specified by a formal model, fault model is used to describe what kind of faults could happen based on the formal model. Therefore, test suites can be created to cover those faults.

Most formal models used to describe the behaviors of distributed and real time systems are based on Finite State Machines (FSMs) or Input Output Automaton (IOA). The FSM fault model includes [24]:

Output faults: The implementation machine provides a different output from one specified by the output function in the specification.

Transition faults: For a given state and input, the transition of the implementation machine arrives at a different state from the one specified by the specification.

Additional or missing transition faults: These are additional transitions for the corresponding pair of states, or a transition is missed in the implementation machine. This is considered an error for deterministic machines, where only one transition is allowed for a given input and state.

Additional states faults. The implementation machine enters into a state, which is not specified by the specification model.

2.5 FSM-Based Test Suite Generation

To cover the faults described in the fault model based on FSM, several test case generation methods have been developed. All these methods are based on the assumptions. Firstly, the implementation machine is completed and it has limited number of extra states; Secondly, the implementation FSM is deterministic, which means that for a given input and a pair of states, there is only one transition; And finally, there exists a reset function, which can set the implementation machine to the initial state. These test derivation methods are described as following:

Transition Tour (TT-method) [25]: The TT-method generates a test sequence called "transition tour". For a given FSM, a transition is a sequence, which takes the FSM from an initial state, traverses every transition at least once, and returns to the initial state.

The TT-method allows the detection of all output errors but there is no guarantee that all transfer errors can be detected. This method has a limited error detection power compared to other methods since it does not consider state checking. However, an advantage of this method is that the test sequences obtained are usually shorter than test sequences generated by the other methods.

Distinguishing Sequence (DS-method) [26]: A distinguishing sequence (DS) is used as a state identification sequence. An input sequence is a DS for an FSM, if the output sequence produced by the FSM is different when the input sequence is applied to each different state. The test sequences obtained by the DS method guarantee to identify a particular FSM from all other FSMs. It has a full fault coverage (detecting both transfer and output errors). However, the disadvantage of this method is that a DS may not be found for a given FSM (as one single sequence is the UIO for all states). Also applying a fixed length sequence may not lead to the shortest state identification sequence.

Unique Input Output (UIO-method) [27]: A UIO sequence for a state is an input/output behavior that is not exhibited by any other state. For a test, the input portion of distinguishing sequence is the same for all states; different states are distinguished by distinct outputs. For the UIO sequence, the input portion is normally different for each state. When the input part of the UIO sequence for a certain state s is applied to the FSM, the output sequence is compared with the expected output sequence. If they are the same, then the FSM was in state s. On the other hand, when the input portion of the distinguishing sequence is applied to the FSM, the outputs contain sufficient information to decide not only whether the machine was in state s, but also if not s, then, which one. UIO sequences are typically shorter than the distinguishing sequences.

UIO sequences are also different from the so called characterization set generated using the W-method. The characterization set consists of input sequences, which distinguish between every pair of states. On the other hand, the UIO sequence for a state distinguishes it from all other states. Again, this method cannot guarantee full fault coverage.

W-method [28]: This method includes two sets of input sequences: W-set, which consists of input sequences that can distinguish between every pair of states; and P-set, which consists of input sequences that take the machine from the initial state to a given state for a given transition. Test sequences are formed by concatenating these two sets: It provides a way to test all misbehaviors of the implementation machine. The logic of w-method is that the prefix input sequences bring the FSM to an identified state, and then the specific transition from the state is tested. This method guarantees the detection of all faults, but it cannot guarantee the existence of W-set for a given FSM.

Wp-method [29] W_p-method has the same detection power as W-method. The main advantage of the Wp-method over W-method is that the length of test suite is reduced. Instead of using a W-set to check each reachable state, only a subset of W-set is used. Another advantage is that Wp-method is always applicable.

Those test derivation methods are based on FSM as formal model. In conformance testing, the formal models of the specification are used to derive test suites. However, these methods have well known problem of state space explosion. The length of test sequence and the number of test cases could be too high so that the test cost is too high. That makes real world testing very difficult. Test purposes can be used to solve this problem by testing only partial specifications, and only specific faults are targeted to be expressed in formal models.

2.6 Summary

The complete automation of the testing process requires the use of formal methods for providing a model of the required system behavior. In this chapter, we defined conformance testing framework and discussed different test methods and shown the importance of modeling the aspects to be tested (the right model for the right problem).

In the next chapter, we will introduce the implementation architecture CORBA, which is our platform for implementing of timed test case execution.

Chapter 3

CORBA

In this chapter, we investigate the conformance testing of objects that sit on top of a CORBA platform and interact via the object request broker, e.g. an implementation of a particular CORBA object service. Section 3.1 introduces the overview of CORBA. Section 3.2 describes the real time CORBA platform and finally Section 3.3 represents the CORBA implementation model by Borland's Visibroker.

3.1 Overview of CORBA

CORBA is a standard promoted by the OMG consortium (*Object Management Group*), which is widely accepted in the telecommunication industry, and the main actors of this sector have made large investments in this technology.

The CORBA standard defines a general model for distributed programming addressing heterogeneity at different levels (hardware, operating systems, networks and protocols, programming languages). CORBA can be viewed as a software infrastructure allowing the interconnection of heterogeneous pieces of software. It is based on an object-oriented approach and on the client/server model.

The core level of a CORBA implementation is called an ORB (*Object Request Broker*). CORBA objects are described in a pivot language called IDL (Interface Definition Language) CORBA. Unlike other programming languages, such as C++ and Java, IDL is a declarative language to define object interface in a manner that is

independent of any particular programming language. With different available IDL compliers, IDL can be translated to different programming languages, such as C, C++, Java, Smalltalk and Ada. The standard language mapping makes it feasible that the developer can implement different portions of a distributed system in different languages. For example, based on CORBA technology, a high-throughput server application may be written in C++ for efficiency and client can be developed using Java Applet. The language independence of CORBA is the key to its value as an integration technology for heterogeneous systems.

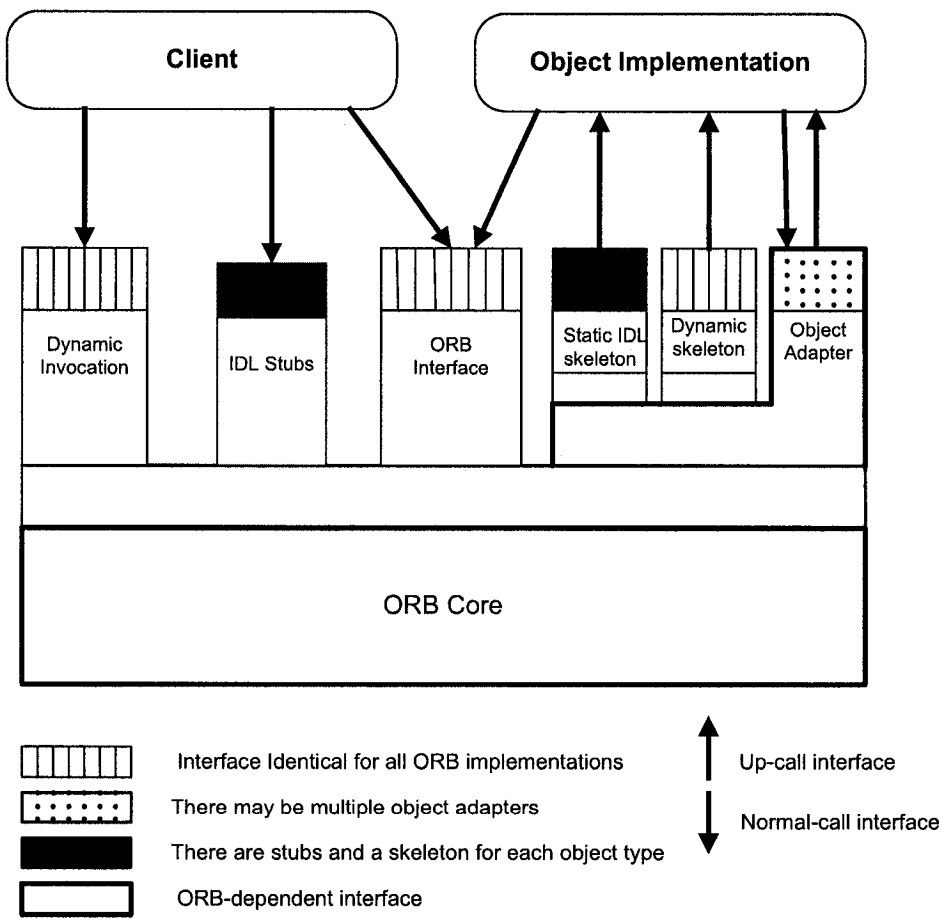

Fig. 3-1: The Structure of Object request Interface

An IDL CORBA definition describes the interface of software and not a complete implementation. The IDL definitions are stored in an interface repository. This repository is mainly used to dynamically construct a method invocation (dynamic invocation interface).

The choice of CORBA as a test environment is motivated by the following reasons: (1) The telecommunication companies are moving to CORBA; (2) CORBA is a distributed environment guarantying the transparency of resource location and communication, despite heterogeneity; (3) CORBA is well fit for a modular architecture.

Fig. 3-1 shows the main components of CORBA and the interconnections [32]. Here, client is the entity that wishes to perform an operation on the object and the server is the code and data that actually implements the object. A client invokes a method of an object by issuing a request to the object through an interface. The request carries information including an operation, the object reference of the service provider.

The interfaces of the clients are completely independent of where the object is located, what programming language it is implemented in, or any other aspect, which is not reflected in the object's interface. CORBA interfaces are defined in IDL. This language defines the types of objects according to the operation that may be performed on them and the parameters to those operations [33].

3.2 Real-Time CORBA

A growing class of real-time systems requires end-to-end support for various qualities of service (QoS) aspects, such as bandwidth, latency jitter and dependability.

Real-time CORBA is an optional set of extensions to CORBA tailored to equip ORBs to be used as a component of a real-time system [34].

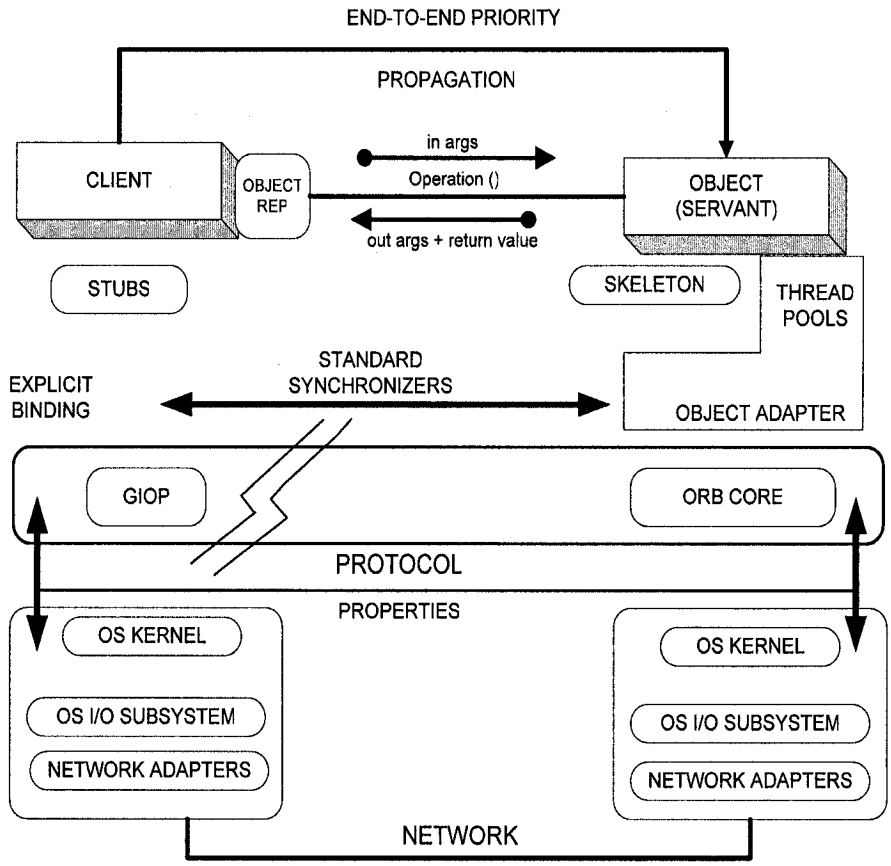

Fig. 3-2: ORB end system features for Real-Time CORBA

The goals of Real-time CORBA specification is to support developer in building predictable distributed system, a prerequisite for ensuring real-time performance, by providing mechanisms to control the use of processor, memory, and network resources.

As shown in Fig.3-2 an ORB endsystem [36] consists of network interfaces, operating system I/O subsystems and communication protocols, and CORBA-compliant middleware components and services. The RT-CORBA specification identifies capabilities that must be *vertically* (i.e. network interface ↔ application layer) and *horizontally* (i.e., peer to peer) integrated and managed by ORB endsystems to ensure

end-to-end predictable behavior for activities that flow between CORBA clients and servers. We outline these capabilities, starting from the lowest level of abstraction and building up to higher-level services and applications are given below:

(1) Communication infrastructure Resource Management

An RT-CORBA endsystem must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from (a) managing the choice of the connection used for a particular invocation to (b) exploiting advanced QoS features, such as controlling the ATM virtual circuit cell pacing rate.

(2) OS scheduling mechanism:

ORBs exploit OS mechanisms to schedule application-level activities end. Since the RT-CORBA 1.0 specification targets fixed priority real-time systems, these mechanisms correspond to managing OS thread scheduling priorities. The RT-CORBA specification focuses on operating systems that allow applications to specify scheduling priorities and policies.

(3) Real-Time ORB endsystem:

ORBs are responsible for communicating requests between clients and servers transparently. A real-time ORB endsystem must provide standard interfaces that allow applications to specify their requirements to the ORB. The policy framework defined by the OMG Messaging specification [36] allows applications to configure ORB endsystem resources, such as thread priorities, buffers for message queuing, transport-level connections, and network signaling, in order to control ORB behavior.

(4) Real-time services and applications:

Having a real-time ORB manage endsystem and communication resources only provides a partial solution. Real-time CORBA ORBs must also preserve efficient, scalable, and predictable end-to-end behavior for higher level services and application components. For example, a global scheduling service can be used to manage and schedule distributed resources. Such a scheduling service can interact with an ORB to provide mechanisms that support the specification and enforcement of end-to-end operating timing behavior.

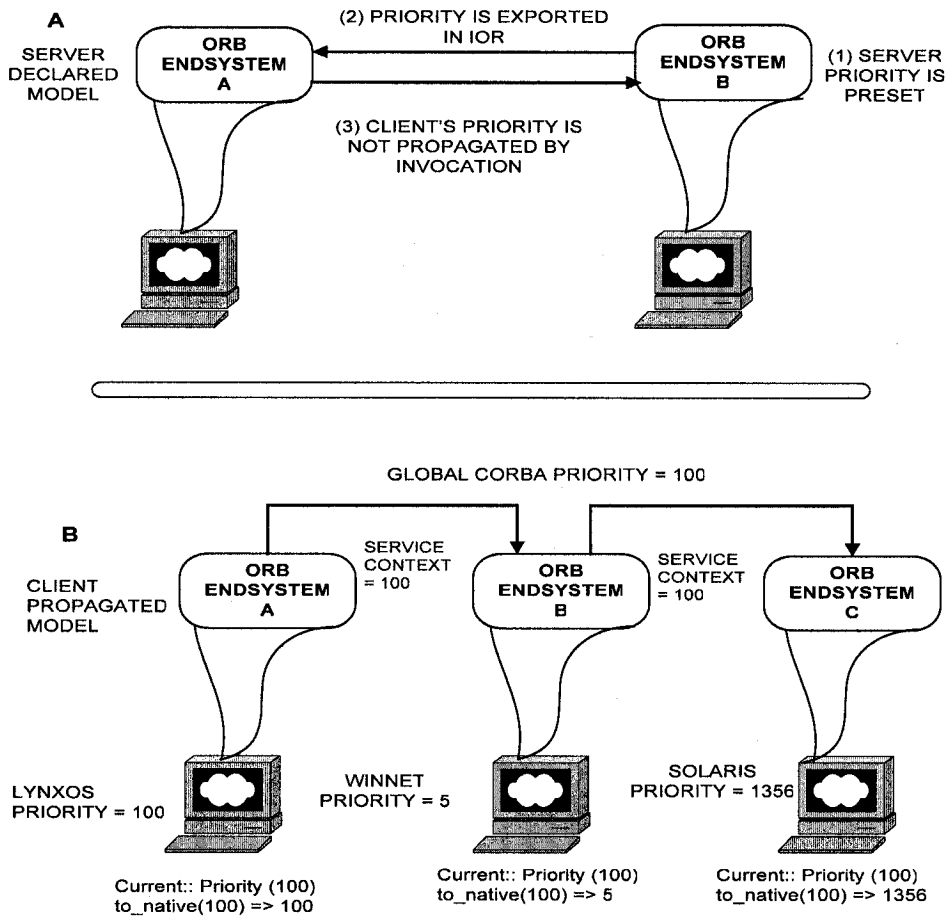

Fig. 3-3: Real-time CORBA priority models

To manage these capabilities, RT-CORBA defines standard interfaces and QoS policies that allow applications to configure and control (1) *Processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory* resources via buffering requests in queues and bounding the size of thread pools.

Priority mechanisms: The RT-CORBA specification defines a *PriorityModel* policy with two values, SERVER DECLARED and CLIENT PROPAGATED, as shown in Fig. 3-3.

This model allows a server to dictate the priority at, which an invocation made on a particular object will execute. In the server declared model, the priority is designated a priori by a server based on the value of the *PriorityModel* policy in the POA (Portable Object Adapter) where the object was activated. A single priority is encoded into the object reference, which is then published to the client as a tagged component in an object reference as shown in Fig.3-3(A). Although the server declares the priority, the client ORB is aware of the selected priority model policy and can use this information internally.

Thread Pools: Many embedded systems use multi-threading to distinguish between different types of services, such as high-priority vs. low-priority tasks and support thread preemption to prevent unbounded priority inversion. To address the concurrency issue therefore, the RT-CORBA specification defines a standard *thread pool* model. This model allows server developers to pre-allocate pools of threads and to set certain thread attributes, such as default priority levels. Thread pools are useful for real

time ORB end systems and applications that want to leverage the benefits of multi-threading, while bounding the amount of memory resources, such as stack space, they consume.

RT-CORBA provides an interface that allows server developers to pre-allocate an initial number of so-called *static* threads, while allowing this pool to grow dynamically to handle bursts of client requests. If a request arrives and all existing threads are busy, a new thread may be created to handle the request. No additional thread will be created, however, if the maximum number of threads in the pool has been spawned.

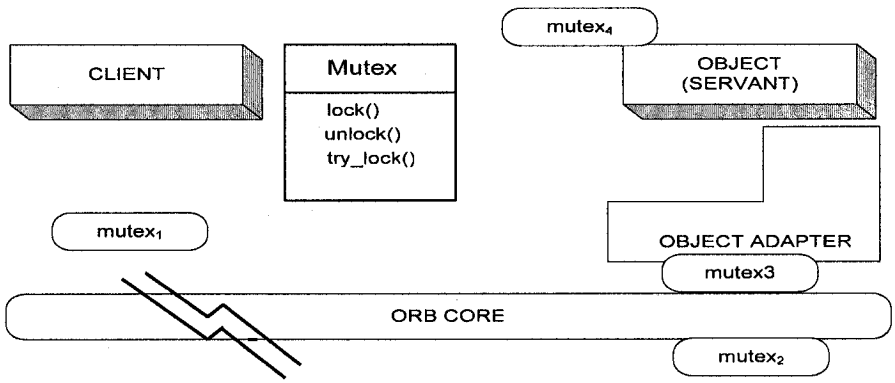

Fig. 3-4: Standard Synchronization

Standard Synchronizers: The CORBA specification does not define a threading model. Thus, there is no standard, portable API that CORBA applications can use to ensure semantic consistency between their synchronization mechanisms used by an ORB. Real time applications, however, require this consistency to enforce priority inheritance and priority ceiling protocols.

Global Scheduling: More intuitive manner that the RT-CORBA specification defines a global scheduling service. This service is a CORBA object that is responsible for allocating system resources to meet the QoS needs of the applications that share the

ORB end system. Applications can use the real-time scheduling service to specify the processing requirements of their operations in terms of various parameters, such as worst-case execution time or period, as shown in Fig. 3-5.

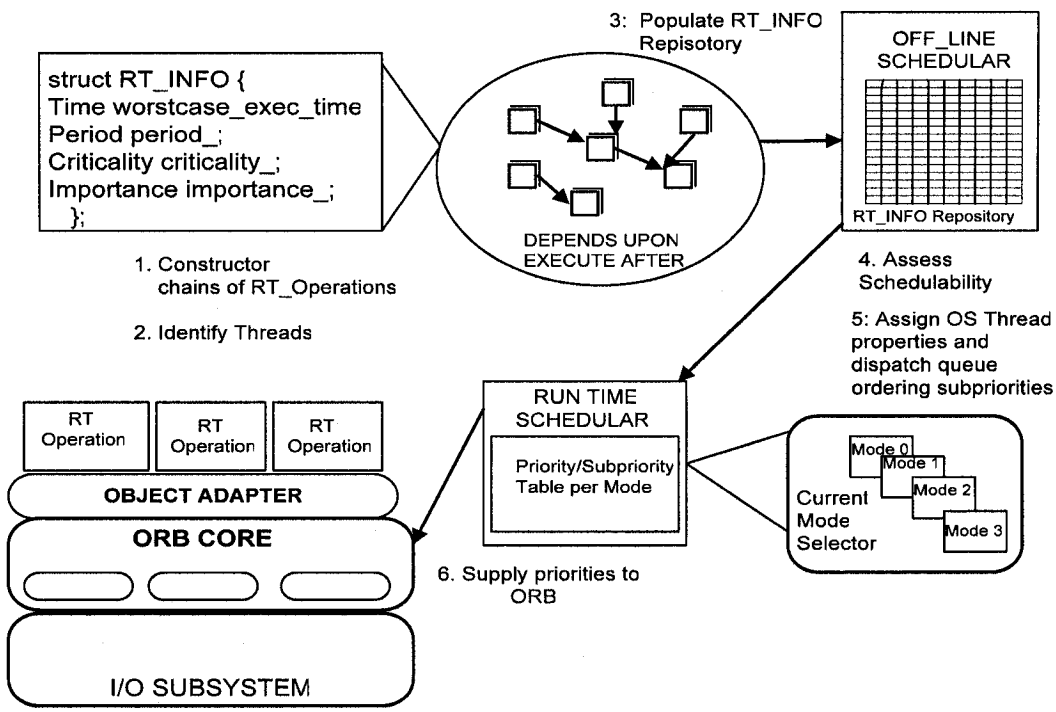

Fig. 3-5: Real-Time CORBA global scheduling service

Selecting and Configuring Protocol Properties: CORBA uses inter-ORB communication mechanisms to exchange requests between clients and servers. These mechanisms are built upon lower level protocols that provide various types of QoS.

Explicit Binding: This mechanism enables clients to (1) pre-establish connections to servers and (2) control how client requests are sent over these connections. The following two policies – *priority-banded* and *private connections* – are defined to support explicit binding in RT-CORBA.

3.3 CORBA Implementation Model

VisiBroker for Java provides a complete CORBA 2.3 ORB runtime and supporting development environment for building, deploying, and managing distributed Java applications that are open, flexible, and inter-operable. VisiBroker has a built-in implementation of IIOP that ensures high-performance and inter-operability. VisiBroker for Java has several key features as described in the following sections.

3.3.1 Visibroker Smart Agent Architecture

VisiBroker's Smart Agent (`osagent`) is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Multiple Smart Agents on a network cooperate to provide load balancing and high availability for client access to server objects. The Smart Agent keeps track of objects that are available on a network, and locates objects for client applications at invocation time. VisiBroker can determine if the connection between client application and a server object has been lost, due to an error such as a server crash or a network failure. When a failure is detected, an attempt is automatically made to connect the client to another server on a different host, if it is so configured.

If the PERSISTENT policy is set on the POA, and `activate object with id` is used, the Smart Agent registers the object or implementation so that it can be used by client programs. When an object or implementation is deactivated, the Smart Agent removes it from the list of available objects. Like client programs, the communication with Smart Agent is completely transparent to the object implementation.

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production versions of client programs and object implementations while another domain might be made up of test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want to establish their own ORB domain so that their testing efforts do not conflict with one another.

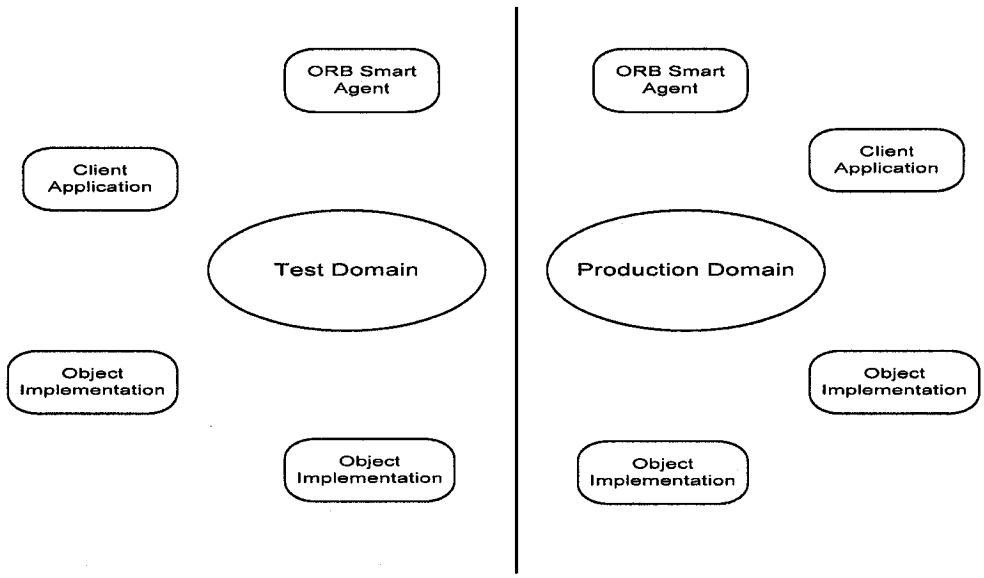

Fig. 3-6: Running separate ORB domains simultaneously

VisiBroker allows us to distinguish between multiple ORB domains on the same network by using a unique UDP port number for the Smart agents for each domain. By default, the `OSAGENT_PORT` variable is set to 14000. If we wish to use different port numbers, our system administrator check and determine, which port numbers are available. To override the default setting, the `OSAGENT_PORT` variable must be set

accordingly before running a Smart Agent, an OAD, object implementations, or client programs assigned to that ORB domain.

3.3.2 Robust Thread and Connection Management

VisiBroker provides native support for single and multithreading thread management. With VisiBroker's thread-per-session model, threads are automatically allocated on the server-per-client connection to service multiple requests, and then are terminated when the connection ends. With the thread pooling model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client will be serviced by multiple threads-ensuring that the requests are quickly executed. Less active clients can share a single thread, and still have their requests immediately serviced.

VisiBroker's connection management minimizes the number of client connections to the server. All client requests for objects residing on the same server are multiplexed over the same connection, even if they originate from different threads. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the same server. The size of the thread pool grows based upon server activity and is fully configurable-either before or during execution-to meet the needs of specific distributed systems. With thread pooling, we can configure the following:

- Maximum and minimum number of threads
- Maximum idle time

Each time a client request is received, an attempt is made to assign a thread from the thread pool to process the request. If this is the first client request and the pool is empty, a thread will be created. Likewise, if all threads are busy, a new thread will be created to service the request.

A server can define a maximum number of threads that can be allocated to handle client requests. If there are no threads available in the pool and the maximum numbers of threads have already been created, the request will block until a thread currently in use has been released back into the pool.

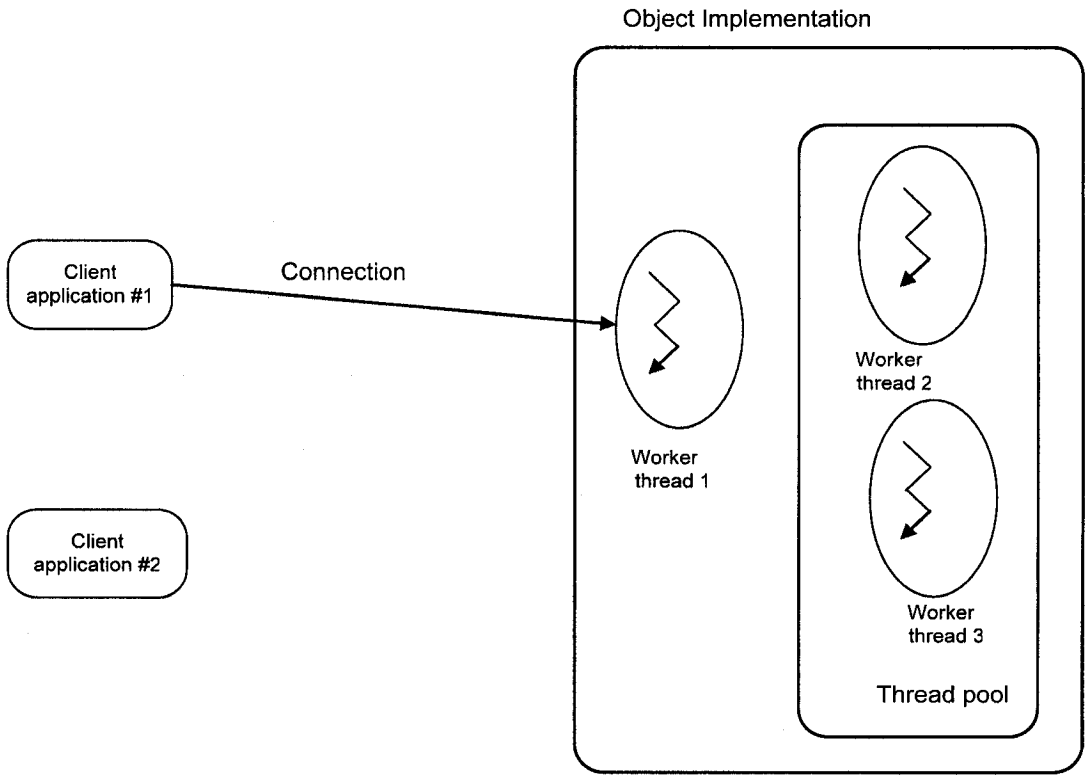

Fig. 3-7: Thread connection management

Fig. 3-7 shows, client application #1 establishes a connection to the Object Implementation and a thread is created to handle requests. In thread pooling, there is one connection per client and one thread per connection. When a request comes in, a worker thread receives the request and then it is no longer in the pool. A worker thread is removed from the thread pool and is always listening for requests. When a request comes in, worker thread reads the request and dispatches the request to the appropriate object implementation. Prior to dispatching the request, the worker thread wakes up one other worker thread, this then listens for the next request.

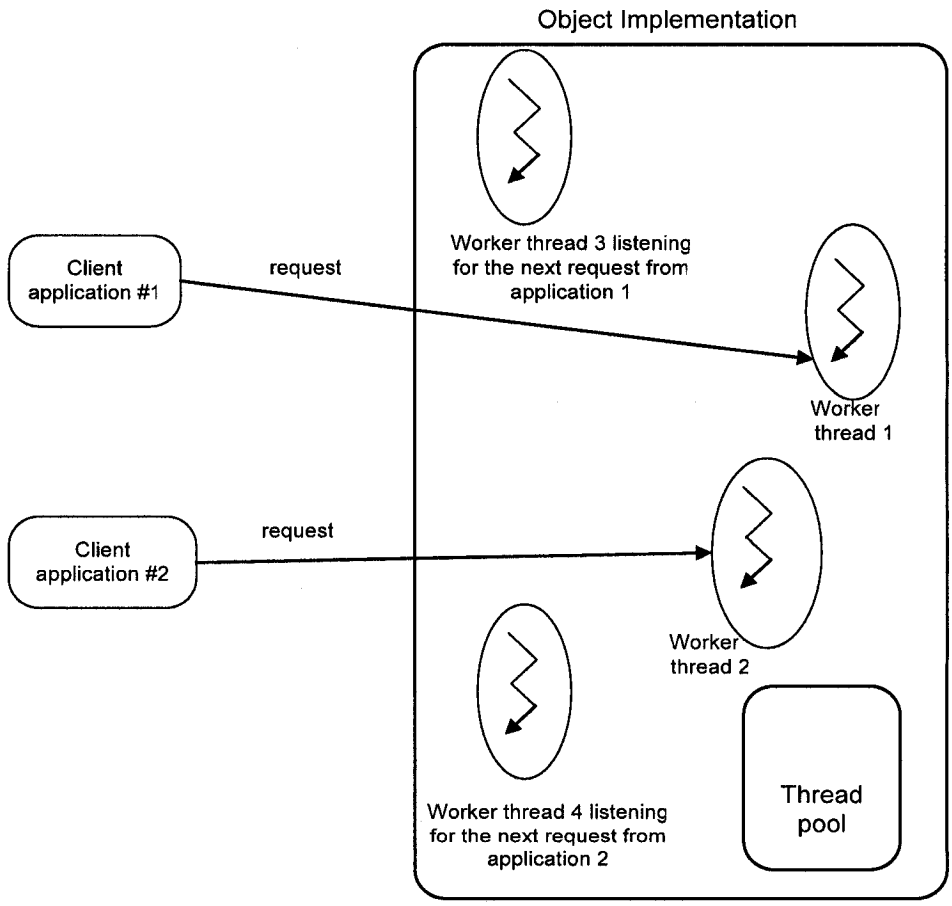

Fig. 3-8: Client application #2 establishes connection

Fig. 3-8 shows, when Client application #2 establishes its own connection and sends a request, a second worker thread is created. Worker thread #3 is now listening for incoming requests. If more requests came in from Client application #1, more threads would be assigned to handle them-each spawned after the listening thread receives a request. As worker threads complete their tasks, they are returned to the pool and become available to handle requests from any client.

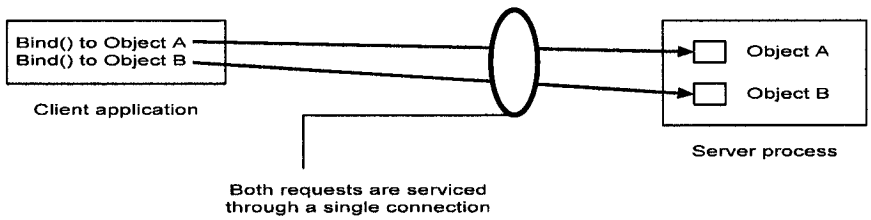

Fig. 3-9: Binding to two objects in the same server process

In Fig. 3-9, a client application is bound to two objects in the server process. Each `bind ()` shares a common connection to it, even though the `bind ()` is for a different object in the server process.

3.4 Summary

CORBA is a vendor independent specification designed to let different platforms, operating systems and compilers work together. The goals of real-time CORBA specification is to support developer in building predictable distributed system, a prerequisite for ensuring real-time performance by providing mechanisms to control the use of processor, memory and network resources. Overall, VisiBroker's connection management minimizes the number of client connections to the server. In other words

there is only one connection per server process, which is shared. All client requests are multiplexed over the same connection, even if they originate from different threads. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the server.

In the next chapter, we provide centralized and distributed test methods with their architectures.

Chapter 4

Test Case Execution Based on np-TIOA

Two new methods are presented to do test case execution in this chapter. We consider that n-port TIOA is used for the specification of *IUT* and the execution of the test in order to determine whether *IUT* conforms to the specification. Therefore, transition tour method is used for the generation of test cases. Section 4.1 introduces an overview of the methodology for testing distributed real-time system. Section 4.2 describes the timed test case execution framework step by step. Section 4.2.1 presents the model of input-output Timed Automata with n ports (*np-TIOA*) for centralized and distributed test system. Section 4.3 gives the detailed architecture of the centralized test system and algorithm. Section 4.4 discusses the distributed test system architecture and algorithm. Finally, section 4.5 gives the summary.

4.1 Overview of the Methodology

Testing consists of generating test sequences and applying them to an implementation, which is known as Implementation Under Test (*IUT*).We consider distributed real-time systems as *IUT*. Similar to [9, 14, 15, 16], we consider that a distributed system *(IUT)* has several ports and that it is reactive. As in [1], we assume during test execution, controllability represents the ability of test system *(TS)* to

guarantee timing constraints of inputs, and observability represents the ability of *TS* to check timing constraints of outputs, which are sent from *IUT*.

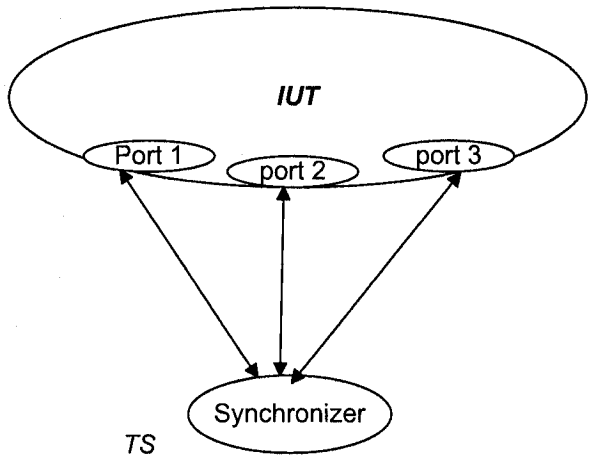

Fig. 4-1: Centralized test architecture for testing distributed system

By centralized test method, we consider a *Synchronizer* as a single tester in *TS*, which communicates with different ports of *IUT* (as shown in Fig. 4-1). Our target is to study the phase of test execution i.e. the phase of checking test sequences, which is sent from *Synchronizer* to *IUT* and to conform every given test sequence. The consequence of conformance testing returns a verdict *pass*, *fail* or *inconclusive*, which has been described in proposed test procedure in section 4.3.

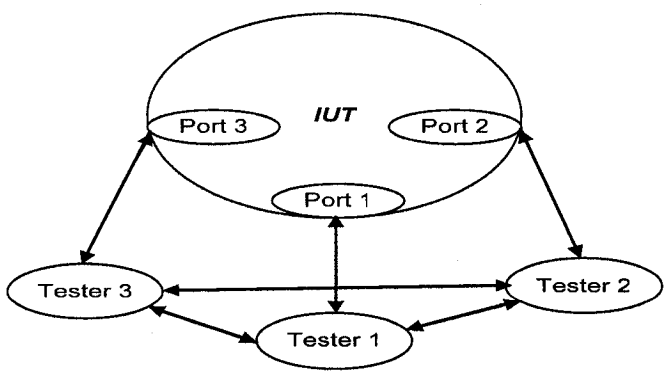

Fig. 4-2: Distributed Test Architecture

By distributed test method (Fig. 4-2), we consider a test system, which consists of several testers communicating through each port of the distributed *IUT* and also sending coordination messages among other testers. During coordination, message O is used to guarantee order constraints of inputs and message T is used to guarantee timing constraints of inputs.

4.2 Testing Process

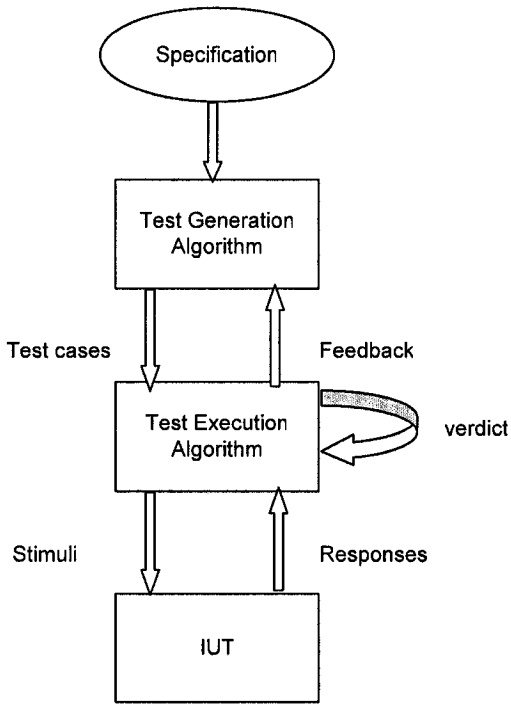

Fig. 4-3: Step by step testing process

In this process (Fig. 4-3), the specification is the input of the test case generation algorithm. The specification describes the actions that the system is allowed to do. Using it, the algorithm produces test cases, which are taken by the test system and executed against the Implementation Under Test (*IUT*). The tester and the *IUT* exchange stimuli

and responses. If one of the executions leads to an error, the verdict will be *fail*. If no error is discovered the verdict will be *passed*. If the tester gives feedback to test generation algorithm, which will be used for building up the test cases, the test derivation is called on-the-fly (test generation and test execution are combined in one phase) procedure. In any other case, it is called batch oriented (test generation and test execution are separated phase) procedure. The correctness of an implementation with respect to a specification is checked by executing test cases (which specifies a behavior of the implementation under test). Now we describe the step by step of the testing process as in Fig. 4-3.

4.2.1 Input-output Timed Automaton with n Ports (np-ioTA)

In this section, we will introduce the concepts and formal models that are used in our methods. This is our first step in testing process that would be FSM based specification.

Our timed automaton operates with finite control – a finite set of locations and a finite set of real valued clocks. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were started (or reset). Each transition of the automaton has an associated constraint on the values of the clocks that satisfy this constraint. A transition, in addition to changing the location of the automaton, may reset some of the clocks.

4.2.1.1 Definition of Centralized n-port TIOA Model

As in [1], we consider a model called input-output Timed Automata with n ports (np-ioTA) for describing the specification. An np-ioTA is a 7-tuple ($£, K, I, O, \delta, \lambda, l$) *where* $n \geq 1$ *and:*

- $£$ is a finite set of states and $l_0 \in £$ is the initial state.
- K is a finite set of intervals bounded by positive values. Every element of K is written in the form $[a, b]$ and denotes the set $\{x \mid a \leq x \leq b\}$.
- I is a n-tuple $(I_1, I_2, \ldots\ldots I_n)$, where I_i is a finite set of all the inputs of I, $I_i \cap I_j = \Phi$ *for* $i \neq j$ *and* $i,j = 1,\ldots\ldots,n$. Let then $I = (I_1 X K) U (I_2 X K) U \ldots\ldots\ldots U (I_n X K)$.
- O is a n-tuple $(O_1, O_2, \ldots\ldots O_n)$, where O_i is a finite set of all the outputs of i, $O_i \cap O_j = \Phi$ *for* $i \neq j$ and $i,j = 1,\ldots\ldots,n$. Let then $O = (O_1 X K) U \{\varepsilon\} X (O_2 X K) U \{\varepsilon\} X\ldots\ldots\ldots X (O_n X K) U \{\varepsilon\}$, where ε stands for the empty output.
- δ is a transition function : $D \rightarrow £$, and λ is an output function: $D \rightarrow O$, where $D \supseteq £ X I$.

A transition T_r of a np-ioTA is therefore defined by $\delta (q, \sigma, K) = r$ and $\lambda (q, \sigma, K) = ((s_1, K_1),(s_2,K_2),\ldots\ldots\ldots\ldots\ldots\ldots(s_n, K_n))$ where:

- q and r are origin and destination states of the transition, respectively;
- σ is the input of the transition and the interval K defines the timing constraint of σ.
- $s_1, s_2, \ldots\ldots\ldots\ldots\ldots, s_n$ are the outputs of the transition, in ports $1,2,\ldots\ldots n$, respectively, and each interval K_i defines the timing constraint of s_i. When the

transition has no output in port i, then (s_i, K_i) is not defined and is represented by ε.

- If q is the current state and σ is received at an instant, which falls within K relatively to the instant when q has been reached.

  Then for each $i = 1,.....n$: s_i (if any) is sent in port i at an instant, which falls within K_i relatively to the instant when σ has been received. State r is reached immediately after that all the outputs s_i have been sent. If T_r contains no output then state r is reached immediately after the reception of σ.

- If T_r is the initial transition of an execution, then K is ignored. Note that K is not ignored when T_r is not the initial transition of an execution even if its origin state is l_0.

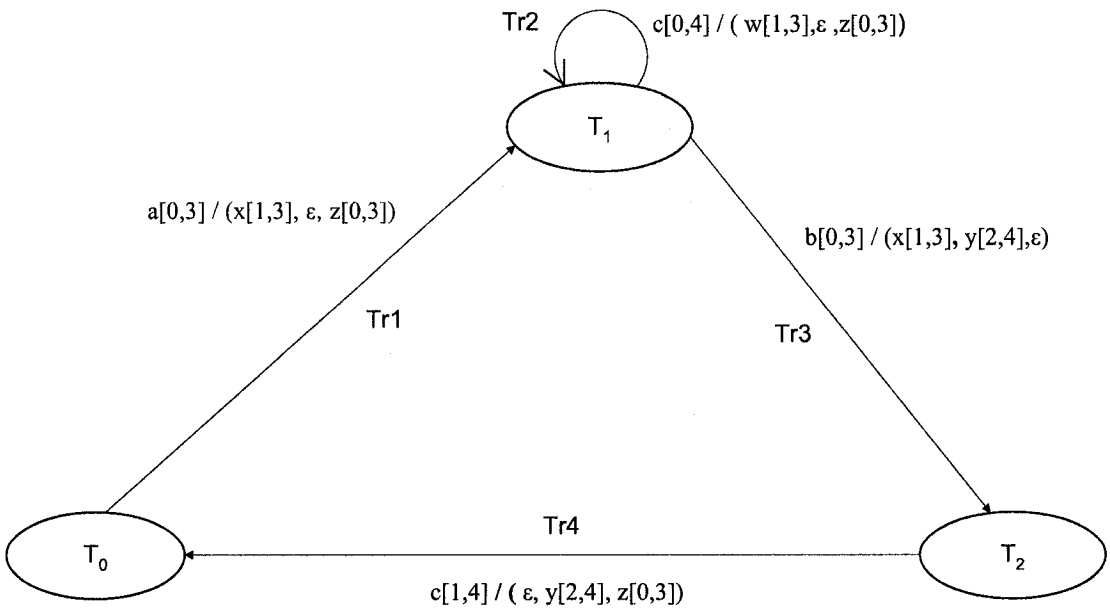

Fig. 4-4: n-port TIOA for centralized test method

An example of np-io TA is represented in above Fig. 4-4: With $I_1 = \{a\}$, $I_2 = \{b\}$, $I_3 = \{c\}$, $O_1 = \{w, x\}$, $O_2 = \{y\}$, $O_3 = \{z\}$. The nodes are the states and the directed edges are the transitions linking the states. A label $\sigma K / (s_1K_1, s_2K_2, \ldots\ldots, s_nK_n)$ on an edge linking t and r means $\delta(q,\sigma,K) = r$ and $\lambda\ (q, \sigma, K) = ((s_1, K_1),(s_2,K_2), \ldots\ldots\ldots\ldots (s_n, K_n))$. For example, if the current state T_1 has been reached at instant τ and the input a is received at an instant τ_1 such that $(\tau_1 - \tau) \in [0, 3]$, then the transition Tr_3 is executed, which means that the state changes to T2 and the outputs x and y are sent in ports 1 and 2 at instants v1 and v2, respectively, such that $(v1 - \tau_1) \in [1, 3]$ and $(v2 - \tau_1) \in [2, 4]$. Therefore the state T2 is reached at instant $v = sup\ (v1, v2)$. The transition Tr4 is executed if and only if c is received at an instant t such that $(t - v) \in [1,4]$.

We consider the test sequence, which is derived by **Transition Tour method** Tr1.Tr2.Tr3.Tr4.

Input: *a [0, 3].c [1, 4].b [0, 4].c [1, 4]*

Expected Output: *(x [1, 3], ε, z [0,3]).(w [2,5], ε, z [0,3]). (x [1,3], y [2,4], ε). (ε, y [2,4], z [0,3])*

Inputs a, b, c are received by *IUT* in ports 1, 2, 3 respectively, and outputs x, y, z are sent by *IUT* in ports 1, 2, 3, respectively.

4.2.1.2 Definition of Distributed n-port TIOA Model

We model a specification by a finite state machine with n ports io-TA. A np-ioTA is a 7-tuple $(Q, I, O, \delta, \lambda, C, q0)$ where $n \geq 1$ and: Q is a finite set of states and

$q0 \in Q$ is the initial state; I is a n-tuple $(I_1, I_2, I_3 \ldots\ldots\ldots I_n)$, where I_i is a finite set of inputs of port i, $I_i \cap I_j = \phi$ for $i \neq j$ and $i, j = 1, \ldots\ldots\ldots n$; Let $I = I_1 U I_2 U \ldots\ldots\ldots U I_n$;

O is an n tuple $(O_1, O_2, \ldots\ldots\ldots O_n)$ where O_i is a finite set of all the outputs of i, $O_i \cap O_j = \Phi$ for $i \neq j$ and $i,j = 1, \ldots\ldots, n$. Let then $O = (O_1 U \{\varepsilon\}) X (O_2 U \{\varepsilon\}) X \ldots\ldots\ldots X (O_n U \{\varepsilon\})$, where ε stands for the empty output. δ is a transition function : $D \rightarrow Q$ and λ is an output function. $D \rightarrow O$, where $D \supseteq Q X I$. And finally, clock C is a tuple $(C_1, C_2 \ldots\ldots\ldots C_n)$

- ξ_T is a set of transitions.

A transition is therefore defined by $\xi_T = < q, r, \{?, !\}a, EC, \lambda, \mu >$ where q and r are origin and destination locations and a is an event or action (denoted by {?, ! }a), which would be input or output action. ξ_T may occur only if EC = True. The subset $\lambda \supseteq C_T$ specifies the clocks to be reset in this transition and μ is associated clock constraint. A clock constraint is a boolean combination of atomic conditions of the form $x<m$, $x \geq m$, $x \leq m$, $x = m$, $x > m$, where $x \in C_T$ and m is the integer constant.

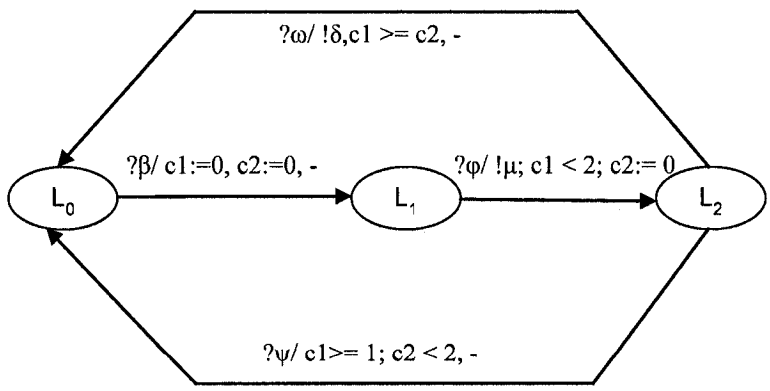

Fig: 4-5: n-port TIOA model for distributed test method

We consider a system modeled by TIOA where locations are represented by nodes, and a transition Tr = (q, r, {?, ! }a, EC, {ci, cj.....}) is represented by an arrow linking q to r .The absence of EC or of clocks to reset is indicated by "-".

The system is initially in location Lo. It reaches location L1 by the reception of β. Two clocks are set to zero. In location L1, by the reception of φ the system sends simultaneously μ while c1 < 2 is true then the transition will be occurred and the system goes to L_2.

4.2.2 Test Generation Method

Different test generation methods have been discussed in section 2.5 in chapter 2. Here, we focus only transition tour approach, which is going to apply for test case execution procedure.

The purpose of a test selection method is to come up with a set of test cases, usually called a "test suite", which has the following properties:

a) The test suite should be relatively short; that is, the number of test cases should be small, and each test case should be fast and easily executable in relation with the Implementation Under Test (*IUT*).

b) The test suite should cover, as much as possible, all faults, which any implementation may contain. In the case of a formal specification of the protocol being available, the test selection and fault analysis can be based on this specification.

A test generated by the state transition tour approach [19, 20] checks whether the implementation generates correct outputs in response to a certain input in all states. But it

does not check whether the implementation enters a correct next state after making a transition. The extreme example is the transition tour (TT) method [21], which executes all transitions of the specification at least once but does not make any effort to identify the target state.

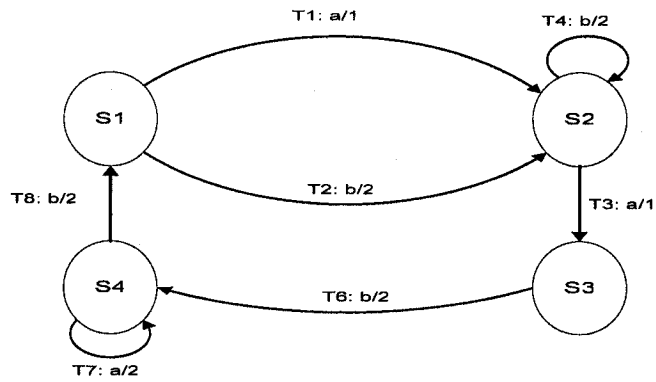

Fig 4-6: FSM example

In Fig. 4-6: the possible transition tour is formed by the transitions' sequence: $T_1.T_4.T_3.T_6.T_7.T_8.\ T_2$. No transition needs to be traversed twice. It's clear that the arrival state of the last transition, T_2, is not checked. As a result, if T_8 transition reaches to *S2* instead of *S1*, this transfer fault will not be detected. In order to systematically detect the transfer faults, one has to identify the state into, which the implementation goes after the execution of the tested transition.

4.2.3 Test Case Execution

In [18], Automated test execution is important, especially for debugging (Where the same tests are executed on different versions of implementations) and for regression testing. In most cases, abstract test cases are translated into programs, which are compiled and executed, after being linked directly or through some inter-task communication

facilities with the *IUT* (test engine for example). Various tools exist for the management of a test campaign. Such tools control the execution of many test cases in sequence, including conditional execution depending on the verdicts of previous test cases. The test results are usually automatically recorded in the form of so-called test traces. No standard exists for the description of these traces. Moreover, there are possibilities for partially automating the test result analysis and diagnostics processes, and obtaining automatically an oracle starting from a given specification. In the case of a deterministic specification, which is written in an executable specification language, an execution environment for this language provides the means for obtaining an oracle, since it is sufficient to execute the specification with the same inputs as applied to the *IUT*; the output obtained from the specification is the output to be expected from the *IUT*.

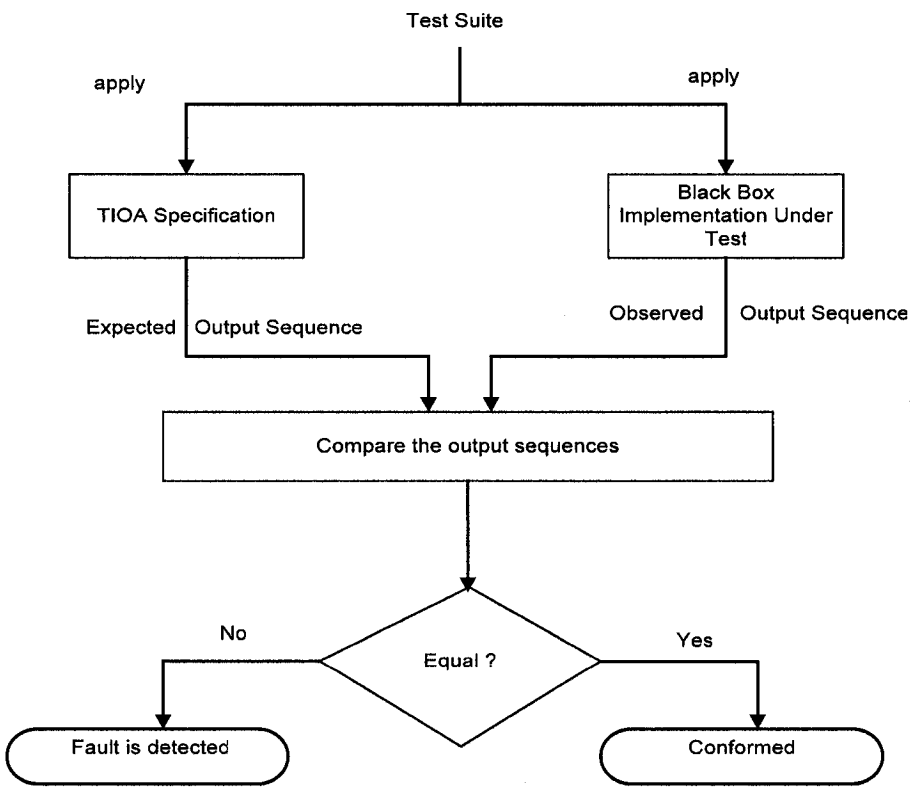

Fig: 4-7: Oracle methodology

In the case of non deterministic specifications, the execution will provide only one of all the possible outputs (or output sequence), which is allowed. Their direct comparison with the outputs obtained from the *IUT* is not suitable for determining whether this output is correct.

Simulated execution with back-tracking has been explored for the automatic generation of oracles and diagnostics from a given executable reference specification (e.g TETRA for LOTOS specifications, and TANGO for Estelle specifications, developed at university of Montreal). Unfortunately, in many cases there exists an explosion of possibilities for unsuccessful back-tracking, and these methods become very inefficient, if not infeasible. However, in other cases, useful results have been obtained.

4.3 Centralized Method for Test Case Execution

In centralized method, the test system is controlled by a single synchronizer or tester and it is composed of many small internal testers and one global clock. Here, the synchronizer sends inputs or test sequences to different ports of *IUT*. Global clock starts timer for different inputs and waiting for outputs. After receiving those inputs, *IUT* gives feedbacks or outputs to the synchronizer. The synchronizer checks the semantics of outputs and verifies the timing constraint of receiving feedbacks. Therefore, the small testers of synchronizer finally conform to the specification by checking if each input sequence corresponds to those outputs with their timing intervals.

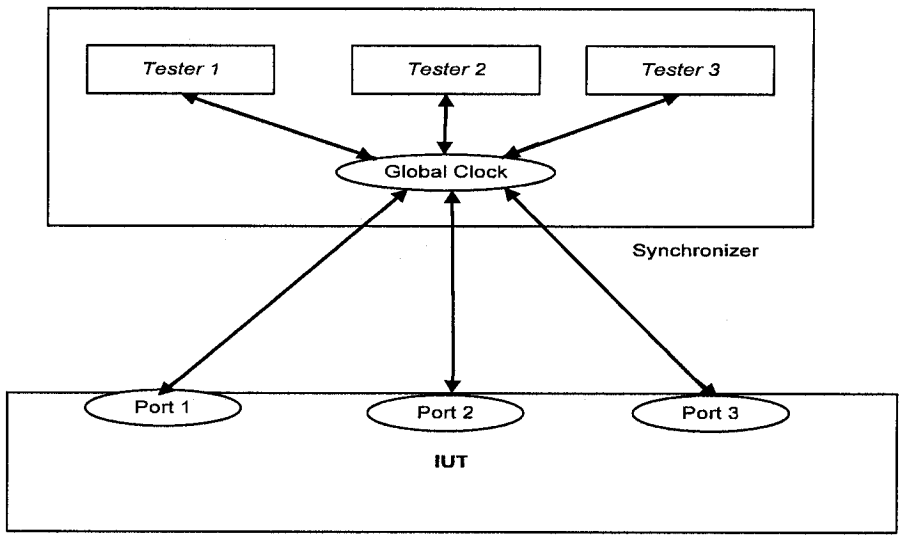

Fig. 4-8: Physical connection with Test System and IUT (Centralized)

This section describes the algorithm that will be used by the *TS* (Test System) to verify *IUT*. The function of the *Synchronizer* is to check ID (address) of local testers and port of *IUT*, where it makes the data structure to send test cases to the different port of *IUT*. In addition, it receives the exact time from global clock at the instant of forwarding the inputs to the specific port of *IUT* and also verifies the timing constraint when output of *IUT* arrives first in the *Synchronizer*. For the simplicity, we ignore the timing difference between *Synchronizer* and local small testers. Further, the *Synchronizer* reserves the route or path for communicating between testers and *IUT*. In our test procedure, we consider the *Reaction time* of *IUT*, , which is denoted RT_{IUT}, is an upper bound of time separating (i) any instant when an event *e* is received by *IUT* and (ii) the instant when *IUT* has terminated to send all the outputs (if any) in response of e. We also consider the *Transfer time* between *IUT* and *TS*, $[TT^{min}, TT^{max}]$ the time separating (i) the instant when a message *M* is sent by *IUT* and (ii) the instant when the message *M* is

received by *TS*. These two timing parameters help our test method for real judgment of receiving and sending data with their required duration.

As in parallel probing approach [38], we first define some node types, and then define some procedures that will be invoked when a node receives data or packet. The nodes of the network, with respect to call, are classified into four types:

- SOURCE: The source node of a call.
- DESTINATION: The destination node of a call.
- ID: An intermediate destination of the call. A call can have more than one ID, and the search for paths proceeds sequentially from one ID to another. The selection of IDs and the sequence in, which the IDs are to be searched for, is determined by least-cost metric between the SOURCE and DESTINATION of the call. Let ID_0, ID_1, ID_2 .....IDn be the sequence of IDs of a call. Without loss of generality, ID_0 is the SOURCE and IDn is the DESTINATION.
- IID. An intended intermediate destination of the call, which is an ID to which a route is currently being probed, ID_i is an IID if and only if ID_{i-1} was the previous IID and ID_{i+1} will be the next IID. Without loss of generality, SOURCE (ID_0) is the first IID and DESTINATION (IDn) will be the last IID.

Here, the procedures executed by nodes during call establishment are classified into two categories:

- ***Reserve (N, ID_j, Tr_P)***. Route the test sequence (Tr_P) from node N (internal tester) to its preferred neighbor for reaching node ID_j based on heuristic. This involves performing a call admission test on the best preferred link of node N, checking for the availability of bandwidth required by the call. If call admission is successful,

it returns "success", and the resources necessary for transmission are reserved; otherwise, it returns "failure". If the admission test fails on the best preferred link, the call admission test is performed on the next-best preferred link. This procedure is repeated until either the call admission is successful or a fixed number of tries has been made.

- ***Forward (ID_i, IID, Tr_i)***.This procedure initiates parallel probes on k paths using k different heuristics and sending test sequences to the destination ID.

Reaction time: *RT // As we defined above*

Transfer time: *TT // same as*

No. of test sequences: k;

No. of inputs in one test sequence: m;

Inputs:

Test sequences Tr_1, Tr_2,..... Tr_K.

Outputs:

Conformance: PASS, FAIL and INCONCLUSIVE.

Procedure:

When a test suite (new message) is created from any internal tester, Synchronizer reserves the path on the preferred link.

SOURCE: Assemble test suite with intermediate destinations (ID_0, ID_1, ...ID_{n-1}, ID_n), without loss of generality, ID_0 is SOURCE and ID_n is DESTINATION.

Begin

Let H_1, H_2.....,H_k be the k heuristics.

For P = 1 to K do

Fig. 4-9: Centralized Test System (Pseudo code)

Select the best neighbor based on heuristic H_p;.

Status [P] = Reserve (N, ID_j, Tr_P); / ID_j is the destination node port ID of IUT, N is the*/*

*Begin /*Source ID of the local tester and the test sequence is Tr_P.*/*

Repeat

If (call admission is successful) then / Request is accepted by the Synchronizer */*

Reserve bandwidth on the preferred link.

Forward (ID_i, IID, Tr_P); / Sending inputs from tester to the specific port through*

/ Synchronizer.*/*

If (Expected output occurs) then / Returning the Boolean value from function */*

Return (success). / Forward (). Now outputs of IUT */*

Receive current time from Global clock by Synchronizer.

*/*Waiting for receiving outputs*/*

While (no outputs are left)

Synchronizer receives the outputs and checks destination local tester ID and source port ID of IUT.

Synchronizer sends outputs to the respected local tester.

If (IUT does not satisfy the expected condition, EC) then

Conformance: = FAIL; BREAK

Else if ((RT + TT) <= EC) and (Tr_i > Tr_{i-1}) then

Fig. 4-10: Centralized Test System (Pseudo code) Contd.

```
Conformance: = INITPASS;  /* It means initial pass. Therefore, IUT responses against */
 End If                          /* one input of one test sequence */
Increment the number of outputs
End while
Conformance = PASS; /* It means complete' pass' of the respected test sequence. */
Return (success);
Else If (i.e. a non-expected output occurs) and (IUT satisfy the condition, EC) then
      Conformance: = INCONCLUSIVE; BREAK
 Else: Conformance: = FAIL; BREAK
End If
Else
Select the next preferred link.
Until (maximum number of neighbors have been tried)
Return (Failure);
End
```

Fig. 4-11: Centralized Test System (Pseudo code) Contd.

4.4 Distributed Method for Test Case Execution

Distributed test architecture has been studied [37] for testing distributed real time systems. In this architecture, the implementation under test (*IUT*) contains several ports, and the test system (*TS*) consists of local testers for each port of the *IUT*. Each local tester communicates with the *IUT* through its corresponding port. Messages exchanged with other testers are called coordination messages. The local testers allow exchanging

coordination messages with one another, through a communication medium, which is independent of *IUT*. In Fig. 4-12, there are two types of coordination messages- message O is used to guarantee (response check) order constraints of inputs (response outputs), and message T is used to guarantee (response check) timing constraints of inputs (response outputs). A local tester, *TesterP,* in order to obtain timing information from *ClockP,* (1) asks *ClockP* to send the current time; (2) receives the current time from *ClockP*; the received current time is used by *TesterP* to check whether the timing constraint of a just received output is respected; (3) asks *ClockP* to notify when the current time reaches a given value, say τ; and (4) is notified by *ClockP* when the current

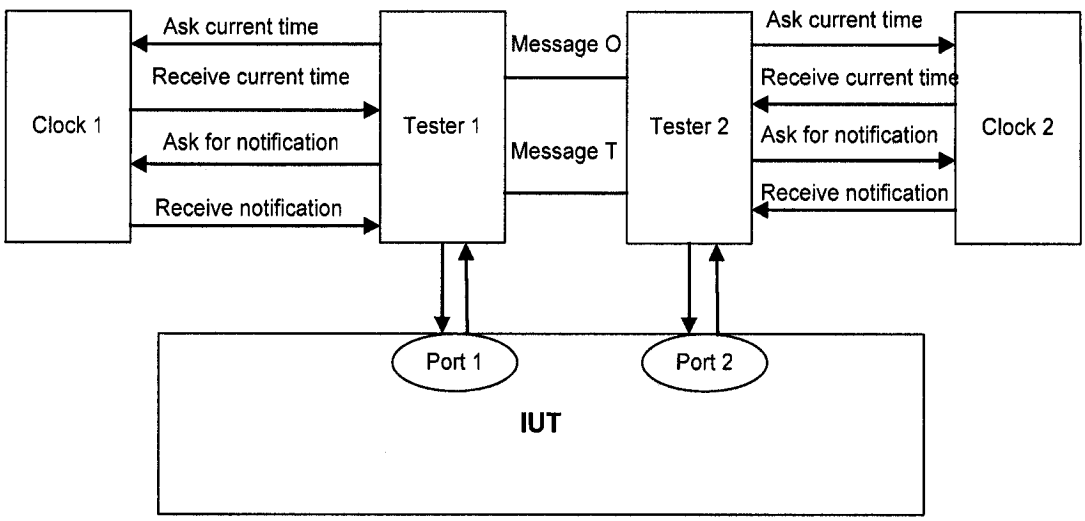

Fig. 4-12: Physical connection with Test System and *IUT* (Distributed)

time is τ; the notification is used by *TesterP* to guarantee the sending of an input at a selected instant, which respects the timing constraint of the input. The controllability and observability problems can actually be resolved if and only if certain timing constraints are satisfied by the *TS*.

***Reaction time**: RT // As we defined above*

***Transfer time**: TT // same as*

No. of test sequences: k;

No. of inputs in one test sequence: m;

N number of Testers: TS_1, TS_2……TS_n;

N number of Clocks: C_1, C_2……C_n;

Inputs:

Test sequences Tr_1, Tr_2,…… Tr_K.

Outputs:

Conformance: PASS, FAIL and INCONCLUSIVE.

Procedure:

When a test suite (new message) is created from any Tester, it reserves the path on the preferred link for connecting to the port of IUT,

SOURCE: Assemble test suite with intermediate destinations (ID_0, ID_1, ……ID_{n-1}, ID_n), without loss of generality, ID_0 is the SOURCE and ID_n is the DESTINATION.

Begin

Let H_1, H_2……,H_k be the k heuristics.

For P = 1 to K do

Select the best neighbor based on heuristic H_p.

Status [P] = Reserve (N, ID_j, Tr_P); /ID_j is the destination node port ID of IUT, N is the*/*

*Begin /*Source ID of the local tester and the test sequence is Tr_P.*/*

Repeat

If (call admission is successful) then / Request is accepted by other testers*/*

Fig. 4-13: Distributed Test System (Pseudo code)

```
Reserve bandwidth on the preferred link.
Forward (ID_i, IID, TrP); /* Sending inputs from tester to the specific port*/
Forward (ID_i, IID, O_i); /* Sending Coordination message (O) to the other testers */
Forward (ID_i, IID, T_i); /* Sending Coordination message (T) to the other testers */
Receive current time from local clock C_i;
If (Expected output occurs) then /* Returning the Boolean value from function*/
Return (success);.                    /*Forward (). Now outputs of IUT */
// Waiting for receiving outputs
While (no outputs are left)
Tester receives the outputs and checks the destination tester ID and source port ID of IUT.
If (IUT does not satisfy the expected condition, EC) then
        Conformance: = FAIL; BREAK
Else if ((RT + TT) <= EC) and (Tr_i > Tri-1) then
    Conformance: = INITPASS; /*It means initial pass. Therefore, IUT responses*/
 End If                              /* against one input of one test sequence.*/
Increment the number of outputs
Endwhile
Conformance = PASS; /* It means complete pass of the respected test sequence.*/
Return (success);
Else If (i.e. a non-expected output occurs) and (IUT satisfy the condition, EC) then
    Conformance: = INCONCLUSIVE; BREAK
 Else Conformance: = FAIL; BREAK
End If
Else
```

Fig. 4-14: Distributed Test System (Pseudo code) Contd.

Select the next preferred link.

Until (maximum number of neighbors have been tried)

Return (Failure);

End

Fig. 4-15: Distributed Test System (Pseudo code) Contd.

4.5 Summary

We provided two methods for timed test case execution. These methods are based on specification expressed in np-TIOA. We have distinguished centralized and distributed test architecture and their procedures. Coordination messages are exchanged among testers in distributed test system. Synchronizer virtually acts as a single tester and controls input-output with timing constraint, which has been applied on centralized test system only. We have also given the definition of n port TIOA for these two methods. In addition, step by step testing process has been discussed in this chapter.

In the next chapter, we will provide the detailed implementation of the algorithms of these two methods.

Chapter 5

Implementation of the Methodology

We have already introduced our two methods including architectures and algorithms for test case execution based on n-port TIOA in chapter 4. Now, we implement those architectures by CORBA- a general model for distributed programming and object oriented programming language with Java. In this chapter, the design and specification of centralized and distributed test architectures are presented. A real life automata model is given to illustrate how our methods work. Section 5.1 introduces the overview of implementation. Section 5.2 describes the implementation of centralized test architecture. Section 5.3 represents the implementation of distributed test architecture and sections 5.4 and 5.5 explain a case study, results and results analysis. Section 5.6 presents the summary.

5.1 Overview of the Implementation

Our main focus is to implement the test system environment and to apply test cases on *IUT*- a distributed real-time system, we consider Train-Gate-Controller – a railroad crossing system act as an *IUT*, which is also a reactive system. By using two methods, we implement centralized and distributed test architecture by CORBA distributed model. It communicates by sending and receiving inputs and outputs from *IUT* i.e. Train-Gate-Controller system. According to our methods and by sending test cases to *IUT*, the Test system checks every output coming from *IUT* on different ports

corresponding to the inputs and their timing constraints. If the verdict passes according to our FSM based specification, which is expressed in n-port TIOA, then system is successfully called "PASSED"; otherwise it is called "FAILED". For testing of sequential software, it is usually sufficient to provide the same input (and program state) in order to reproduce the output. The behavior of distributed real time system, on the other hand, not only depends on inputs but also on the order and timing of the concurrent tasks that execute and communicate with each other.

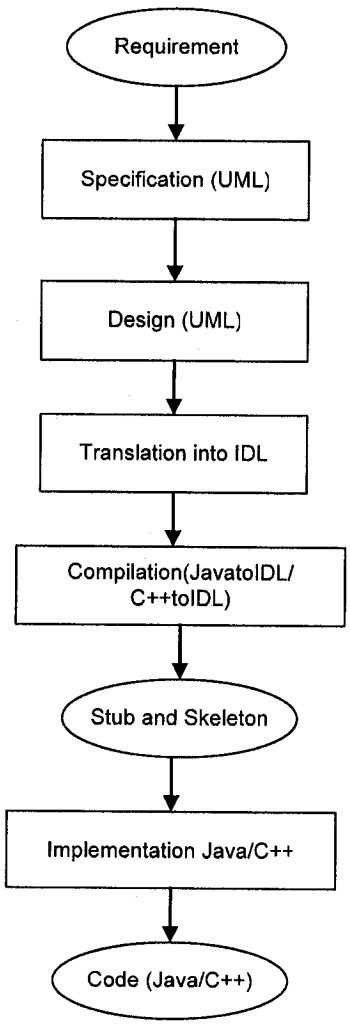

Fig. 5-1: A general approach for designing the CORBA system

Fig. 5-1 depicts an overview of CORBA implementation process. The whole process has been implemented in our centralized and distributed test system. We analyze the necessary requirements for CORBA on testing environment. The essential aspects such as observability are important for determining if *IUT* performs correctly. In section 5.4, we show that Train-Gate-Controller creates three individual automata models and discuss about test cases or stimulus generated by Transition Tour method from combined automata model that ensures correct ordering and timing constraint of each test case. Also, our Test System observes the actions or outputs created by *IUT* in response of stimulus without hampering timing behavior. During testing, we must be able to control the execution of the system, so that it is possible to reproduce arbitrary test scenarios such that the system's reaction is deterministic. Fundamental to our methodology is the notion of a real time transaction, which is used to express the concerns about functionality and timeliness of a sequence of computational and communication steps in a single concept. Here, Real time transactions are used to describe the runtime behavior of Train-Gate-Controller system.

In order to show the detailed design and implementation, we use class diagrams, sequence chart diagrams and collaboration diagrams based on Rational rose software.

5.2 Implementation of Centralized Test Architecture

To capture and display the behavior of component–based system, we build a test system, which is named here Synchronizer. It coordinates the global clock and three internal testers and also is responsible for injecting required test cases into Train-Gate-Controller and then display the behavior with respect to the application architecture. The

design of a CORBA system fits in regular software development process. The IDL code generated from the logical view serves as baseline for implementing CORBA mechanism.

We now describe the whole process of implementation (Fig. 5-1) for centralized test system in the following subsections.

5.2.1 Specification and Design

Fig 5-2 is the physical layout of client server programming through CORBA architecture. We represent the client as a Test system and the server as an *IUT*. Object Request Broker (ORB) is the communication channel that guaranties the transparency of resource location and communication between client and server components. An IDL (Interface Definition Language) describes the interface of software and not a complete implementation. The IDL definitions are stored in an interface repository. This repository is mainly used to dynamically construct a method invocation (dynamic invocation interface). Through interfaces, client side is completely independent of where the object is located, what programming language it is implemented in, or any other aspect, which is not reflected in the object's interface.

The following subsections explain the main component of Centralized Test System, which is shown in Fig.5-2 and has been implemented by CORBA distributed model.

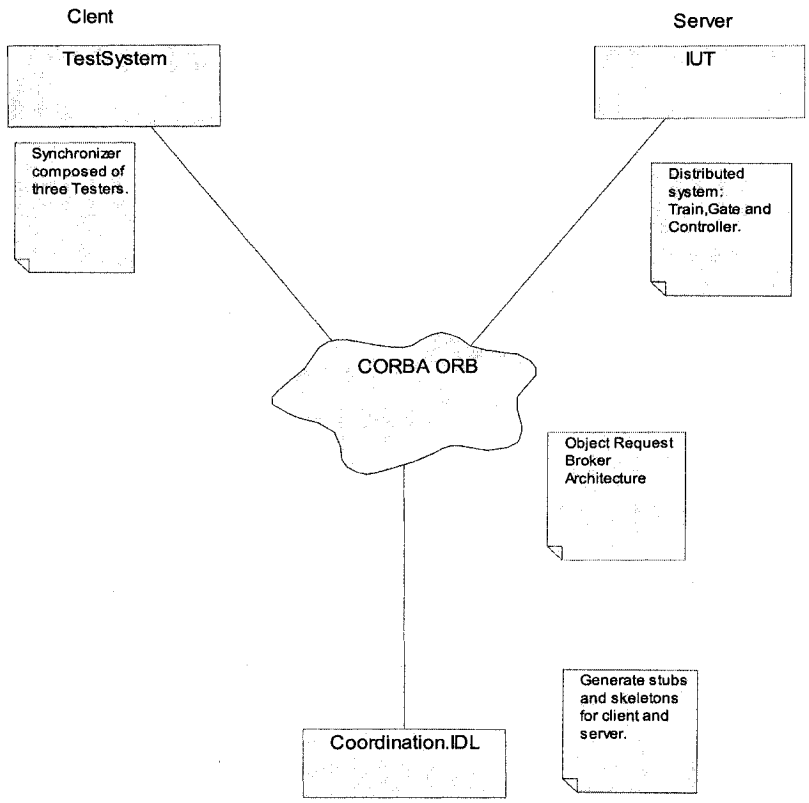

Fig. 5-2: Physical architecture of Centralized Test System

5.2.1.1 CORBA ORB

The *ORB* is the object bus. It lets objects transparently make request to-and receive responses from-other objects located locally or remotely. The client is not aware of the mechanisms used to communicate with, activate, or store the server objects. A CORBA ORB provides a wide variety of distributed middleware services. The ORB lets objects discover each other at run time and invoke each other's service. An ORB is much more sophisticated than alternative forms of client/server middleware including

traditional Remote Procedure Calls (RPCs), Message Oriented Middleware (MOM), database stored procedures, and peer to peer services. A CORBA ORB lets us either statically define our method invocations at compile time, or it lets us dynamically discover them at run time. It invokes methods on server objects using high level language of choice irrespective of the language server objects are written in. Every CORBA ORB must support an *Interface Repository* that contains real-time information describing the functions and their parameters. The ORB includes context information in its messages to handle security and transactions across machine and ORB boundaries.

5.2.1.2 Test System

Our centralized test system is originated from synchronizer, which aggregates different internal testers and coordinates among them by global clock. The synchronizer is the heart of the Test system. It controls the sending of test suite to the *IUT*, synchronizing of different testers, receiving output from *IUT*, sending those outputs to the assigned testers and finally concluding with a verdict: either "Pass", "Fail" or "Inconclusive".

Fig. 5-3 shows the class diagram of Test system where Synchronizer aggregates Tester classes and associates with AnnoyingBeep Timer class. This Timer class also associates with RemindTask class, which specially validates the timing constraint of incoming output. Automatically generated classes, which are produced by IDL compiler, depend upon Coordination interface classes.

In the Test System, the synchronizer acts as a client object of the CORBA architecture. Firstly, it initializes the ORB to communicate with server and binds to the

client stub (CoordinationHelper), which is generated by IDL compiler. Secondly, it sends test suite to the specific port of the server and in the meantime it starts the timer. Finally, it waits for the *IUT*'s response.

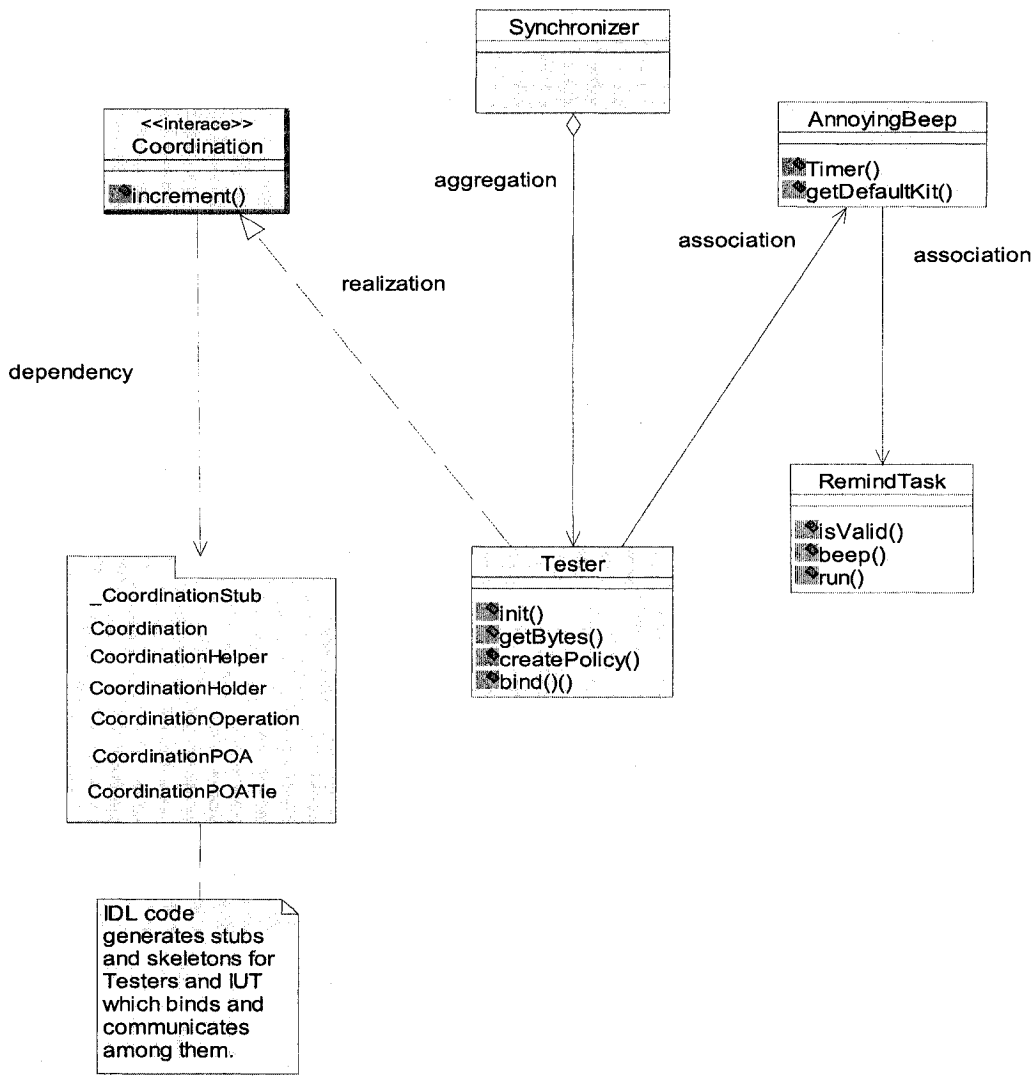

Fig. 5-3: Class diagram for Test System architecture

There are some basic CORBA syntax and their operations, which are given below:

By invoking *init* method Synchronizer initializes CORBA ORB.

org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, null);

In order to identify the Id object, TesterA invokes *getBytes* method to retrieve correct Id. This Id is also used in server side.

Byte [] managerId = "TestManager".getBytes ();

The Client obtains a reference to this policy object by invoking the ORB object's *create_policy* method and then narrowing the reference of PolicyManager by invoking *resolve_initial_reference* method.

Org.omg.CORBA.Policy ctoPolicy = orb.create_policy (RELATIVE_CONN_TIME OUT_POLICY_TYPE.value, ctopolicyValue);

PolicyManager orbManager = PolicyManagerHelper.narrow (orb.resolve_initial_ References ("ORBPolicyManager"));

By invoking *bind* method, the client stub CoordinationB.SynchronizerHelper object binds ORB to the actual client object by passing Id.

CoordinationB.SynchronizerB op = CoordinationB.SynchronizerBHelper.bind (orb, ("/TestAdmin", managerId);

Tester class associates the Timer class AnnoyingBeep and create a new timer thread to start counting and validate time and data by another class RemindTask.

As shown in Fig. 5-4, the synchronizer controls and communicates every object through different messages in centralized Test system. The main objective of this system is to maintain and perform testing process by single tester, which is known as

Synchronizer. Here, we show one transition in collaboration diagram (Fig. 5-4) and detailed scenarios are depicted in sequence diagram in Fig. 5-5.

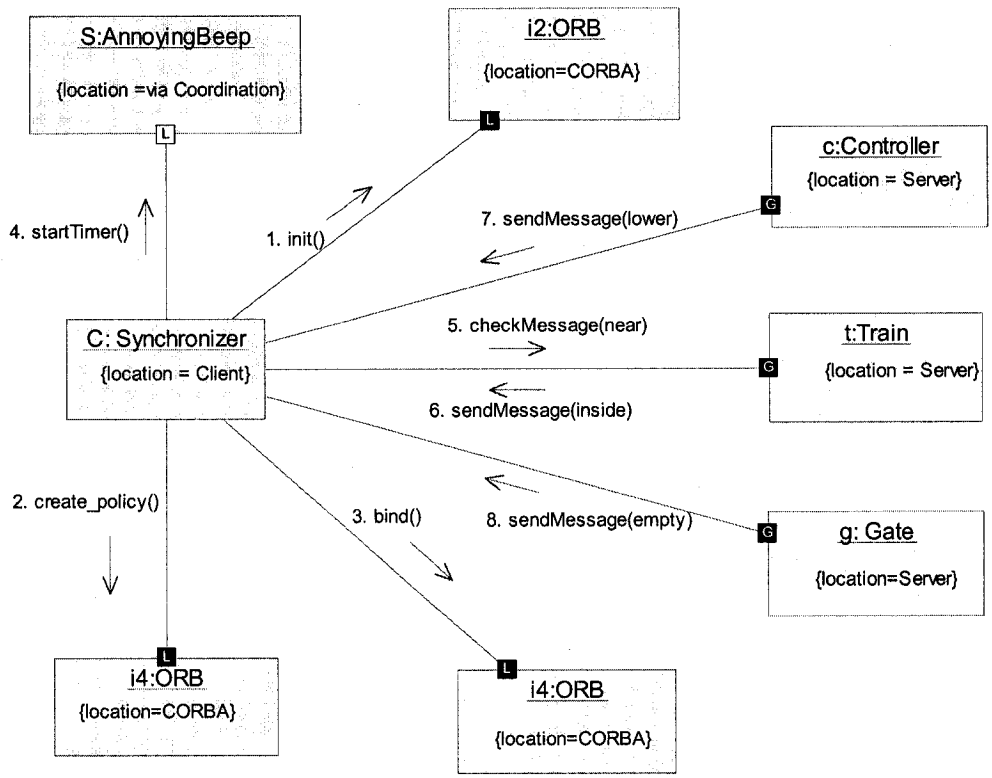

Fig. 5-4: Collaboration diagram of Synchronizer

As shown in Fig. 5-5, the synchronizer invokes *init()* method for initializing the CORBA ORB, then binds to the message object by invoking *bind()* method. Synchronizer now sends the message "near", which is our first test suite for Train component, which is the first port of *IUT*. Timer class AnnoyingBeep is instantiated and starts counting time. After receiving the test suite "near", Train takes few moments, which is called reaction time and then it is supposed to send "inside" message within 30 seconds to the Synchronizer. Besides these, Gate should send no message i.e. empty message and Controller should send "lower" message to the Synchronizer. And every

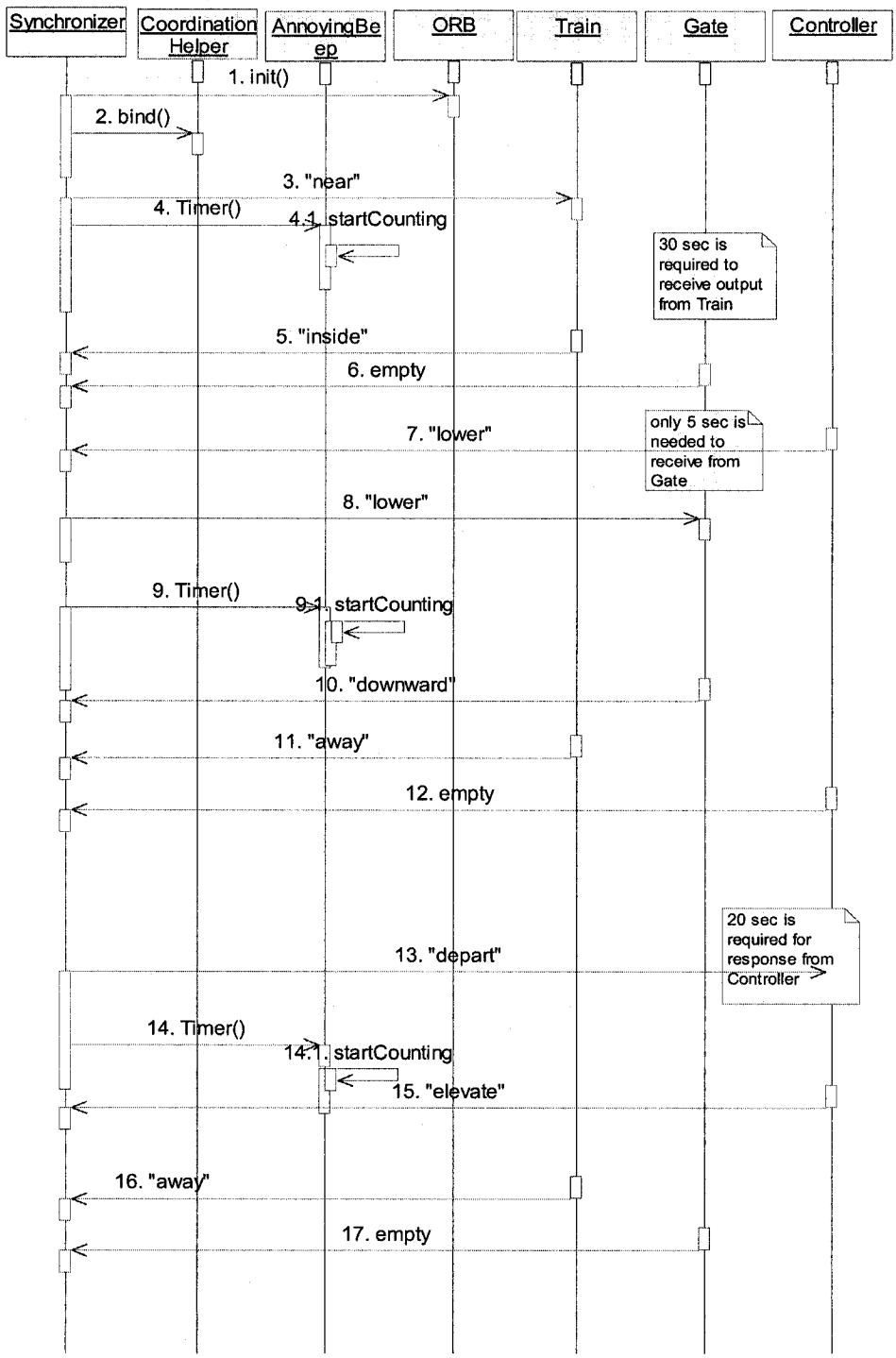

Fig. 5-5: Sequence diagram of Centralized Test System

time Synchronizer will check the timing constraint and returning output from Train-Gate-Controller. This way the first transition is completed.

Now, Synchronizer sends the stimulus to the Gate component. And it will send "lower" signal to the Gate. Then Synchronizer will be waiting for responses, which will take few moments for transfer time of their stimulus to reach *IUT* and reaction time of their outputs generated by *IUT*. After 5 seconds, the first returning message will come from Gate, which is "downward" message to the Synchronizer. Simultaneously, Train will send "away" message and Controller will send no message or empty message to the Synchronizer.

In the same way, Synchronizer will send the input "depart" message to the Controller. 20 seconds later, Synchronizer will receive "elevate" message from Controller, and "away" message from Train and it will get no message or empty message from Gate.

5.2.1.3 CORBA.IDL

The first step in the CORBA development process is to create language independent IDL definitions of our server's interface. We have two IDL files, which are CoordinationA and CoordinationB for TRAIN-GATE-CONTROLLER interface. The CORBA IDL syntax should look very familiar to a C++ or Java programmer. CORBA is a strongly typed system. All variables, parameters and return values, and attributes are typed.

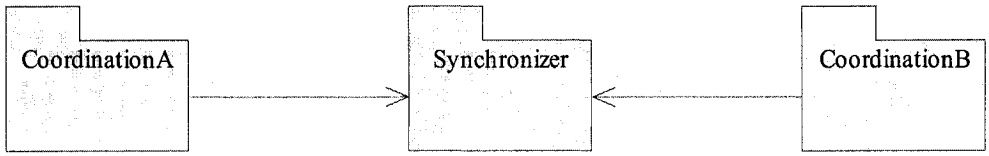

Fig. 5-6: IDL Package diagram

We must map these IDL files into something that Java clients and servers can understand. So we will need an IDL-to-Java compiler. Here, we use Borland's Visibroker compiler to compile these IDL files and to generate many Java classes and interfaces. We describe some operations of those generated files.

_CoordinationStub is a java class that implements the client-side stub for the Synchronizer object. It's really an internal implementation of the Coordination interface that provides marshaling functions.

CoordinationHelper is a java class that provides useful helper functions for Synchronizer clients. For example, the compiler automatically generates code for a narrow function that lets client cast CORBA object reference to Synchronizer type. In addition to the required CORBA helper functions, the VisiBroker implementation provides a very useful, but non–standard, function called *bind*; it is used by clients to locate objects of this type.

CoordinationHolder is a java class that holds a public instance member of type synchronizer. It is used by clients and servers to pass objects of type synchronizer as *out* and *inout* parameters inside method invocations.

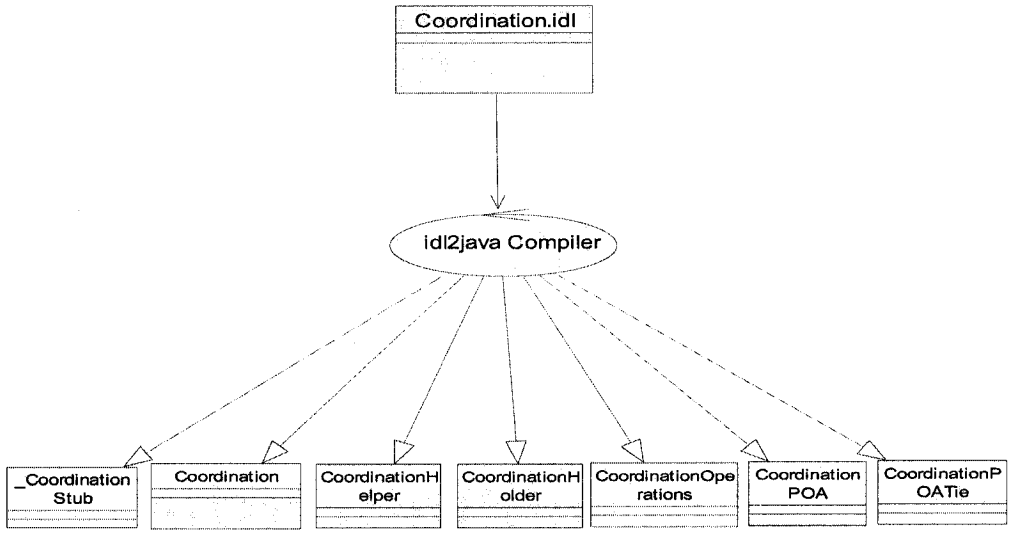

Fig. 5-7: The Java classes and Interfaces generated by idl2java compiler

Besides other generated files, Exception-handling is a generic part of client/server programming with CORBA. It all starts with the idl2java compiler, which automatically generates the CORBA *system exceptions* where a method can be generated- these exceptions are typically generated by the ORB. The idl2java will generate the appropriate Java exception classes for both types of error. All exceptions have a name and an interface repository ID. To take advantage of CORBA/Java error-handling, we must surround our code with standard Java try blocks.

5.2.1.4 Train-Gate-Controller (*IUT*)

There are three types of interacting entities: *Train, Gate and Controller*. The behavior of the systems components are modeled using reactive classes as shown in Fig. 5-8.

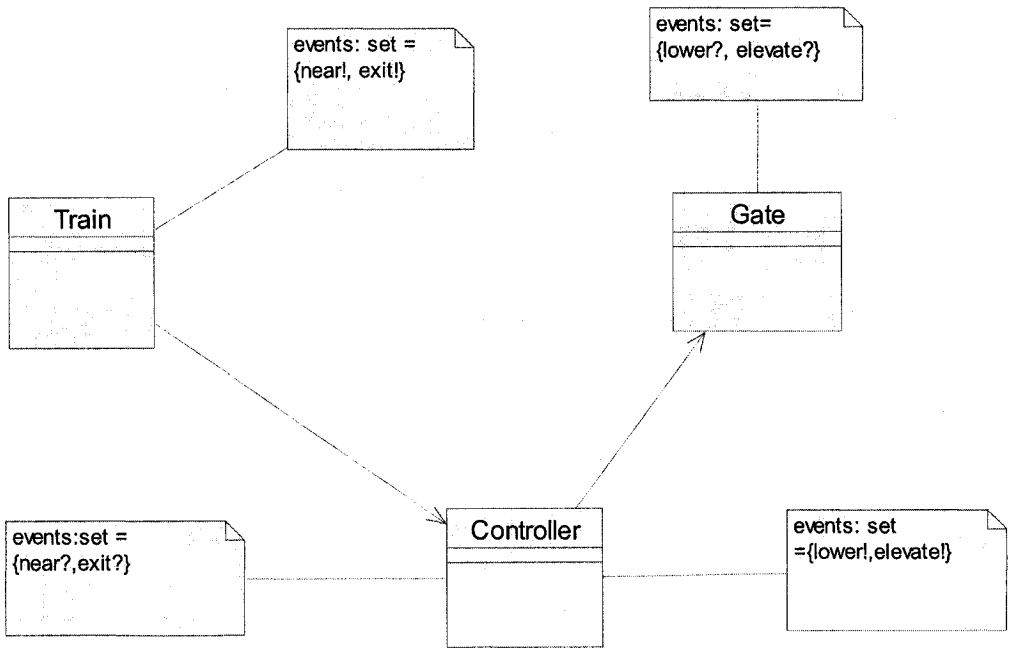

Fig. 5-8: Class diagram for Train-Gate-Controller

Our *IUT* is a reactive distributed system. 'Train' realizes the 'Controller' and sends some messages to it at certain interval for rail road crossing. 'Controller' also realizes the 'Gate' and conveys some messages to it for some events. These events must be maintained in real-time in order to avoid fatal consequences.

Fig. 5-9 shows, 'stillT' state represents the idle state of 'Train' component. The 'Train' communicates with 'Controller' using two events- "near" and "depart". The 'Train' is required to send a signal "near" to the controller within 2 to 4 time units before it enters the crossing. Moreover, the 'Train' must send another signal "depart" to the 'Controller' whenever it leaves the crossing and it requires not more than 6 time units from the initial state.

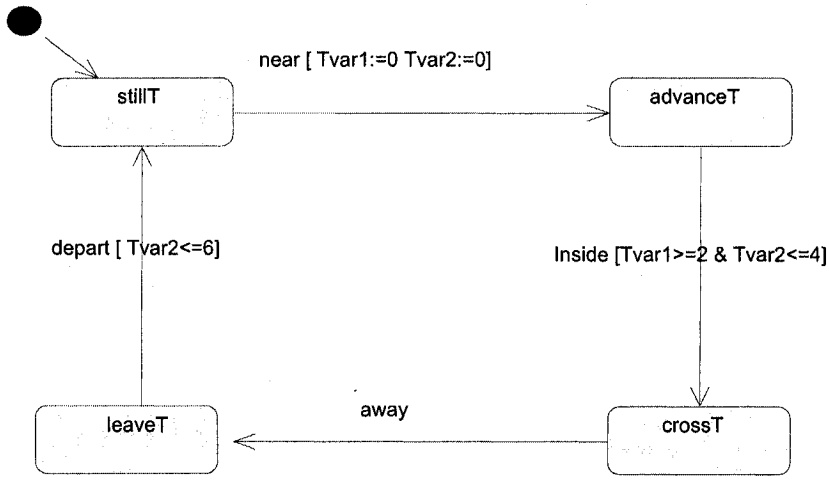

Fig. 5-9: State chart diagram for Train

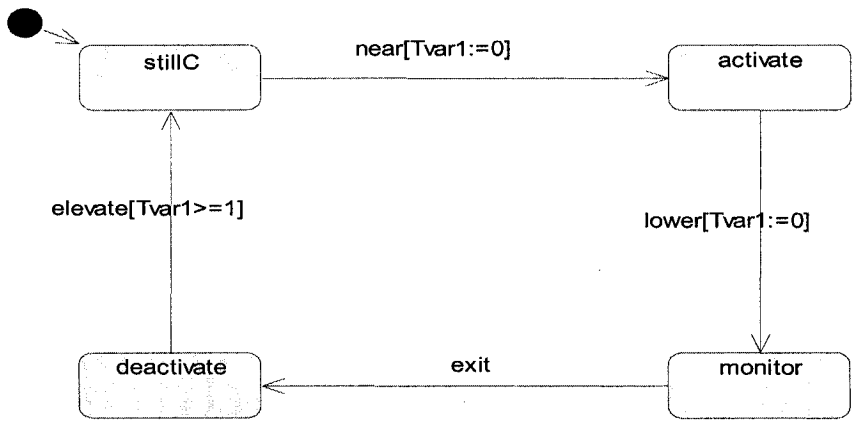

Fig. 5-10: State chart Diagram for Controller

Fig. 5-10 shows state chart diagram for 'Controller'. The idle state of 'Controller' is 'stillC' and waiting for Train's signal. Whenever the 'Controller' receives "near" signal from 'Train', it responds by sending the signal "lower" to the 'Gate'. The respond

time is 1 time unit. Whenever it receive "exit" signal from 'Train', it responds to the 'Gate' by "elevate" signal within 1 time unit.

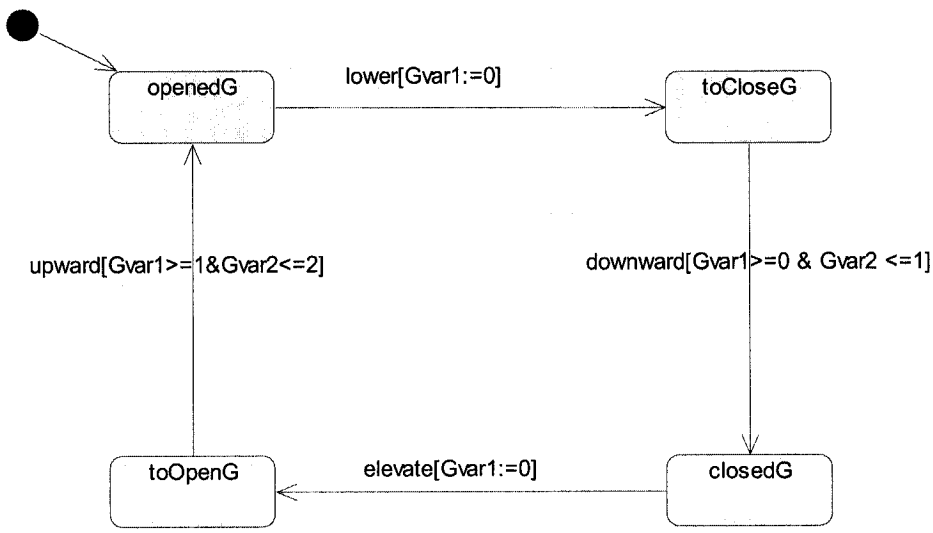

Fig. 5-11: State chart diagram for Gate

The 'Gate' is open at 'openedG' state. It communicates with 'Controller' through the signals "lower" and "elevate". Within 1 time unit it closes the Gate by "downward" event after receiving "lower" signal from 'Controller'. Again it goes to 'openG' state by "upward" event whenever it receives "elevate" signal originated from 'Controller' within 1 time unit.

The internal sequence diagram shows (Fig. 5-12) the following scenarios: At first, The Train object initializes the ORB by invoking init() method. Then it binds CoordinationHelper object which is automatically generated by IDL compiler. Now, ORB invokes create_lifespan_policy and passes PERSISTENT value as parameter. By invoking narrow() method, Train object gets a reference to the root POA.

Now, it creates policy for persistent POA and also creates the servant object. Further, it decides on the ID for the servant and activates the POA manager. Finally, the servant is waiting for incoming request by invoking run() method.

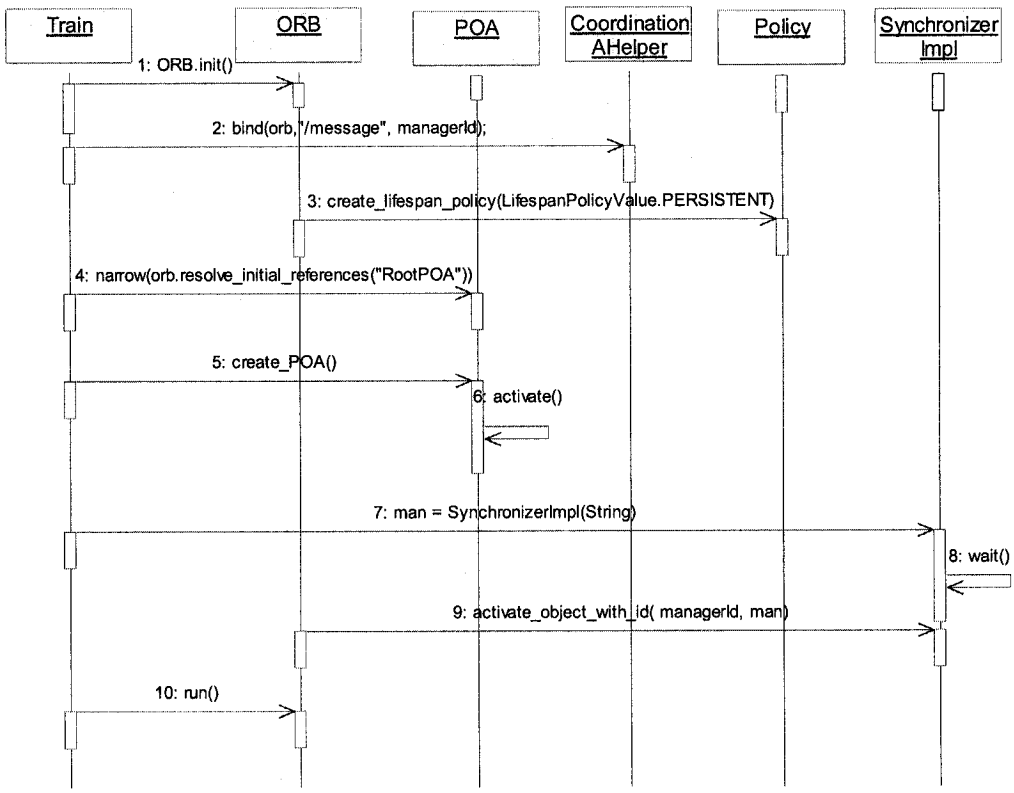

Fig. 5-12: Internal sequence diagram for Train

The other two components of IUT (Gate and Controller) behaves same action with ORB. This is the way to establish the communication channel between IUT and Test System.

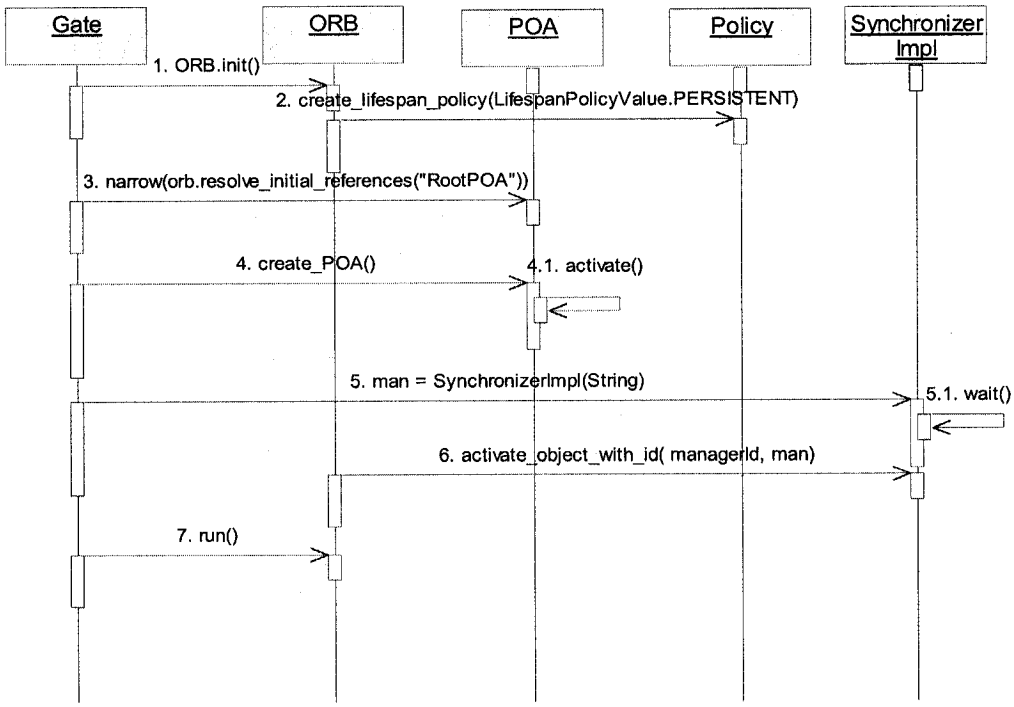

Fig. 5-13: Internal sequence diagram for Gate

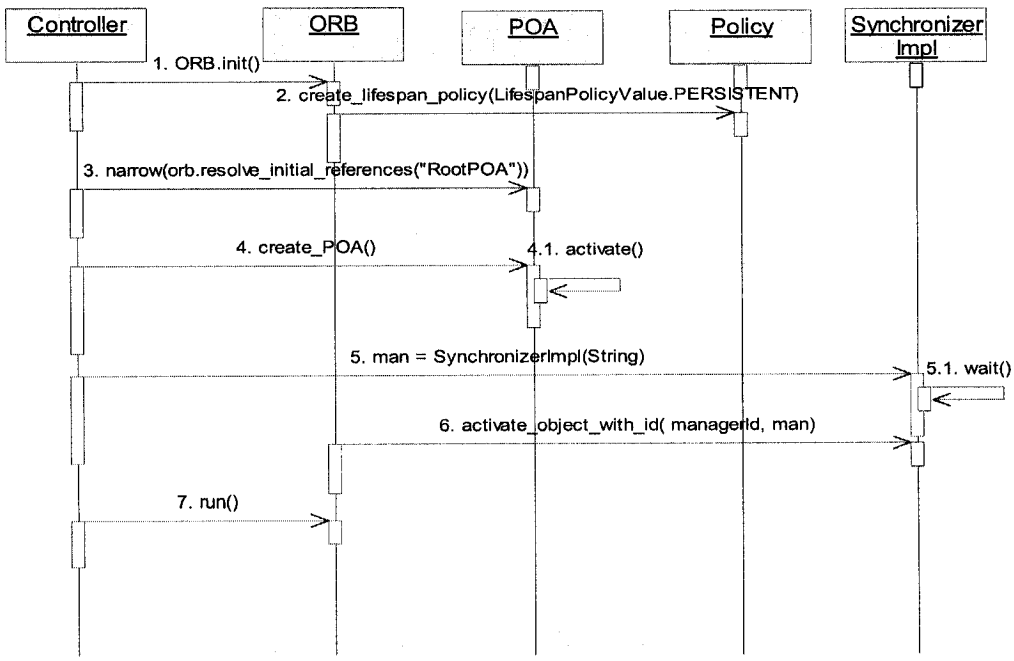

Fig. 5-14: Internal sequence diagram for Controller

5.3 Implementation of the Distributed Test Architecture

Three testers are created individually in order to develop the distributed test architecture. Besides those testers a Timer class has important role to start and stop timing whenever test cases are sent to *IUT* and returned output to the test system. Each tester sends coordination message to leave current status of particular tester and to send 'ready' signal to the other testers. We assume *IUT* has three ports to access test cases such as Train, Gate and Controller. Four IDL files are generated to create client stubs and server skeletons and to communicate among testers and *IUT* through ORB.

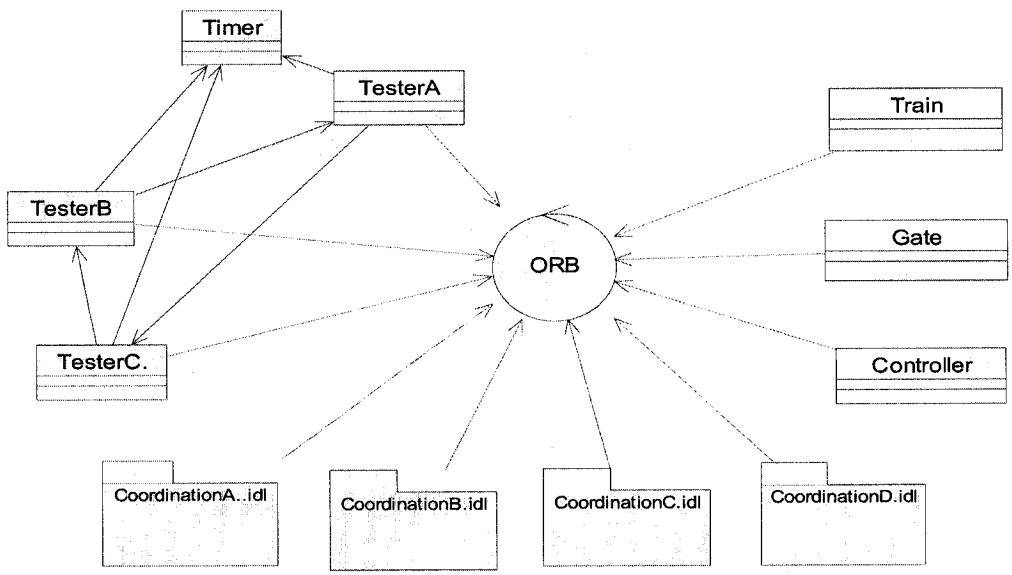

Fig. 5-15: Physical architecture of Distributed Test System

Fig. 5-15 shows the main components of the distributed test system where the client part represents TesterA, TesterB and TesterC communicate each other through ORB and Timer class also associates with test system. On the other hand, server part depicts three main components Train, Gate and Controller, which is our *IUT*. There are

four packages like CoordinationA, CoordinationB, CoordinationC and CoordinationD, which specify the interfaces and generated client stubs and server skeleton programs. These classes are also realized to the ORB and make smooth and reliable communication between client and server.

Fig.5-16 shows the sequence diagram, which basically represents the operations or interactions between multiple clients and server components through CORBA. By this interaction, we will able to see the scenarios of transition in TIOA.

At first, we determine the client objects, which constitute our test system. *TS* consist of three testers like TesterA, TesterB and TesterC. Server is composed of three components like Train, Gate and Controller. CORBA provides the communication channel, which is object request broker (ORB) and CORBA IDL provides the interface between client and server. We implement this interface by Java.

Now, TesterA invokes *init()* method for initialization the CORBA ORB, then binds to the message object by invoking *bind()* method. TesterA now sends the message "near", which is our first test suite for Train component, the first port of *IUT*. Timer class AnnoyingBeep has been instantiated and starts counting time. Besides counting, TesterA sends the coordination message "ready" to the TesterB and TesterC. It means TesterB and TesterC should have to wait for the response of *IUT* and also those testers start their timer to check the time constraint whenever they receive the output.

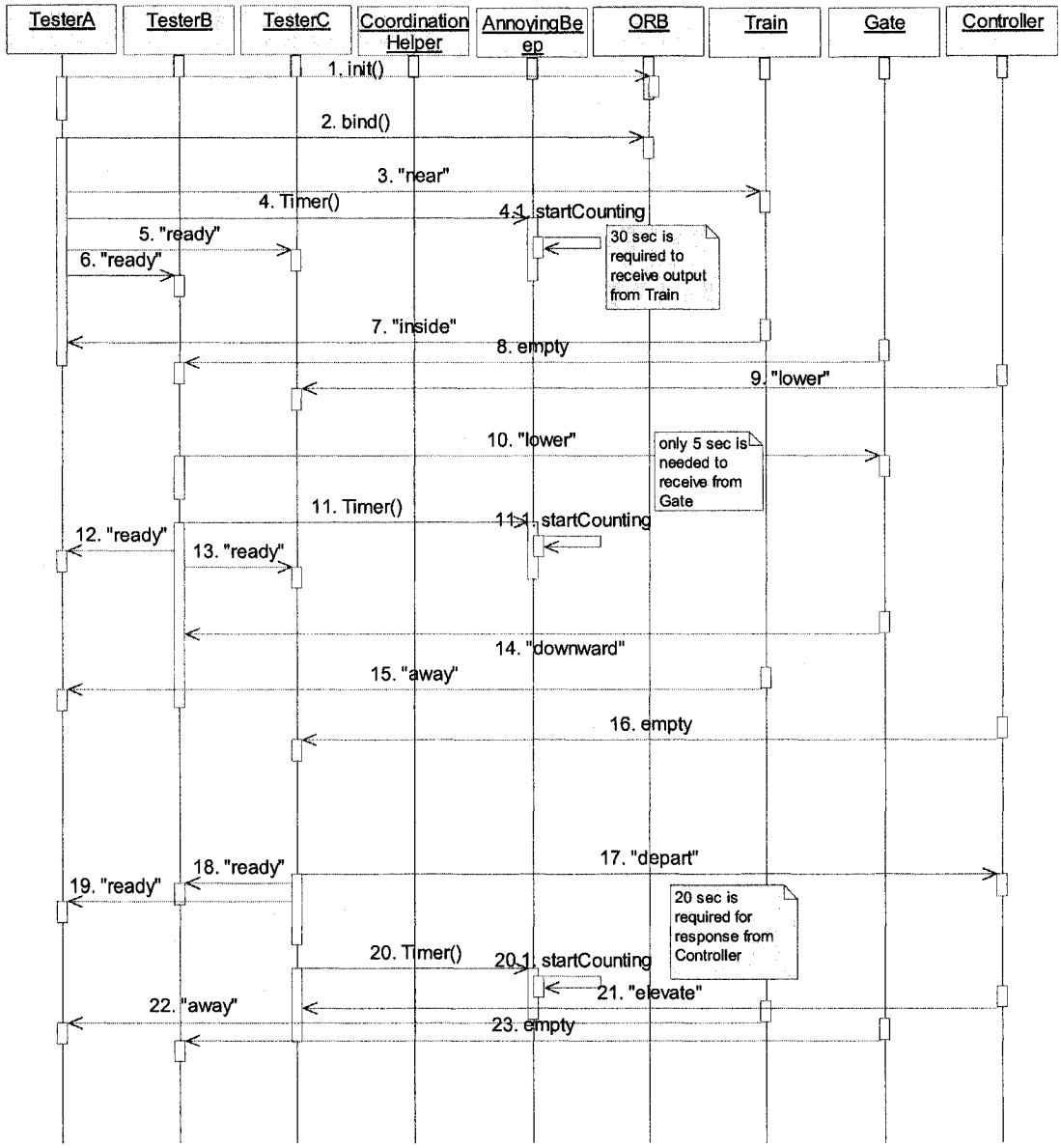

Fig. 5-16: Sequence diagram for distributed Test System

After receiving the test suite "near", Train will take few moments, which is called reaction time and then it is supposed to send "inside" message within 30 secs to the TesterA.

Besides these, Gate should send no message i.e. empty message to the TesterB and Controller send "lower" message to the TesterC. Each time testers will check the timing constraint and returning output from Train-Gate-Controller. This way the first transition is completed.

Now, TesterB sends the stimulus to the Gate component. TesterB will send "lower" signal to the Gate and also send "ready" signal, which is coordination message to the TesterA and TesterC as they should get ready for receiving the output. Then all testers will be waiting for responses, which will take few moments for transfer time of their stimulus to reach *IUT* and reaction time of their outputs generated by *IUT*. After 5 secs, the first returning message will come from Gate, which is "downward" message to the TesterB. Simultaneously, Train will send "away" message to the TesterA and Controller will send no message or empty message to the TesterC.

In the same way, TesterC will send the input "depart" message to the Controller and also send "ready" signal to other testers. 20 secs later, TesterC will receive "elevate" message from Controller, TesterA will receive "away" message from Train and TesterB will get no message or empty message.

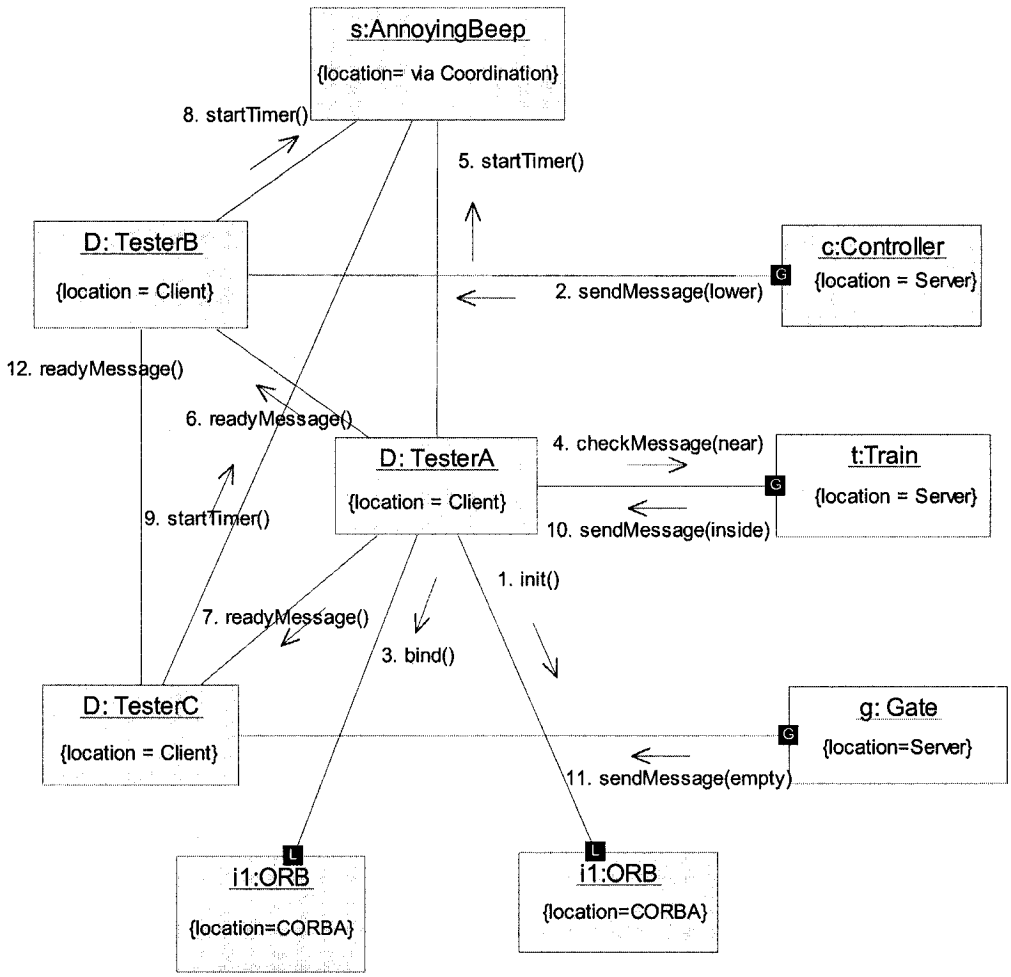

Fig. 5-17: Collaboration diagram for distributed Test System

As in Fig. 5-17, within the distributed Test system, TesterA, TesterB and TesterC, communicate each other through coordination message readyMessage() in order to inform order and timing constraint of inputs and outputs. All testers connect to the AnnoyingBeep Timer class to validate the timing constraint. At the initial stage, TesterA initializes ORB by *init()* and binding the ORB by *bind()*.

5.4 Case Study and Results

In this section, we will provide Train-Gate-Controller automata model on, which our two test methods are applied. We also look at timed test suite generation through transition tour method from the specification of the combined automata model and then through application of our test case execution algorithms on that specification for conformance testing.

5.4.1 The Real Life Automaton Model

This example involves a Train-Gate-Controller distributed reactive system. The scenario of this distributed system is as follows: The Train normally sends some messages to the Controller component at certain time interval with two events "near" and "depart". The events "inside" and "away" mark the entry and exit of the Train from the railroad crossing respectively. Whenever Controller receives the signal "near" from the Train, it responds by sending the signal "lower" to the Gate. Again, whenever the Controller receives signal "depart" from Train, it responds with a signal "elevate" to the Gate within certain period. Gate communicates with the Controller through the signals "lower" and "elevate". The events "upward" and "downward" denote the opening and closing of the Gate. The Gate responds to the signal "lower" by closing within certain interval, and responds to the signal "elevate" by opening within certain period.

Now, Fig. 5-18 shows the automaton modeling of TRAIN component. There are five events, which might be a set of events {near, depart, inside, away, restT}.The Train starts in state S0. The event "restT" represents its resting event. The Train communicates

with the Controller with two events "near" and "depart" as mentioned before. The Train is required to send the signal "near" at least 2 time units before it enters the crossing. Thus the minimum delay between "near" and "inside" is 2 time units. Furthermore, we know that the maximum delay between the signals "near" and "depart" is 5 time units. This is a liveness requirement on the Train. Both the timing requirements are expressed using a single clock x.

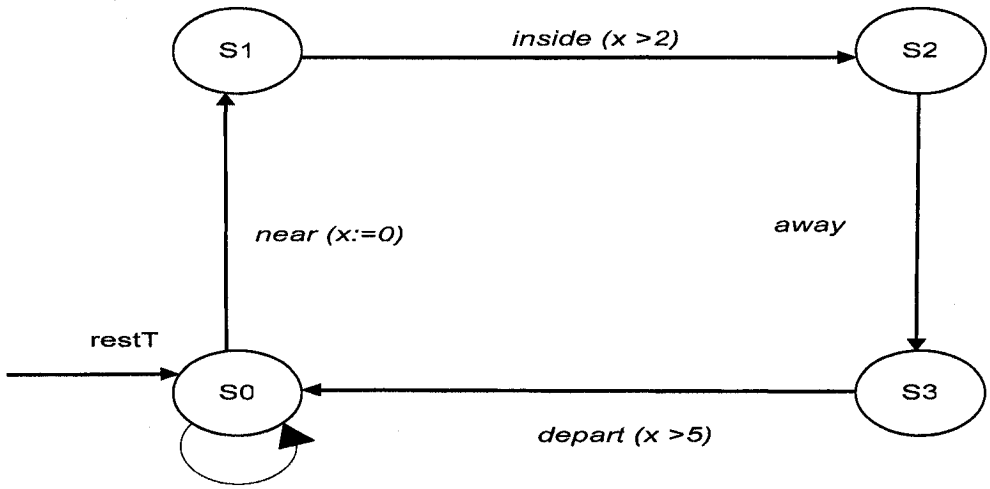

Fig. 5-18: Train

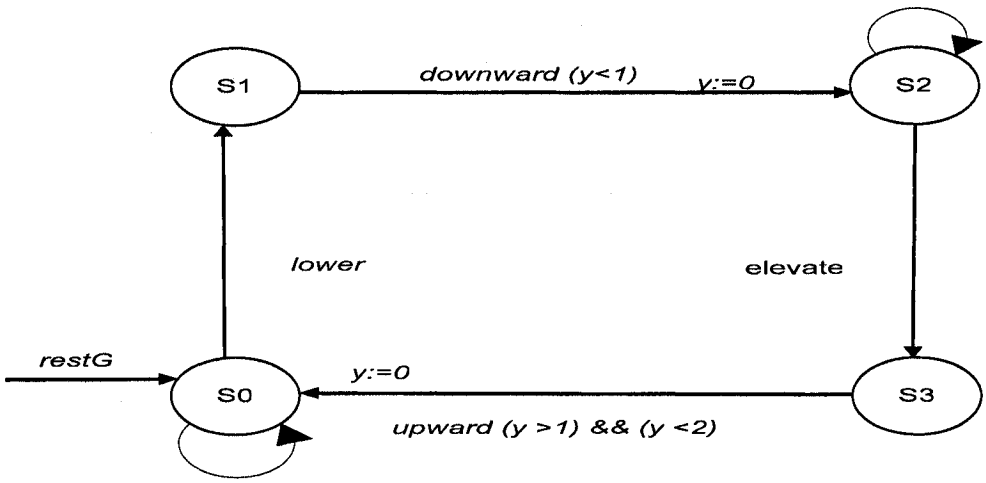

Fig 5-19: Gate

The Gate is opened in state S0 and closed in state S2. It communicates with the Controller through the signals "lower" and "elevate". The events "upward" and "downward" denote the opening and closing of the Gate. The Gate responds to the signal "lower" by closing within 1 time unit, and responds to the signal "elevate" within 1 to 2 time units. The Gate can take its resting position "restG" in states S_0 or S_2 forever.

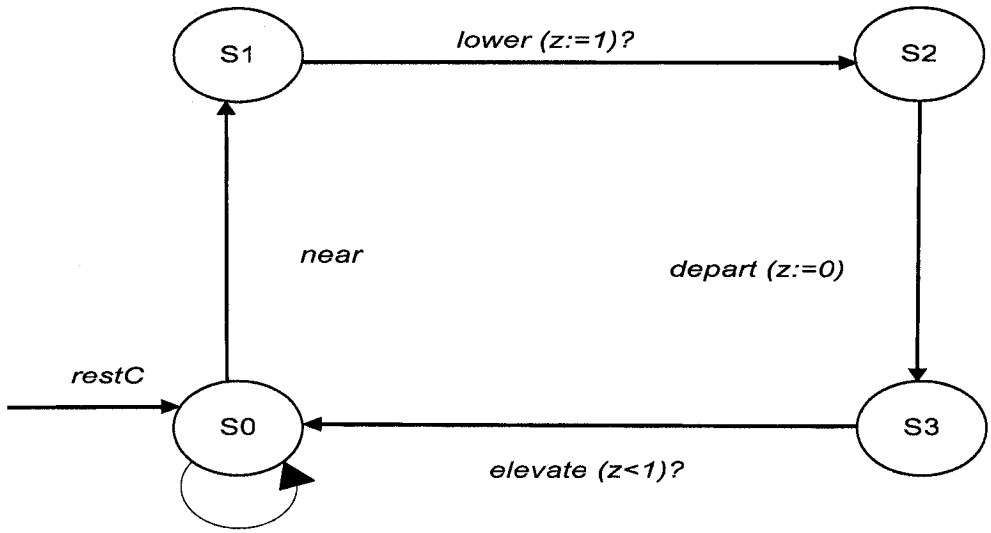

Fig. 5-20: Controller

Finally, Fig. 5-20 shows the automaton modeling of Controller. The event set is {near, depart, elevate, lower, restC} and the Controller idle state is S_0. Whenever it receives the signal "near" from the Train, it responds by sending the signal "lower" to the Gate. The response time is 1 time unit. Whenever it receives the signal "depart", it responds with a signal "elevate" to the Gate within 1 time unit. The entire system is then

[TRAIN || GATE || CONTROLLER]

5.4.1.1 Three-port TIOA

In the previous subsection, we have shown three different automata models to describe the scenarios for rail road crossing. Now we intend to build only one three-port TIOA, which is the combination of three automata models and it will be able to specify the whole operations. In fact, this is our desire or expected specification for our test case execution. By implementing centralized and distributed test architecture, we apply different test suite on *IUT* with respect to our new three-port TIOA and carefully observe the responses with their timing constraint in order to get the verdict.

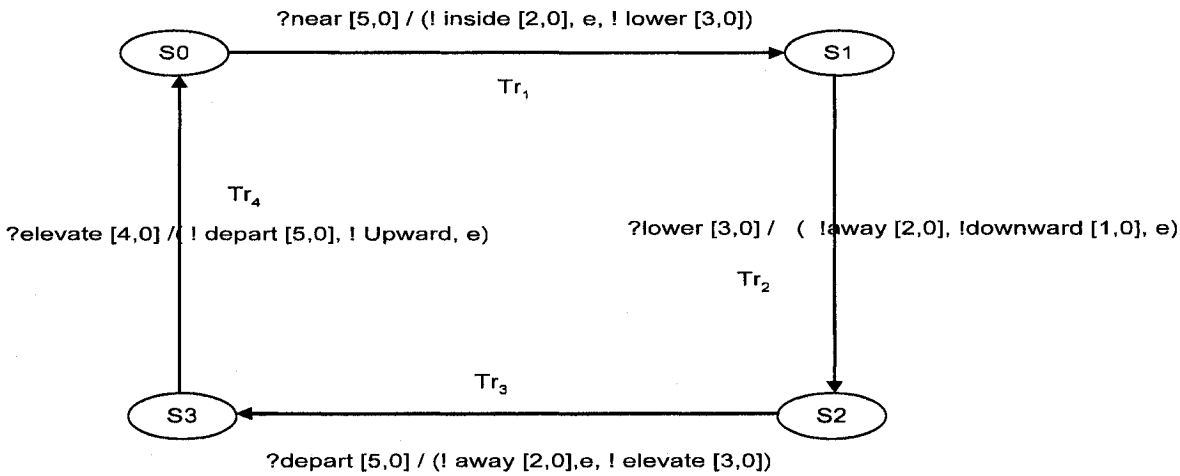

Fig. 5-21: TIOA of Train-Gate-Controller

Fig. 5-21 shows three-port TIOA of Train-Gate-Controller distributed system. In centralized test system, the environment (e.g.*TS*), which is the Synchronizer in this case, communicates with each port of *IUT* through an input queue and an output queue. First of all, we use transition tour method to generate test cases from our three-port TIOA. We consider the test sequence, which corresponds to the sequence of transitions

$Tr_1.Tr_2.Tr_3.Tr_4$, which would be <?near [5,0] / (! inside [2,0], e, ! lower [3,0])> < ?lower [3,0] / (!away [2,0], !downward [1,0], e)> < ?depart [5,0] / (! away [2,0],e, ! elevate [3,0])> <?elevate [4,0] /(! depart [5,0], ! Upward, e) >

Inputs "near[5,0]", "lower[3,0]", "depart[5,0]", "elevate[4,0]" are received by *IUT* in ports 1, 2, 3 respectively, and outputs "inside[2,0]", "away[2,0]", "depart[5,0]", "downward[1,0]", "upward", "elevate[3,0]", "lower[3,0]" are sent by *IUT* in ports 1, 2, 3, respectively.

5.4.1.2 Results

The above three port TIOA is the specification of Train-Gate-Controller distributed system, which is applied to our program.

First, we generate the test cases by transition tour method from TIOA. During the execution of a test sequence λ, *TS* guarantees (*resp. checks*) the timing constraints of the inputs (resp. outputs) of λ.. We do not consider the order constraints separately because they are implied by the timing constraint.

CENTRALIZED TEST SYSTEM RESULT ANALYSIS

Testing Object Created

Stub[repository_id=IDL:CoordinationA/SynchronizerA:1.0,key=ServiceId[service=/message,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%0000%04%00%00%00%09/message%00%20%20%20%00%00%00%0bTestManager],codebase=null] is ready.

Current time =1125019484703msecs

The synchronizer send "near" to the Train

The synchronizer send "near" to the Train

The synchronizer send "near" to the Train

The synchronizer send "near" to the Train

The synchronizer send "near" to the Train

Time's up!

Current time =1125019489703msecs

Avg sending time = 5000 msecs

Initializing the ORB

Binding to message Object

Current time =1125018504609msecs

Avg Client processing time = 1.719 msecs

Testing Object Created

Stub[repository_id=IDL:CoordinationA/SynchronizerA:1.0,key=ServiceId[service=/me

ssage,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%

0%04%00%00%00%09/message%00%20%20%20%00%00%00%0bBankManager],

codebase= null] is ready.

Current time =1125018664844msecs

The Train send "inside" to the synchronizer

The Train send "inside" to the synchronizer

The Train send "inside" to the TesterA

Time's up!

Testing Object Created

Stub[repository_id=IDL:CoordinationB/SynchronizerB:1.0,key=ServiceId[service=/sidd iq,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%00% 4%00%00%00%06/Niil%00%20%20%00%00%00%0bTestManager],codebase=null] is ready.

Current time =1125018854453msecs

The Gate send "empty" to the synchronizer

The Gate send "empty" to the synchronizer

The Gate send "empty" to the synchronizer

The Gate send "empty" to the TesterB

Time's up!

Initializing the ORB

Binding to message Object

Time's up!

Current time =1125018858641msecs

Reaction time = 4141 msecs

SUCCESS

The result is true

U:\MULTITESTER\CENTRAIZED>vbj Controller

Testing Object Created

Stub[repository_id=IDL:CoordinationB/SynchronizerB:1.0,key=ServiceId[service=/Man ager,id={11 bytes: [T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%00% 04%00%00%00%07/Naila%00%20%00%00%00%0bTestManager],codebase=null] is ready.

Current time =1125021468984msecs

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

The Controller send "lower" to the synchronizer

Time's up!

U:\MULTITESTER\CENTRAIZED>vbj TesterC

Initializing the ORB

Binding to message Object

Avg Client processing time = 0.484 msecs

Current time =1125019578156msecs

Reaction time = 7156 msecs

The reaction time of Controller is VALID

The result is true

The final Test is given below:

THE TESTERC IS true

THE TESTERB IS true

THE TESTERA IS true

THE SYSTEM IS SUCCESSFULLY PASSED

DISTRIBUTED TEST SYSTEM RESULT ANALYSIS

U:\MULTITESTER\DISTRIBUTED>vbj TesterC_Coord

Initializing the ORB

Binding to message Object

Coordination Object Created

Stub[repository_id=IDL:CoordinationC/SynchronizerC:1.0,key=ServiceId[service=/Na

ila,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%00

%04%00%00%00%07/Naila%00%20%00%00%00%0bTestManager],codebase=null] is

ready.

This is coordination message from TesterC to TesterB

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

The Controller sends "lower" to the TesterC

Time's up!

Current time =1127100832953msecs

Reaction time = 7172 msecs

FAILED

The result is false

U:\MULTITESTER\DISTRIBUTED>vbj TesterA_Coord

Testing Object Created

Stub[repository_id=IDL:CoordinationD/SynchronizerD:1.0,key=ServiceId[service=/Na

ila,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%00

%04%00%00%00%07/Naila%00%20%00%00%00%0bTestManager],codebase=null] is

ready.

This is coordination message from TesterA to TesterB

The Train sends "inside" to the TesterA

The Train sends "inside" to the TesterA

The Train sends "inside" to the TesterA

Time's up!

Current time =1127101271828msecs

Reaction time = 3187 msecs

FAILED

The result is false

U:\MULTITESTER\DISTRIBUTED>vbj TesterB_Coord_A

Initializing the ORB

Binding to message Object

Testing Object Created

Stub[repository_id=IDL:CoordinationD/SynchronizerD:1.0,key=ServiceId[service=/Na

ila,id={11bytes:[T][e][s][t][M][a][n][a][g][e][r]},key_string=%00PMC%00%00%0%

04%00%00%00%07/Naila%00%20%00%00%00%0bTestManager],codebase=null] is

ready.

This is coordination message from TesterB to TesterA

The Gate sends "empty" to the TesterB

The Gate sends "empty" to the TesterB

The Gate sends "empty" to the TesterB

The Gate sends "empty" to the TesterB

Time's up!

Current time =1127101310750msecs

Reaction time = 4156 msecs

The reaction time of Controller is VALID

The result is true

5.4.1.3 Result Analysis and Comparison

The centralized test method may result in performance problems, due to the fact that all *TS-IUT* interactions are achieved through a network. This problem has been reduced a lot by using CORBA to implement this method. Because, ORB lets objects transparently make request to-and receive responses from-other objects located locally or remotely, it lets us either statically defines our method invocations at compile time, or it lets us dynamically discover them at run time; no matter what language server objects are written in. An ORB is much more sophisticated than alternative forms of client/server middleware including traditional Remote Procedure Calls (RPCs), Message Oriented Middleware (MOM), database stored procedures, and peer to peer services.

As we use CORBA in the distributed test system too, the network problem can be reduced while coordination takes place with ORB communication channel among local testers. As a result we improve the test execution procedure with the help of CORBA architecture.

During the implementation of distributed test system, we provide three testers for three components of *IUT*. It's not difficult to deal with such *IUT*. But in the case of complex distributed system, we need to apply large number of testers and so many clocks simultaneously. As a result, the complexity of code would be higher and difficult to maintain real-time transactions. In that case, we can keep our centralized test method for conformance testing.

5.5 Summary

We provided the implementation for the timed test case execution on distributed real-time systems. We used sequence diagrams, collaboration diagrams, state diagrams and class and package diagrams to describe the design and specification. We also compared between centralized and distributed test method by analyzing their results and formal models.

In the next chapter, we will discuss the unsolved problems and possible improvements regarding our methods.

CHAPTER 6

Conclusion and Future Work

6.1 Summary

The testing methodology described in ISO IS-9646 provides a unified framework to conduct the conformance testing process. We defined conformance testing framework and discussed different test methods. Then we demonstrated the importance of modeling to be tested (the right model for the right problem).

We provided two methods for test case execution- centralized test system and distributed test system based on the specification expressed in Timed Input Output Automaton. In centralized test system, the synchronizer acts as a single tester, which controls the test sequence and observes the feedback from *IUT*. The synchronizer is composed of few internal testers and single global clock. In distributed test system, TS consists of many local testers and the same number of clocks. These communicate each other through a communication medium, which is independent of *IUT* and also maintains input and output path with different ports of *IUT*. Testers guarantee order constraints and timing constraints of inputs by sending coordination messages among each other.

We chose CORBA as a test environment to implement centralized and distributed test architectures for the following reasons:

- CORBA is well fit for modular architecture.
- CORBA is a distributed environment that guarantees the transparency of resource location and communication, despite heterogeneity and

- Our implementation model is Borland's VisiBroker for Java, which provides a complete CORBA 2.3 ORB runtime and supporting development environment for building, deploying, and managing distributed Java applications that are open, flexible, and inter-operable.

Both of our methods are applicable to various kinds of applications, which could be modeled by np-TIOA. We discussed in detail Train-Gate-Controller distributed system. Also, we created automata models expressed in 3-port TIOA as specification. We made real time transactions, in order to send test cases to different ports of *IUT* and to receive the output as well. We can also apply our two methods on video-on-demand service where a server multicasts a video stream to different number of clients. We can consider a test sequence (consisting of a single input/outputs transition) with a single input, corresponding to data stream, which is multicast by the server and an output of each client, corresponding to the data stream, which is received by each client and a timing constraint between the sending of the input and the reception of each output.

Again, some applications need to acquire multiple inputs before issuing an output. An example of such kind of applications is a cruise control system that has to acquire information about, among others, the current speed, desired speed, and operating mode, before deciding whether to increase or decrease the speed.

For the sake of simplicity, we always provide a single clock for both methodologies during implementation. But theoretically, we provide single clock for only centralized test system and multiple clocks are used depends upon the number of testers in distributed test system.

6.2 Future Work

We have applied and tested our two methods on Train-Gate-Controller distributed system and they performed very well in CORBA platform but still there are some provisions or scope of works to do in future.

Efficient Test Case Generation Method: Our timed test case execution procedure has been implemented by using only transition tour method in the purpose of test case generation for the sake of simplicity. There are some limitations for transition tour method to cover maximum faults from n port input output automata. We intend to apply most efficient test generation methods like W_p, timed W_p or DS methods into our test execution procedure.

Complex distributed Systems: We intend to apply our two methods on concrete and complex examples, for example in the areas of communications protocols and robotics. Also, we need to apply and possibly adapt the proposed methods for testing multimedia applications with timing constraints.

Bibliography

[1] A. Khoumsi. *Testing distributed real-time systems: an efficient method, which ensures controllability and optimizes observability* Intern. Conf. on Real-Time Comp. Syst. and Applic. (RTCSA), Tokyo, Japan, March 2002.

[2] D. Mandroli, S. Morasca, and A. Morzenti. *Generating test cases for real-time systems from logic specifications.* ACM Transactions on Computer Systems, 13(4):365-398, November 1995.

[3] D. Clarke and I. Lee. *Automatic generation of tests for timing constraints from requirements.* In Third International Workshop on Object-Oriented Real-Time Dependable Systems, Newport Beach, California, February 1997.

[4] A. En-Nouaary, R. Dssoulie, and A. Elqortobi. *Génération de tests temporiss.* In 6th Colloque Francophone de l'Ingnierie des Protocols. HERMES, 1997.

[5] j. Springintveld, F. Vaadranger, and P. Dargenio. *Testing timed automata.* Technical Report CTIT97-17, University of Twente, Amsterdam, The Netherlands, 1997.

[6] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. *Timed test generation based on state characterization technique.* In 19th IEEE Real-Time Systems Symposium (RTSS), Madrid, Spain, December 1998.

[7] A. Khoumsi, M. Akalay, R. Dssouli, A. En-Nouaary, and Granger. *An approach for testing real time protocol entities. In 13th Intern.Workshop.On Testing of Communicating Systems (TestCom),* Ottawa, Canada, Kluwer Academic Publishers. Aug-Sept. 2000.

[8] A. Khoumsi, A. En-Nouaary, R. Dssouli, and M. Akalay. *A new method for testing real time systems. In 7th Intern. Conference* on Real-Time Computing Systems (RTCSA), Cheju Island, South Korea, December 2000.

[9] G. Luo, R.Dssouli, G.v.Bochmann, P. Venkataram, and A. Ghedamsi. *Test generation with respect to distributed interfaces*. Computer standards and Interfaces 16: 119-132, 1994.

[10] C. Jard, T. Jeron, H. Kahlouche, and C. Viho. *Towards automatic distribution of testers for distributed conformance testing.* In PSTV/FORTE, Paris, France, November 1998.

[11] C. Jard, T. Jeron, L. Tanguy, and C. Viho. *Remote testing can be as powerful as local testing.* In PSTV/FORTE, Beijing, China, October 1999.

[12] J. Zhang, S.C. Cheung, and S. T. Chanson. *Stress testing of distributed multimedia software systems.* In PSTV/FORTE, Beijing, China, October 1999.

[13] J. Bi and J. Wu. *A formal approach to conformance testing of distributed routing protocols*. In PSTV/FORTE, Beijing, China, October 1999.

[14] L. Cacciari and O. Rafiq. *Controllability and observability A formal approach to in distributed testing.* Information and Software technology, 1999.

[15] A. Khoumsi. *Timing issues in testing distributed systems*. In 4th IASTED Intern. Conf. on Software Engineering and Applications (SEA), Las Vegas, USA, November 2000.

[16] A. Khoumsi. *A centralized method for testing distributed real-time test systems*. In 10th IEEE North Atlantic Test Workshop (NATW), Gloucester, Massachusetts, USA, May 2001.

[17] A. Khoumsi. *New results for testing distributed real-time reactive systems using a centralized method.* In 20th IASTED Intern. Multi-Conf. on Applied Informatics,

Symposium on Parallel and Distributed Computing and Networks, Innsbruck, Austria, February 2002.

[18] R.Dssouli, K. Saleh, E. Aounlhamid, A. En-Nouaary and C. Bourhfir. *Test Development for Communication Protocols Towards Automation.* Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 31, Issue 17, Pages: 1835 - 1872, June 1999.

[19] S.Natio and M. Tsunoyama, *Fault Detection for sequential Machines by transition Tours,* Proc. 11th IEEE Fault Tolerant Computing Symposium (1981).

[20] B. Sarikaya and G.v. Bochmann, *Some experience with Test Sequence Generation for Protoclos*, C. Sunshine, ed., Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification, Pages: 555 – 567,1982.

[21] G.v. Bochmann and C.A. Sunshine, *A Survey of Formal Methods*, in P.E. Greened., Computer Networks and protocols (Plenum Press, New York) chapter 20. 561-578, 1983.

[22] S. Naito and M. Tsunoyama, *"Fault detection for sequential machines by transition-tours,"* in proc. FTCS (Fault Tolerant Comput. Syst) Proccedings of the 11th. IEEE Fault Tolerant Computing Symposium, pages 238--243, 1981.

[23] ITU-T, TTCN-2. The Tree and Tabular Combined Notation (TTCN). *Conformance Testing Methodology and Framework*, part 3, Recommendation X.292, 1997.

[24] B. Nielsen and A. Skou. *Automated Test Generation from Timed Automata.* Proc. Workshop Tools and Algorithms for the construction and Analysis of Systems, Apr. 2001.

[25] S. Naito, M.Tsunoyama. *Fault detection for sequential machines by Transition Tours.* Proc. Fault Tolarent Computer Systems, pp.238-243, 1981.

[26] G. Gonec. *A Method for the design of fault detection experiments.* IEEE Transactions on Computers C-19 551-558. 1970.

[27] K. Sabnani and A. Dahbura. *A New Technique for Generating Protocol Test.* ACM Computer Communications, 15(4), September 1985.

[28] T.S. Chow. *Testing Software Design Modeled by Finite State Machine.* IEEE Transactions on Software Engineering, Vol. 4, pp:178-187, 1978.

[29] S. Fujiwara, and G.v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. *Test selection based on finite state models. IEEE Transactions on Software Engineering*, Vol. 17, pp.591-603, June 1991.

[30] R.Alur and D.Dill. *A theory of Timed Automata.* Theoretical Computer Science, 126:183-235, 1994.

[31] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. *Compiling Real-Time Specifications into Extended Automata.* IEEE Transactions on Software Engineering, 18(9): 794-804, September 1992.

[32] Object Management Group, *"The Common Object Request Broker Architecture: Architecture and Specifications"*, version 3.0, July 2002.

[33] Jeremy L. Rosenberger, *Teach Yourself CORBA in 14 days*, Sams Publishing 1998.

[34] Cathy Hustich, *"CORBA For Real-Time, High Performance and Embedded Systems"*, Fourth IEEE International Symposium an Object Oriented Real-Time Distributed Computing, 200. pp. 345-349.

[35] Real-time CORBA, Version 1.1. The Object Management Group, August 2002, available electronically http://cgi.omg.org/docs/formal/02-08-02.pdf

[36] Object Management Group, *CORBA Messaging Specification.* Object Management Group, Document orbos/98-05-05 ed, May 1998.

[37] Ahmed Khoumsi, *Timing Issues in Testing Distributed Systems*, IASTED Intern. Conf. on Soft. Engin. and Applic. (SEA), Las Vegas, USA, November 2000.

[38] G. Manimaran, H. S. Rahul, and C. Siva Ram Murthy. *"A new distributed route selection approach for channel establishment in real-time networks."* IEEE/ACM Trans. Networking, vol. 7, no 5, pp 698-709, Oct. 1999.