TESTING DISTRIBUTED REAL-TIME SYSTEMS WITH A DISTRIBUTED TEST APPROACH

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖKHAN ÖZTAŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

MAY 2008

Approval of the thesis:

TESTING DISTRIBUTED REAL-TIME SYSTEMS WITH A DISTRIBUTED TEST APPROACH

submitted by **GÖKHAN ÖZTAŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department**, **Middle East Technical University** by,

Prof. Dr. Canan ÖZGEN ________________
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. İsmet ERKMEN ________________
Head of Department, **Electrical and Electronics Engineering**

Asst. Prof. Dr. Şenan Ece SCHMIDT ________________
Supervisor, **Electrical and Electronics Engineering Dept., METU**

Examining Committee Members

Prof. Dr. Semih BİLGEN ____________
Electrical and Electronics Engineering Dept., METU

Asst. Prof. Dr. Şenan Ece SCHMIDT ____________
Electrical and Electronics Engineering Dept., METU

Assoc. Prof. Gözde Bozdağı AKAR ____________
Electrical and Electronics Engineering Dept., METU

Asst. Prof. Dr. Cüneyt BAZLAMAÇCI ____________
Electrical and Electronics Engineering Dept., METU

Ali BİLGİN (M.Sc.) ____________
Expert Engineer, ASELSAN

Date: (08/05/2008)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Gökhan ÖZTAŞ

Signature :

ABSTRACT

TESTING DISTRIBUTED REAL-TIME SYSTEMS WITH A DISTRIBUTED TEST APPROACH

Öztaş, Gökhan

M.S., Department of Electrical and Electronics Engineering

Supervisor: Asst. Prof. Dr. Şenan Ece Schmidt

May 2008, 77 pages

Software testing is an important phase the of software development cycle which reveals faults and ensures correctness of the developed software. Distributed real-time systems are mostly safety critical systems for which the correctness and quality of the software is much more significant. However, majority of the current testing techniques have been developed for sequential (non real-time) software and there is a limited amount of research on testing distributed real-time systems. In this thesis, a proposed approach in the academic literature "testing distributed real-time systems using a distributed test architecture" is implemented and compared to existing software testing practices in a software development company on a case study. Evaluation of the results show the benefits of using the considered distributed test approach on distributed real-time systems in terms of software correctness.

Keywords: Software Testing, Distributed Real-Time Systems, Software Testing Techniques

ÖZ

GERÇEK ZAMANLI DAĞITIK SİSTEMLERİN DAĞITIK BİR YAKLAŞIMLA TEST EDİLMESİ

Öztaş, Gökhan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Şenan Ece Schmidt

Mayıs 2008, 77 Sayfa

Yazılım testi yazılım geliştirme sürecinde hataların bulunmasını sağlayan ve geliştirilen yazılımın doğruluğunu garantileyen önemli bir aşamasıdır. Gerçek zamanlı dağıtık sistemler genellikle yazılım güvenirliğinin ve kalitesinin çok önemli olduğu güvenlik kritik sistemlerdir. Fakat şu anki test tekniklerinin çoğu sıralı (gerçek zamanlı olmayan) yazılımlar için geliştirilmiştir ve gerçek zamanlı dağıtık sistemlerin testi üzerine yapılmış sınırlı miktarda çalışma bulunmaktadır. Bu tez kapsamında akademik literatürde önerilen "Gerçek zamanlı dağıtık sistemlerin dağıtık test mimarisi ile testi edilmesi" bir örnek çalışma üzerinde uygulanmıştır ve bir yazılım geliştirme firmasında uygulanan yazılım test yöntemi ile karşılaştırılmıştır. Sonuçların değerlendirilmesi gerçek zamanlı dağıtık sistemler için dağıtık test mimarisi kullanımının yazılım doğruluğu açısından yararlarını göstermektedir.

Anahtar Kelimeler: Yazılım Testi, Gerçek Zamanlı Dağıtık Sistemler, Yazılım Test Teknikleri

To my Family

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor Asst. Prof. Dr. Şenan Ece Schmidt for her guidance, advice, criticism, encouragements, insight throughout this thesis study and my whole graduate life.

I would like to thank my colleagues in ASELSAN Inc for their valuable support.

I would like to thank ASELSAN for providing tools and other facilities throughout this study.

Thanks a lot to all my friends for their great encouragement and their valuable help to accomplish this work.

I would also like to thank my family for giving me encouragement during this thesis and all kind of supports during my whole education.

Finally, I would like to thank to my dear Duygu for her endless love and encouragement throughout this entire journey.

TABLE OF CONTENTS

ABSTRACT iv
ÖZ vi
ACKNOWLEDGEMENTS ix
TABLE OF CONTENTS x
LIST OF TABLES xii
LIST OF FIGURES xiii
ABBREVIATIONS xiv
CHAPTER
1. INTRODUCTION 1
2. BACKGROUND 4
2.1. Real-Time Concepts 4
2.1.1. Hard Real-Time Systems 6
2.1.2. Soft Real-Time Systems 6
2.1.3. High-Performance I/O 6
2.2. Distributed Real-Time Systems 7
2.3. Software Testing 9
3. TESTING DISTRIBUTED REAL-TIME SYSTEMS 17
3.1. Related Work 17
3.2. Current Approach Used in a Company for Testing Distributed Real-Time Systems...23
4. TESTING DISTRIBUTED REAL TIME SYSTEMS WITH DISTRIBUTED TEST ARCHITECTURE..25
4.1. Timed Automata with N Ports 28
4.2. Fault Model...30
4.3. Test Architecture...31

4.4. Defining Test Sequence 32
4.5. Conformance of IUT to GTS 33
4.6. Test Execution 34
4.7. Coordination of Testers for Solving Controllability and Observability Problem 34
4.7.1. Coordination Method for Order Constraints 35
4.7.2. Coordination Method for Timing Constraints 36
4.8. Algorithm for Distributing Global Test Sequence on to Local Testers 38
5. IMPLEMENTATION OF IEEE 1588 PRECISION TIME PROTOCOL ON A DISTRIBUTED TEST SYSTEM 44
5.1. IEEE 1588 PRECISION TIME PROTOCOL 44
5.2. Implementation of PTP on a Distributed Test Architecture 47
6. IMPLEMENTATION 49
6.1. IUT Architecture and Target System 49
6.1.1. PowerPC 7410 Daughtercard 50
6.1.2. IUT Architecture 51
6.2. 4P-Timed Automata Model of the IUT 53
6.3. Developed Tool for Automatic Local Test Sequence Generation 56
6.4. Global Test Sequence and Local Test Sequences 58
6.5. Method for Analysis of Test Results 59
6.6. Implementation of Local Testers 60
6.7. Test Execution and Analysis of Test Results 66
6.8. Test Results for Different Global Test Sequences 70
6.9. Evaluation of Test Results 70
7. CONCLUSION 72
REFERENCES 74
APPENDIX A 76
PowerPC 7410 Daughtercards Specifications 76

LIST OF TABLES

TABLES

Table 6-1 Test Results for First Version of IUT 66

Table 6-2 Test Results for Second Version of IUT 68

Table 6-3 Test Results for Third Version of IUT 69

LIST OF FIGURES

FIGURES

Figure 2-1 Distributed Real Time System 8
Figure 3-1 Real Time System Model for [1] 18
Figure 3-2 CTIOA Model of Real-Time System....19
Figure 3-3 Time Petri Net [12] 21
Figure 3-4 Test Approach Used In the Company 23
Figure 4-1 Distributed Test Architecture 26
Figure 4-2 Distributed Real Time System with 2 Nodes 27
Figure 4-3 2p-Timed Automata 29
Figure 4-4 Distributed Test Architecture for 2p-Timed Automata 32
Figure 4-5 2p-“Distribution of a GTS into N_s LTS” Algorithm Pseudocode 41
Figure 5-1 Offset Calculation with IEEE 1588 PTP....46
Figure 5-2 2p Local Test Sequence Execution with PTP Implementation 48
Figure 6-1 PowerPC 7410 Daughtercard 51
Figure 6-2 IUT Architecture....53
Figure 6-3 4p-Timed Automata Model of IUT 55
Figure 6-4 Developed Tool for LTS Generation 57
Figure 6-5 Pseudocode of $Tester^a$ 62
Figure 6-6 Pseudocode of $Tester^b$ 63
Figure 6-7 Pseudocode of $Tester^c$ 64
Figure 6-8 Pseudocode of $Tester^d$ 65

ABBREVIATIONS

API : Application Programming Interface
CPU : Central Processing Unit
CTIOA : Communicating Timed Input Output Automata
EC : Enabling Condition
ECU : Electronic Control Unit
EOG : Execution Order Graph
FIFO : First Input First Output
GTS : Global Test Sequence
I/O : Input/Output
IUT : Implementation Under Test
LTS : Local Test Specification
MC/OS : Multicomputer Operating System
np-TA : n Port Timed Automaton
PC : Personal Computer
PTP : Precision Time Protocol
TIOA : Timed Input Output Automata
TS : Test System

CHAPTER 1

INTRODUCTION

Real-Time systems are important parts of our daily lives. A real-time application is an application where the correctness of the application depends on the timeliness and predictability of the application as well as the results of computations [1]. *Distributed real time systems* are a special case of real time systems where multiple computing nodes are connected on a network [3]. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units. Applications of distributed real time systems include many critical systems such as manufacturing, instrumentation, surveillance, multi-vehicle control, avionics systems, automotive systems and scientific experiments.

Software testing is an indispensable part of software development projects and in most cases can increase the development cost dramatically. Important performance metrics of software testing include t*est coverage*, *controllability*, *observability*, *reproducibility* and *determinism*.

For safety-critical computer based systems, software testing is much more important because of strict reliability and safety requirements. However, most safety critical computer based systems are distributed real-time systems, and the majority of current testing and debugging techniques have been developed for sequential (non real-time) programs.

Achieving deterministic testing of sequential programs is easy because we need only to control the sequence of inputs and the start conditions, in order to guarantee

reproducibility. That is, given the same initial state and inputs, the sequential program will deterministically produce the same output on repeated executions. Sequential software testing techniques cannot be applied to distributed-real time systems because they neglect timing issues.

There is a lot of research on testing of sequential software but there is a limited study in the literature for testing of distributed real time systems. Furthermore there are limited implementations and outcomes reported for these techniques on distributed real time systems.

In this thesis, first a selected systematic approach to software testing for distributed embedded systems that is proposed in the academic literature is investigated. This approach is then implemented on a real software project that has been developed and previously tested in a software development company. In this company, testing for real time disturbed systems is carried out at a system level without individual testing of the software apart from the rest of the distributed system. A case study on this software project is performed to demonstrate the benefit of applying the new systematic approach by comparing the achieved testing results to that of previous tests held for the same project in terms of the software testing performance metrics. Evaluation of the results show the benefits of using distributed test approach on distributed real-time systems in terms of software correctness.

An important problem for testing of distributed real-time systems is the clock synchronization among different components of the system. This problem is solved in this thesis by implementing IEEE 1588 Precision Time Protocol for the software project under investigation. Designing tests for distributed systems involves computing individual test sequences for each component from a global test sequence. A software tool is developed in this thesis to perform this computation automatically.

After this introductory chapter, the rest of the thesis is organized as follows. In the second chapter, necessary background on real time concepts, distributed real time

systems and software testing are included. In the third chapter, academic literature on testing of distributed real-time systems is investigated and current practice in a software development company for testing of distributed real-time systems is explained. In the fourth chapter, selected approach for testing of distributed real time systems is explained. In the fifth chapter, proposed method, IEEE 1588 Precision Time Protocol, in this thesis for solving synchronization problem is explained. In the sixth chapter, implementation details of the selected approach on a case study are given and software developed for computing individual test sequences is presented. Test execution results are presented and evaluation of test results comparing with the current practice in a software company is given. Finally in the seventh chapter, conclusions and further work for this study is given.

CHAPTER 2

BACKGROUND

This chapter describes the real time concepts, distributed real time systems and software testing.

2.1. Real-Time Concepts

The concept of real-time digital computing systems is an emergent concept compared to most engineering theory and practice. When requested to complete a task in real-time, the common understanding is that this task must be done upon request and completed while the requester waits for the completion as an output response. If the response to the request is too slow, the requestor may consider lack of response as a failure. The concept of real-time computing is really no different. Requests for real-time service on a digital computing platform are most often indicated by asynchronous interrupts. More specifically, inputs that constitute a real-time service request indicate a real world event sensed by the system for example, a new video frame has been digitized and placed in memory for processing. The computing platform must now process input related to the service request and produce an output response prior to a deadline measured relative to an event sensed earlier.

A real-time application is an application where the correctness of the application depends on the timeliness and predictability of the application as well as the results of computations [10]. Examples of real-time applications include process control, factory automation robotics, vehicle simulation, scientific data acquisition, image

processing, built-in test equipment, music or voice synthesis, and analysis of high-energy physics.

Real-time applications provide an answer or an action to an external event in a timely and predictable manner [10]. While many real-time applications require high-speed compute power, real-time applications cover a wide range of tasks with differing time dependencies.

Timeliness has a different definition in each real-time application [10]. What may be fast in one application may be slow or late in another. For example, a vehicle engine control needs to collect data and control actuators with microseconds accuracy, while a scientist monitoring the air pressure might need to collect data in intervals of several minutes. However, the success of both applications depends on well-defined time requirements.

The concept of *predictability* may vary from field of operation, but for real-time applications it generally means that a task or set of tasks can always be completed within a predetermined amount of time [10]. Depending on the situation, an unpredictable real-time application can result in loss of data, loss of deadlines, or loss of plant production.

Real-time applications are usually characterized by a blend of requirements. Some portions of the application may consist of hard, critical tasks, all of which must meet their deadlines. Other parts of the application may require heavy data throughput. Many parts of a real-time application can easily run at a lower priority and require no special real-time functionality. The key to a successful real-time application is the developer's ability to accurately define application requirements at every point in the program. Resource allocation and real-time priorities are used only when necessary so that the application is not overdesigned.

Real time systems are generally developed on *a host system* but executed on *a target system* [8]. Host system is a capable system with development tools and used

for software development. Target system is generally more resource constrained system for the execution of the real time system. A PC can be used as a host system whereas a single board computer can be used as a target system.

Real-time applications can be classified as either *hard real-time* or *soft real-time* according to acceptability to of delays.

2.1.1. Hard Real-Time Systems

Hard real-time applications require a response to events within a predetermined amount of time for the application to function properly [15]. If a hard real-time application fails to meet specified deadlines, the application fails. While many hard real-time applications require high-speed responses, the granularity of the timing is not the central issue in a hard real-time application. An example of a hard real-time application is a missile guidance control system where a late response to a needed correction leads to disaster.

2.1.2. Soft Real-Time Systems

Soft real-time applications do not fail if a deadline is missed [15]. Some soft real-time applications can process large amounts of data or require a very fast response time, but the key issue is whether or not meeting timing constraints is a condition for success. An example of a soft real-time application is an airline reservation system where an occasional delay is tolerable, but unwanted.

2.1.3. High-Performance I/O

Many real-time applications require high I/O throughput and fast response time to asynchronous external events [15]. The ability to process and store large amounts of data is a key metric for data collection applications. Real-time applications that require high I/O throughput rely on continuous processing of large amounts of data. High data throughput requirements are typically found in signal-processing applications such as:

- Telemetric applications
- Radar analysis applications
- Sonar
- Speech analysis

For some applications, the throughput requirements on a single channel are modest. However, an application may need to handle multiple data channels simultaneously, resulting in a high aggregate throughput. Real-time applications, such as medical diagnosis systems, need a response time of about one second while simultaneously handling data from, perhaps, ten external sources.

High I/O throughput may be important for some real-time control systems, but another key metric is the speed at which the application responds to asynchronous external events and its ability to schedule and provide communication among multiple tasks. Real-time applications must capture input parameters, perform decision-making operations, and compute updated output parameters within a given timeframe.

Some real-time applications, such as flight simulation programs, require a response time of microseconds while simultaneously handling data from a large number of external sources. The application might acquire several hundred input parameters from the cockpit controls, compute updated position, orientation, and speed parameters, and then send several hundred output parameters to the cockpit console and a visual display subsystem.

2.2. Distributed Real-Time Systems

We assume a distributed real-time system consisting of a set of nodes. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of an external process [10]. Typically, each computing node is connected to sensors, actuators. Set of connected nodes is called cluster. Architecture of distributed real-time system can be easily visualized with the help of Figure 2-1.

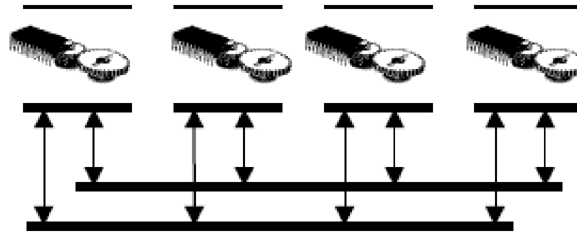

Figure 2-1 Distributed Real Time System

Communication between the nodes is achieved by messages passing and the processes can use synchronization to maintain a precedence relation or mutual exclusion between processes on different nodes. Processes on the same node also use the communication service. A designer of real-time system often chose distributed solutions because of increasing complexity and safety requirements. A distributed solution makes it possible to achieve greater reliability through redundancy. Also the inherited distribution of the system, for example, control systems on a factory floor can be a cause to choose a distributed solution. Applications include manufacturing, instrumentation, surveillance, multi-vehicle control, avionics systems, automotive systems and scientific experiments. For example, a modern car in the premium segment has 40 or more computers (ECUs). Since each computer interacts with physical processes, the passage of time becomes a central feature; it is this key constraint that distinguishes these systems from distributed computing in general.

In addition to interacting over a communication network, the nodes in a distributed embedded system interact through the physical world. Driving an actuator at one node, for example, may affect the data sensed at another node. Moreover, actuation may need to be orchestrated across nodes. The required precision of that orchestration, of course, depends on the application. Robotic applications, e.g. in manufacturing, may require precisions on the order of milliseconds. Instrumentation, where stimuli are generated and responses are measured, may require precisions on the order of nanoseconds or even higher.

2.3. Software Testing

There are several standards for the terminology used when discussing software testing. Terminology that is used in this study is given below.

- **Software Specification** is a complete description of the behavior of the software to be developed [9].
- **Correctness** means the behavior of the program execution conforms to the behavior specified in the software specification [9].
- **Test** is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspects of the system or component [9].
- **Validation** is the process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements [9], i.e., validation aims at answering the question are we building the right system?
- **Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [9], i.e., verification aims at answering the question are we building the system right?

With the evolving technology we are getting more and more dependent on computers and their software. In our daily life when we travel by airplane, use robots at work, or even watch TV at home, we expect them not to fail or malfunction. Therefore, it is important that the software does what the user expects and that it does not fail. To establish the quality of the software Validation and Verification are used [18]. Software testing is just one kind of verification. Software testing is the process used to measure the quality of developed computer software [18]. Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements such as capability,

reliability, efficiency, portability, maintainability, compatibility, and usability. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate [16]. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behavior of the product against a specification [16].

Software testing procedure consists of generating *test cases* and applying them to *Implementation Under Test* (IUT) and testing consists of three steps [16]:

- Modeling software's environment
- Extract test cases that are likely to reveal most failures
- Apply the test cases to the software in a test execution and evaluate results

Modeling Software's environment: The task of testing starts by modeling the software's environment. Interaction between software and its environment must be identified and simulated. Each interface and input that software uses must be defined very well.

Test Case Generation: A test case is a software testing document, which consists of event, action, input, output, expected result, and actual result. A test case is an input and an expected result. Test cases can be generated from specification of the IUT. Specification based test-case generation derives the test cases from the specification; hence the software can be tested early in the development even before the implementation is completed.

Infinite number of test cases can be generated from the specification of IUT but only a subset can be applied to IUT for realistic software test process. Coverage that includes code coverage (execution of each source line at least once) and input

coverage (applying all possible inputs to IUT) defines the completeness of software testing process for selected subgroup of test cases [16]. If code and input coverage is sufficient enough final product of IUT has fewer bugs. Typically it is aimed to find a subgroup of test cases that leads to find the most bugs. To achieve this, test cases that occur often in the field are selected.

Test Execution and Evaluation: Test execution includes running test cases on the IUT and observing outputs. The correctness of a program can only be established according to what have been observed during test executions. Hence, it is essential that we can observe the produced output, intermediate results and the paths traversed by the program. Observation of execution behaviors can be done by using software probes, hardware instrumentation, or a combination of software and hardware instrumentation. Evaluation of test results includes comparison of observed outputs during test execution to expected outputs stated by software specification. Deviations from software specification are considered as failures.

Software testing can be classified according to scope of modeling software's environment into three groups: *unit*, *integration* or *system* testing [16].

- **Unit Testing** ignores the rest of the system except the unit to be tested. Unit testing mainly focuses on testing individual component of a system and takes into account inputs and outputs of individual module.
- **Integration Testing** focuses on testing of multiple components that have previously tested on unit testing level separately. Subset of overall system domain that represents communication between these modules is tested for conformance to requirements.
- **System Testing** takes into account entire system domain and tests collection of components that constitutes a final product.

Test case generation step determines what type of testing is to be executed on IUT and there are mainly two types of testing: *Functional Testing* and *Structural Testing* [16].

- **Functional Testing** requires the selection of test cases without considering the structure and source code of the implementation. For test case generation, execution and evaluation; the specification of the required functionality at defined interfaces must be used not the attributes of code or data structures. Functional Testing is also named as *specification-based testing, behavioral testing and black-box testing*.
- **Structural Testing** requires the selection of test cases considering the structure and source code of the implementation. Test suite is generated from the implemented structures and inputs are based on the structure of the source code and data structures. Structural testing is also named as *code-based testing and white-box testing.*

Typically, white-box testing is more predominant in early test phases where the complexity of tested objects is still (relatively) low. Later test phases, on the other hand, rely more heavily on black-box testing techniques

Software testing procedure finds bugs and these bugs are fixed at a newer version of the software. Any specific fix can introduce new bugs or fail to remove the bug in the newer version of the software. *Regression Testing* includes applying same test cases to IUT to find whether new bugs are introduced or bugs are fixed. Testing procedure progresses through software versions until a release version is reached and for each version regression testing is applied.

Basic concepts of software testing which will be evaluated in this study are listed below.

Observability: Observability is the ability to observe the state before and after an operation [17]. Consequently, it must be possible to observe the input, output and the internal state. Observing the input and output in sequential programs is straightforward, that is if the program does not include any non-deterministic statements. The inputs are observed to determine the behavior of the program's

environment. By observing the internal state, the exact cause of the failure can be located and internal state changes that have no effect on the output can be detected.

Determinism: Executions of sequential programs are repeatable and deterministic [10]. That is, for an input we get the same output regardless of how many times we run the program with that input. This is true if the program does not include any statements that depend on the temporal behavior and/or random behavior. Examples of such statements are random statements or dependencies of clock readings in sequential programs.

Controllability: Controllability is the ability to force the program into a desired state [17]. For sequential programs it is sufficient to give the input to the program to achieve controllability.

Reproducibility: Reproducibility, test repeatability, is the ability to reproduce a previous execution of a program [10]. In other words, for a given input the system always computes the same output in repeated runs of the system. After errors have been corrected the tester wants to assure that the errors have been removed and that no new errors have been introduced. Therefore it is necessary to test the system repeatedly. During repeated test runs with the same test cases, the same outputs must be observed in order to determine if the software is correct. If test executions are not reproducible re-testing cannot determine that corrections have removed the errors. For real-time systems, in which temporal behavior have impact on the execution path, the program is not usually reproducible. To reproduce the exact execution behavior of a sequential program it is sufficient to run the program repeatedly with the same input. In order to reproduce the execution behavior of real time programs it is not sufficient to repeatedly feed the same input to guarantee the same output.

Based on the execution behavior, computer software can be categorized into three domains:

- *Sequential programs*, which are programs that runs from invocation to termination without interruptions or interleaving.
- *Concurrent programs*, which are programs that execute within the same time interval either by interleaved or simultaneous execution.
- *Real-time systems*, which are programs where the correctness depends on the functional behavior as well as the temporal behavior.

For these domains, the objective of testing is to find deviations between the specified requirements and the observed results during operation of the software.

A real-time computer system must provide the intended service in two dimensions: the functional (value) dimension and the temporal dimension. The verification of a real-time system implementation is thus necessarily more complex than the verification of a non-real-time system which has to be checked in the value dimension only. Deterministic and reproducible testing of sequential software (non-real time) can be achieved by controlling the sequence of inputs to software. On the other hand it is not so straightforward to test distributed real time systems. Contrary to the non real-time systems, the behavior of real-time systems depends on both the interactions with the environment and the timing for these interactions. The behavior of distributed real time system depends on the order and timing of inputs. Misbehaviors in real-time systems are generally due to the non respect of timing requirements. On the other hand distributed real time systems include mostly safety critical systems. Testing of these real-time safety critical systems before deployment becomes mandatory to avoid critical failures.

The complexity of real-time systems can be expressed in terms of the number of possible valid execution paths that are traversed during the operation of the system. Complexity caused by the indeterminacy in the interaction between the environment and the program makes exhaustive testing impossible. The complexity increases even more when tasks in the system interact with each other, i.e. interprocess

communication. Hence, out of all generated test cases only a very small subset of test cases is chosen.

An important aspect in the testing of an implementation of a system is the fault model. The power of a test cases generation technique to detect faults in an implementation is referred to as fault coverage. Test cases generation methods can be compared based on their respective fault coverage. We can say that a method A is more powerful than a method B, if A has a better fault coverage than B. In other words, a method A is more powerful than a method B, if A detects more faults than B. However, for a more accurate comparison between test cases generation techniques, other parameters such as the length of test suites should be taken into account.

Most of the real-time systems have *state-based behavior* and state-based modeling is used to model these systems. System history determines the current state of the system and each state is clearly distinguishable from other states for state-based behavior. Some terms related with state-based modeling can be stated as:

- **State:** Abstract situation in the life cycle of a system entity
- **Event:** A particular input (for instance, a message or method call)
- **Action:** The result, output or operation that follows an event
- **Transition:** An allowable two-state sequence, that is, a change of state ("firing") caused by an event
- **Guard:** Predicate expression associated with an event, stating a Boolean restriction for a transition to fire

Verification of systems that exhibit state-based behavior includes whether in response to input events the correct actions are taken and correct final state is reached. State based models can be used by state-based test design techniques to derive test cases for these kinds of systems [21]. State-based test design techniques aim to derive test cases that verify relationship between events, actions, states, and state transitions. State-based behavior can be represented commonly by using

statecharts, and are often described using UML or automata models. Because test cases are derived from models and not from source code, state-based testing is usually seen as one form of black-box testing. State transition test technique described in [21] is just one type of state-based test design techniques. The aim of this technique is to derive all possible paths of transitions starting from initial state and either ending up with final state or initial state.

Test design techniques vary for *coverage criteria* since each test design technique can focus on covering paths or on the possible variations at each decision point. They comprise different kinds of coverage. Widely known types of coverage that can be related with state-based test design techniques are listed in [21] as:

- "**Statement coverage:** Each action is executed at least once"
- "**Branch coverage or Decision Coverage:** Each action is executed at least once and every possible result (true or false) of a decision is completed at least once – this implies statement coverage"
- "**Path coverage:** In the case of path coverage, the focus is on the number of possible paths. The concept of "test depth level" is used to determine to what extent the dependencies between consecutive decisions are tested. For test depth level n, all combinations of n consecutive decisions are included in test paths"

Existing software testing strategies for distributed real time systems are given in the literature survey section.

CHAPTER 3

TESTING DISTRIBUTED REAL-TIME SYSTEMS

Testing is one of the most widely known and most widely applied verification techniques, and is thus of large practical importance, but on the other hand it has not been very thoroughly investigated in the context of real-time systems. Significant studies in the literature on testing for distributed real-time systems are given in the next section.

All of the studies that are explained use a modeling technique to model the real-time system such as; Communicating Timed Automata and Timed Petri Net. Modeled systems differ in each study since some of them model a single process and some of them model all the system with more than one process interacting with each other.

3.1. Related Work

In [1] author propose a test architecture for real time systems which contains component to be tested named as IUT and the rest of the system that interacts with IUT named as *context* as shown in Figure 3-1. Real time system is modeled with Communicating Timed Input Output automata (CTIOA). In this model, a Timed Input Output Automaton (TIOA) is used to model a single network node and nodes can communicate with each other via channels between different TIOAs [20]. One CTIOA specifies the component to be tested and the remaining CTIOAs represent the context.

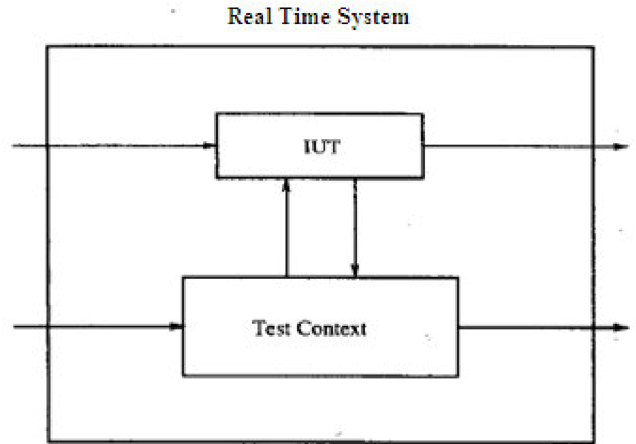

Figure 3-1 Real Time System Model for [1]

CTIOA model used to describe real-time system can be summarized as follows:
CTIOA is formally defined as 7-tuple $CT = (I_{ct},\ O_{ct},\ L_{ct},\ l_{ct}^{o},\ C_{ct},\ T_{ct},\ F_{ct})$ and

- I_{ct} is a finite set of input actions (each input begins with "?" symbol)
- O_{ct} is a finite set of output actions (each output begins with "!" symbol)
- L_{ct} is a finite set of locations (states)
- $l_{ct}^{o} \in L_{ct}$ is the initial location
- C_{ct} is a finite set of clocks that are initialized to zero in l_{ct}^{o}
- T_{ct} is a finite set of transitions and each transition consists of 5-tuple $(l_i, \{?\,,!\}a, R, G, l_j)$. Each transition is depicted as $l_i \xrightarrow{\{?,!\}a,R,G} l_j$ where l_i is the source location and l_j is the target location, $\{?\,,!\}a$ is either input or output action, R is the set of clocks to be reset with the transition and G is the guard condition (time constraint) of transition which is defined by the clocks of C_{ct}.
- F_{ct} is a FIFO channel.

An example real-time system which is modeled with three CTIOAs can be seen in Figure 3.2.

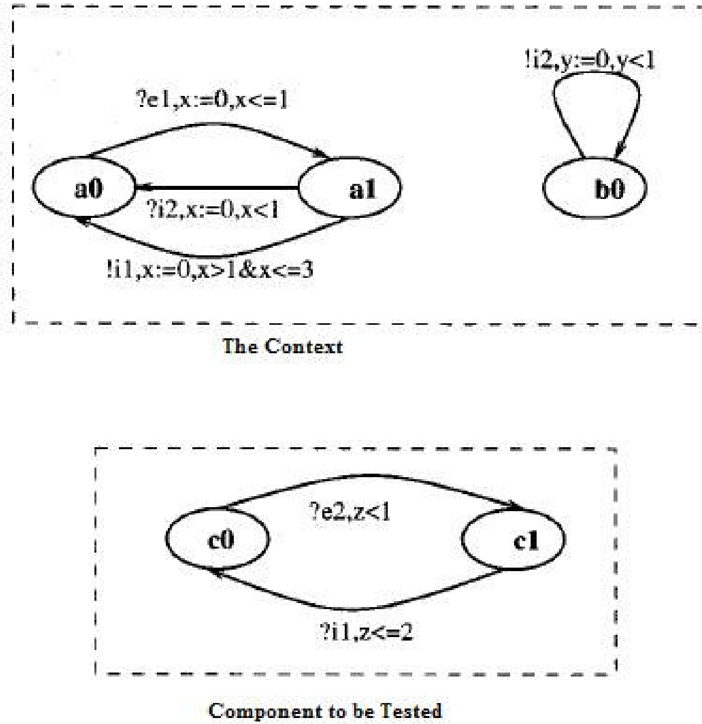

Figure 3-2 CTIOA Model of Real-Time System

It is proposed to test the IUT in context not in isolation. Testing in isolation mean testing a component alone on the other hand testing in context means testing a component with other components it is associated. Testing in context requires components of the context to be tested in isolation and verified before used in tests. Testing in context requires the effects of context to be considered. If the context is faulty, some faults which can be found in isolated test cannot be found in context test. Some transitions may not be executed since the context may not be capable of producing the action of the transition in desired time duration. It is proposed to generate test cases in three steps.

- To overcome state explosion problem that is caused by large system size and can be defined as a system to have too much states to be managed, it is proposed to avoid composing all machines of context and produce system's complete product, which is combination of all automata models into single automata model, but to select context transitions that affect the execution of the IUT. Since the context need not to be tested it is feasible to tailor context specifications.
- A collection a CTIOAs describes a real time system and partial product of these CTIOAs describes all possible executions of real time system. Partial product of the complete system is constructed using IUT CTIOA and context CTIOA that is generated by selecting transitions that effect the execution of IUT. The resulting partial product is a Timed Input Output Automata (TIOA).
- Timed WP-method [4] is used to generate test cases from partial product TIOA.

Quality of the partial products is important for fault coverage. Criteria used for selection of context actions determine the quality of the partial product and fault coverage. This method is proposed for real time systems and it suggests a solution for test case generation. Main drawback of this method is testing in context approach. Testing in context is unreliable if the context is not tested and verified.

In [12] a white box testing approach is proposed for testing real time systems. A test case is proposed to be not only the input data but a predetermined sequence of transition firings. Real-time system is modeled using time Petri nets approach. A Petri net is a graph as seen in Figure 3-3 containing places and transitions, connected by oriented arcs and a set of tokens [12].

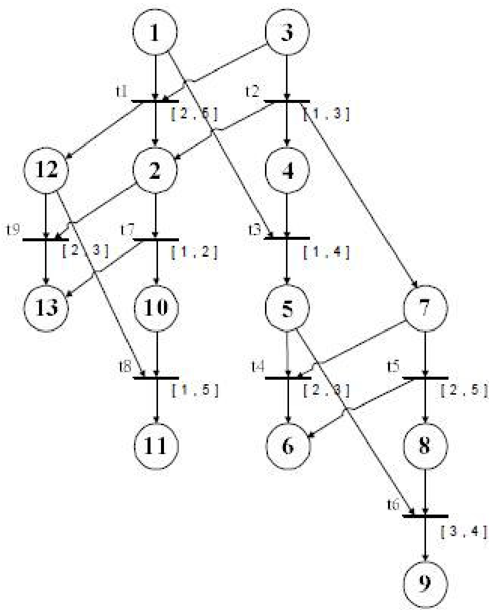

Figure 3-3 Time Petri Net [12]

Time Petri nets support expressing state transition and timing properties. Using a set of rules to identify the graph segments a graph based model is constructed using time Petri nets approach. Relationships of the graph model are converted into a graph matrix which carries the relations between states and transitions. Graph matrix is used to construct formal form of the system. An algorithm is used to generate test cases from the formal form of the system. This method only solves the test case generation problem but it does not propose a solution for test execution.

In [10] a test method is proposed for converting the nondeterministic distributed real-time systems testing problem into a deterministic sequential programs testing problems. With the same input to the tasks and the same execution ordering of the tasks, a system produces the same output for repeated executions. The system is modeled as a distributed real time system consisting of concurrent tasks. Tasks are distributed over the nodes and more than one task execute on each node of the

distributed real time system. At each run orderings of these concurrent tasks can be different. Using the predefined system and task model a method is proposed for deterministic integration testing of distributed real time systems. Testing of single tasks is not considered in this study. It is mainly focused on integration test of distributed real time system tasks. According to this method test strategy is listed as;

1. All the possible orderings of task starts, preemptions and completions of tasks are determined by the proposed execution order analysis on each node of the distributed real time system.
2. The system is tested using any of the existing sequential software test technique.
3. According to observation each test case and output is mapped onto correct execution ordering.
4. Repeat 2-3 until required coverage is achieved.

In the first phase analysis of the software is performed by using an analysis tool. All possible execution order is determined and an Execution Order Graph (EOG) [11] is constructed. EOG displays the non deterministic behavior of the real-time software. In the second phase the system is tested using any technique of choice. At each test case execution ordering is observed and recorded. Testing tools for sequential programs can be used at this phase. Each test case and output is mapped on to the observed correct execution ordering which corresponds to a branch in the EOG. Test coverage is directly related to percentage of the traced branches in the EOG. Until the required coverage for EOG is reached, steps 2 and 3 are executed. Deterministic and reproducible testing of distributed real time software is achieved with the proposed technique.

3.2. Current Approach Used in a Company for Testing Distributed Real-Time Systems

A distributed real-time system is generally a part of a system and it is not tested alone for the projects held in a specific software development company. In the current practice in the company, overall system is tested for a final system validation that serves as system test. The objective of the System Test is to check that the whole system functions according to specification. It is not possible to test the distributed real-time system separate from the rest of the system. Inputs are applied to overall system and outputs of overall system are analyzed against expected outputs. System tests were performed for testing functional (value) correctness of IUT and temporal behavior of IUT could not be tested.

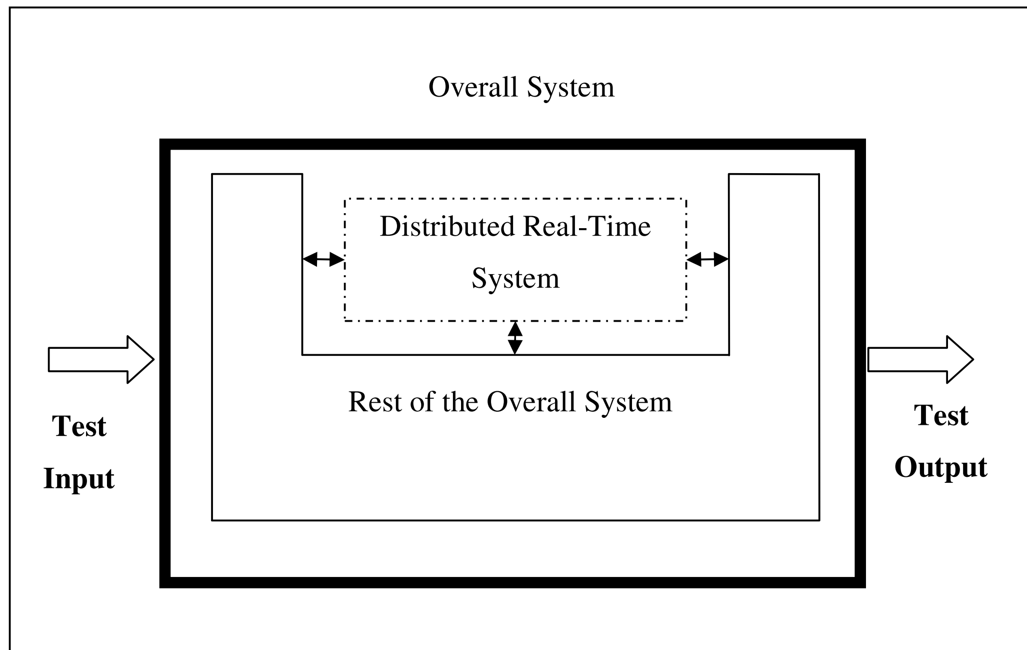

Figure 3-4 Test Approach Used In the Company

Main drawbacks of this test method are

- It is not possible to test real time constraints of Distributed real-time system

- It is not possible to test order constraints of Distributed real-time system
- Test repeatability is not possible for Distributed real-time system since we cannot control the inputs of Distributed real-time system.
- It is not possible to control inputs of Distributed real-time system.
- It is not possible to observe outputs of Distributed real-time system.

In [6] a distributed test architecture is proposed for distributed real time systems. Main contribution of this study to literature is that it considers both real time and distributed aspects into same study. A complete test strategy is proposed for testing of distributed real time systems. A model for specifying distributed real-time systems and a practical test architecture for executing tests is proposed. In the next section testing distributed real time systems with distributed test architecture will be studied.

CHAPTER 4

TESTING DISTRIBUTED REAL TIME SYSTEMS WITH DISTRIBUTED TEST ARCHITECTURE

Khoumsi proposed a first distributed test approach for testing of distributed real time systems. In [6] a test method using a distributed test architecture is proposed for testing of distributed real-time systems. It is assumed that distributed IUT contains several *sites* which are also called *nodes* and each site can communicate with its environment through a *port*. The term port also denotes each connection of distributed real time system with the rest of the system. Each site is a self sufficient computing element with CPU, memory, network access and I/O. Each port that is source for inputs and destination for outputs consists of an input queue and an output queue [17]. A *Test System* (TS) is proposed such that for each port of the distributed IUT there exist a local tester that runs on hardware apart from IUT. Each tester communicates with the IUT through the port of corresponding site and each tester communicates with the other testers. Communication medium of testers is reliable and independent of the IUT. Test System architecture is depicted in Figure 4-1.

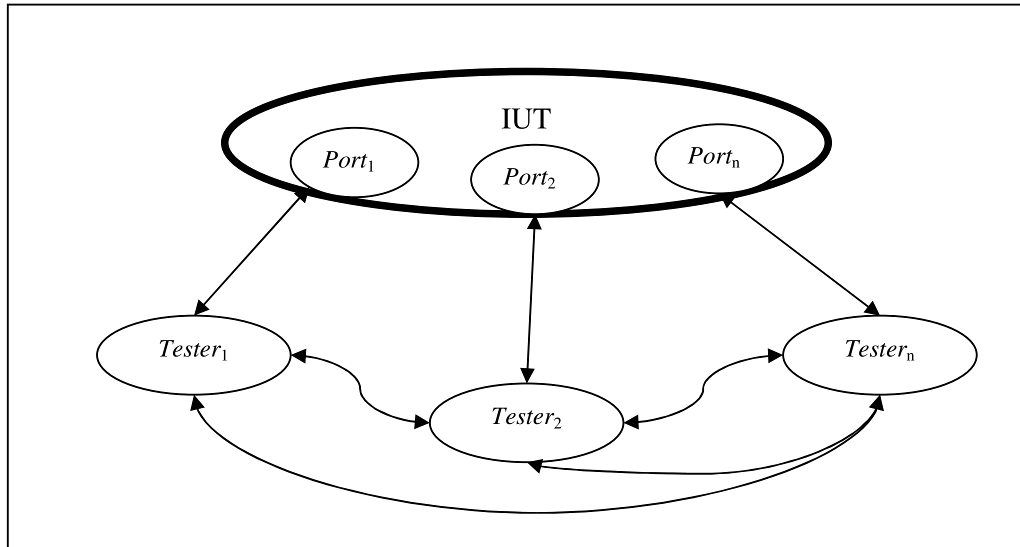

Figure 4-1 Distributed Test Architecture

Assume that we have a distributed real time system which is a part of another system and it has two nodes that communicate with the rest of the system as seen in the Figure 4-2.

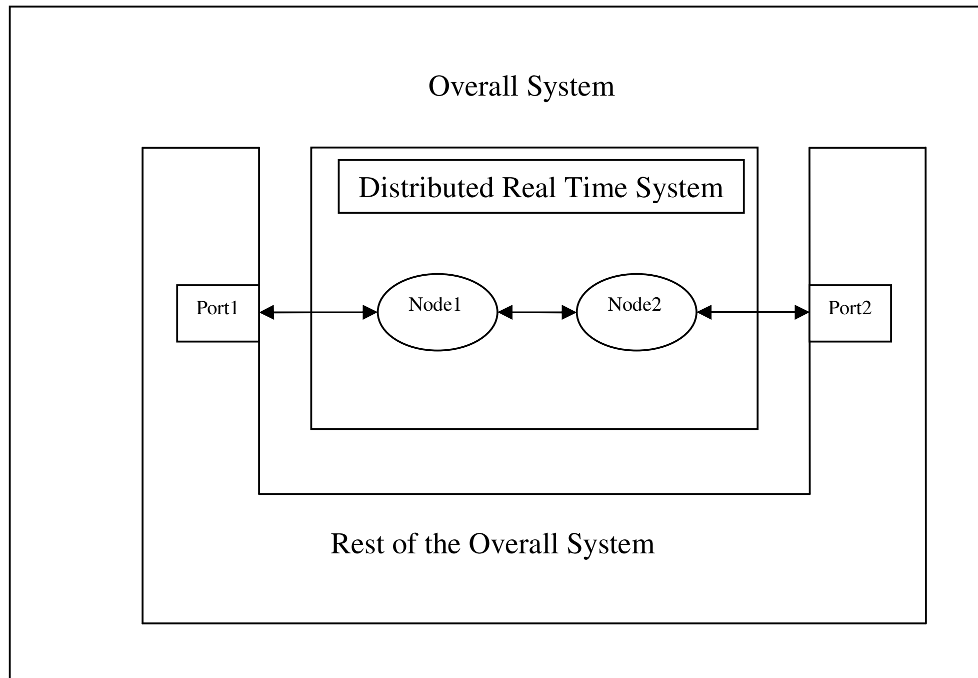

Figure 4-2 Distributed Real Time System with 2 Nodes

Each node of the real time system may have connection with other nodes. Each node may have connection with the rest of the system on a reliable communication medium. This reliable communication medium can be any kind of communication medium but it must have no message loss. For each port of the distributed real time system TS must have a corresponding *local tester*. For the distributed real time system given in Figure 4-2 TS must have 2 local testers.

The remaining of this chapter is structured as follows. Sect. 4.1 describes the model of *timed automata* with N ports that is used to model distributed real-time system. Sect 4.2 describes the fault model considered and sect 4.3 introduces the distributed test architecture. Sect 4.4 describes global test sequence that is used as a test case. Sect 4.5 describes conformance of IUT and sect 4.5 describes test execution procedure for local testers. Sect 4.7 describes coordination of local tester and finally

sect 7.8 describes the algorithm for distributing global test sequence on to local test sequences.

4.1. Timed Automata with N Ports

Timed Automata with N ports is generalized from Timed Automata [5]. A set of clocks and Canonical Enabling Conditions are used to model the temporal behavior of the real time system. In [6] Timed Automata with N ports is proposed to be used to model the temporal behavior of the distributed real time system. Definitions related to Timed Automata with N ports is given as follows;

Definiton 4.1.1. $\boldsymbol{C} = \{\boldsymbol{c_1}, \dots, \boldsymbol{c_{N_c}}\}$ is a set of clocks and each clock can be viewed as a continuous time clock. Continuous time is a real variable that evolves indefinitely and its derivative with respect to time is equal to 1. Each clock's value can be reset at any instant, e.g., with the occurrence of an event.

Definiton 4.1.2. A canonical Enabling Condition (EC) is either constant *True* or a formula in the form "$\boldsymbol{c_i} \sim \boldsymbol{k}$", where $\sim \in \{<, >, \leq, \geq, =\}$ and k is an integer. EC can be a single canonical EC or conjunction of canonical ECs. $\mathcal{E}\boldsymbol{Cc}$ is the set of ECs depending of clocks on $\boldsymbol{C}$, and $\boldsymbol{\mathcal{P}c}$ as being the set of subsets of $\boldsymbol{C}$. It is considered that $\boldsymbol{True} \in \mathcal{E}\boldsymbol{Cc}$.

Definiton 4.1.3. *Timed Automaton with n ports* named as np-TA is defined by $(\mathbf{L}, \mathbf{I}, \mathbf{O}, \mathbf{C}, \mathbf{T}, \boldsymbol{l_0})$. $\mathbf{L}$ is a finite set of locations, $\boldsymbol{l_0}$ is the initial location and $\mathbf{C}$ is a finite set of clocks. $\mathbf{I}$ is an *n-tuple* $(\mathbf{I_1}, \mathbf{I_2}, \dots \mathbf{I_n})$ where $\mathbf{I_i}$ is a finite set of inputs of port $\mathbf{i}$, $\mathbf{I_i} \cap \mathbf{I_j} = \emptyset$ for $\mathbf{i} \neq \mathbf{j}$ and $\mathbf{i}, \mathbf{j} = \mathbf{1}, \dots, \mathbf{n}$ and $\mathbf{I} = \mathbf{I_1} \cup \mathbf{I_2} \cup \dots \cup \mathbf{I_n}$. O is an n-tuple $(\mathbf{O_1}, \mathbf{O_2} \dots \mathbf{O_n})$ where $\mathbf{O_i}$ is a finite set of outputs of port $\mathbf{i}$, $\mathbf{O_i} \cap \mathbf{O_j} = \emptyset$ for $\mathbf{i} \neq \mathbf{j}$ and $\mathbf{i}, \mathbf{j} = \mathbf{1}, \dots, \mathbf{n}$ and $\mathbf{O} = \mathbf{O_1} \cup \mathbf{O_2} \cup \dots \cup \mathbf{O_n}$. $\mathbf{T} \subseteq \mathbf{L} \times (\mathbf{I} \cup \mathbf{O}) \times \mathbf{L} \times \mathcal{E}\mathbf{Cc} \times \boldsymbol{\mathcal{P}}\mathbf{c}$ is a transition relation.

Definiton 4.1.4. A transition is defined by $Tr = (q;\ \sigma; r; EC; Z)$. q and r are origin and destination locations, σ is the reception of an input a in a port i (figured as $?a^i$) or sending of an output x in a port j (figured as $!x^j$). Tr is

executed if and only if $EC = true$ and the clocks in Z are reset after execution of Tr. Z is called *reset* of Tr.

By using np-TA temporal behavior of the system can be modeled and np-TA allows to model constraints on delays between events of the system. To specify that delay between two transitions Tr_1 and Tr_2 must be between the range of [1,2], a clock c_1 is introduced. Reset of Tr_1 is defined as $\{c_1\}$ and EC of Tr_2 is defined as $(c_1 \geq 1) \wedge (c_1 \leq 2)$ to model delay constraints between two transition.

A transition is called input transition if σ is an input and a transition is called output transition if σ is an output. Input and output terms are used in the IUT viewpoint; outputs are sent by the IUT and inputs are received by the IUT. For the rest of this thesis input and output terms will be used from the IUT viewpoint.

Example 1 2p-TA model of distributed real time system given in Figure 4-2 is given in Figure 4-3. Distributed real-time system is modeled as $I_1 = \{?a^1, ?x^1\}$, $I_2 = \{\emptyset\}$, $O_1 = \{!y^1\}$, $O_2 = \{!b^2\}$ and $L = \{l_0, l_1, l_2\}$. In the graph absence of clocks to reset and EC being *True* are implied by "–".

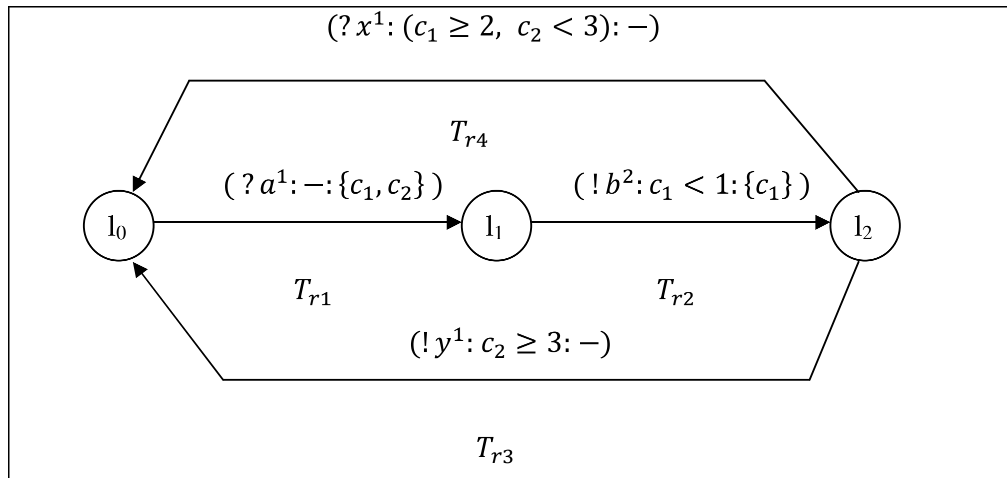

Figure 4-3 2p-Timed Automata

- The system is initially in location l_0 and with the reception of a in port 1 Tr_1 executes. c_1 and c_2 are reset and the system propagates to location l_1.
- In location l_1 Tr_2 executes, output b is sent on port 2, $c_1$is reset and system propagates to l_2. Since $EC = \{c_1 < 1\}$ the delay between $?a^1$ and $!b^2$ must be smaller than 1.
- In location l_2 the system executes either Tr_4 with the reception of x on port 1 or Tr_3 by sending *y* on port 1. The delay separating $!y^1$ and $?a^1$ for Tr_3 must be equal or greater than 3. For Tr_4 delay separating $?x^1$ and $!b^2$ must be equal or greater than 2 and delay separating $?x^1$ and $?a^1$ must be smaller than 3.

4.2. Fault Model

The fault model describes the effects of failures in a system implementation [2]. Fault model depends on the formal model used a basis for the system specification. For Timed Automata with n-ports faults are classified in two groups: (1) faults independent of timing constraints (2) timing faults [3]. First group of faults include output faults, transfer faults and hybrid faults. Second group of faults are caused by the non-respect by the IUT of timing constraints associated to outputs. During a software testing process and for a determined test sequence, the test system respects timing constraints of inputs and checks whether timing constraints of outputs are respected. The system is faulty if timing constraints are not respected during a testing process.

For the 2p-TA given in Figure 4-3, test system respects timing constraints of Tr_4 and checks whetherTr_1, Tr_2 and Tr_3 respected the timing constraints. For Tr_4 test system sends *x* on port 1 at time instant when delay separating $?x^1$ and $!b^2$ is equal or greater than 2 and delay separating $?x^1$ and $?a^1$ is smaller than 3. For Tr_2 test system checks whether the delay separating $?a^1$ and $!b^2$ is smaller than 1. The 2p-TA is faulty if ECs of Tr_1, Tr_2 and Tr_3 are not respected during a testing process.

4.3. Test Architecture

Distributed real-time system is assumed to have several sites and each site has an associated port. Proposed test architecture by [6] consists a local tester for each port of the IUT. Each local $Tester^p$ communicates with the IUT, other testers and local clock as seen in Figure 4-4. Test architecture proposed in [6] can be summarized as follows;

- $Tester^p$ communicates with IUT through port p to send inputs to IUT and receive outputs from IUT.
- $Tester^p$ communicates with other testers through a communication channel which is different from the IUT communication channel. Messages are exchanged between testers to guarantee order and timing constraints of inputs and check order and timing constraints of outputs.
- $Tester^p$ communicates with its local clock $Clock^{\mathrm{p}}$ to get current time. When $Tester^p$ receives an output from the IUT it immediately gets current time from the local clock and using this timing information and checks whether the timing constraints associated with that output is respected. $Tester^p$ asks its local clock $Clock^{\mathrm{p}}$ to be notified when current time reaches a specified value τ. This notification from the local clock is used to guarantee timing constraints of the input.

The following figure presents the test architecture for 2p-Timed Automata system given in Figure 4-2.

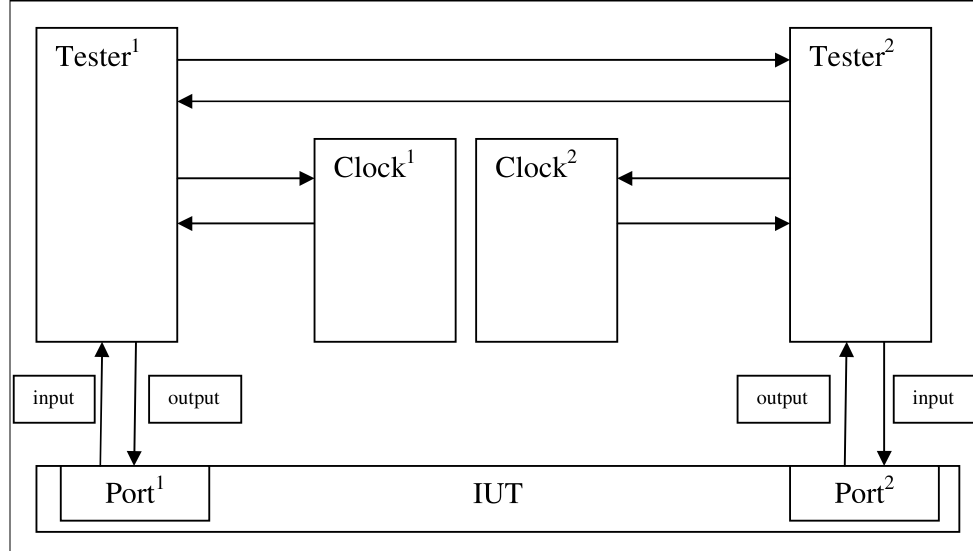

Figure 4-4 Distributed Test Architecture for 2p-Timed Automata

4.4. Defining Test Sequence

Opposed to other testing techniques discussed in 3.1, test case generation is straightforward for this technique. A global test sequence (GTS) is the test case for this method and test system has to check whether the IUT conforms to this GTS.

Definiton 4.4.1. A GTS is a sequence of transitions of the np-TA IUT for which the initial and the final locations of the transition are removed and symbolized as tr_i. Since the initial location is known there is no need to emphasize initial and final locations in the GTS.

For the 2p-TA given in Figure 4-3 $(?a^1: -: \{c_1, c_2\})$. $(!b^2: c_1 < 1: \{c_1\})$. $(?x^1: (c_1 \geq 2,\ c_2 < 3): -)$ can be considered as a GTS. This GTS $tr_1.tr_2.tr_4$ is equal to transitions of $T_{r1}.T_{r2}.T_{r4}$ of IUT given in Figure 4-3.

There are some formal techniques for generating test sequence that we will not implement within the scope of this study. [22] proposes methodical

procedure for fault detection experiments of synchronous sequential machines. This procedure includes

1. Finding an initial sequence which brings the machine into a specified state (called the starting state) regardless of the initial state of the machine
2. Defining a sequence to recognize all the states of machine
3. Defining a sequence to check all the individual transitions in the state table of the machine.

This method assumes that the considered sequential machines must be strongly connected, reduced and possessing a Distinguishing Sequence. However, this assumption does not hold for all of the systems that we will consider.

4.5. Conformance of IUT to GTS

Definiton 4.5.1. For tr_i $(ev_i,\ s_i, \tau_i)$ notation displays the basic information about tr_i. ev_i is the *event* related with that transition, s_i is the site where the event ev_i occurs and τ_i is the instant when the event ev_i occurs. For example; for tr_1 of 2p-TA of Figure 4-3 $ev_i = ?a^1$, $s_i = 1$ and τ_i is the instant when this event occurs.

Definiton 4.5.2. Consider a GTS defined as $tr_1.tr_2 \dots tr_m$, an execution of an IUT is conform with GTS if

- ev_i occurs before ev_{i+1} for $i = 1,2 \dots m-1$
- Timing constraints of tr_i is respected for $i = 1,2 \dots m$

For the 2p-TA given in Figure 4-3 execution of IUT conforms to GTS: $(\,?a^1\!:-\!:\{c_1,c_2\}\,).\,(\,!b^2\!:c_1 < 1\!:\{c_1\}\,).\,(?x^1\!:(c_1 \geq 2,\ c_2 < 3)\!:-)$, if

- $?a^1.!b^2.?x^1$ is executed with the same sequence
- Delay between $?a^1$ and $!b^2$ is < 1

- Delay between $?\,a^1$ and $?\,x^1$ is < 3
- Delay between $!\,b^2$ and $?\,x^1$ is ≥ 2

4.6. Test Execution

During testing process a GTS is applied to IUT by testers and it is checked that whether the IUT conforms to applied GTS or not. GTS contains transitions that are not on the same site. These transitions must be distributed to each local tester correctly and this will be discussed in 4.8. For each transition tr_i in the GTS there is an associated port *k* and corresponding local tester $Tester^k$. As mentioned previously there are two types of transitions: input transition and output transition. During test execution process testers executes as;

- If tr_i is an input transition $tr_i = (?\,a^k; EC; Z)$ then $Tester^k$ sends a through port k to IUT if $EC = true$ and resets the clock in Z. a is received by the IUT through port *k*.
- If tr_i is an output transition $tr_i = (!\,a^k; EC; Z)$ then IUT sends a through port k and $Tester^k$ receives it and records the reception time. $Tester^k$ checks whether $EC = true$ or not. If $EC \neq true$ then test fails and IUT does not conform to GTS.

4.7. Coordination of Testers for Solving Controllability and Observability Problem

As mentioned previously during a testing process each tester is responsible for checking whether order and time constraints are satisfied. Since testers are distributed and apart from each other, to verify that IUT conforms to GTS, testers must be coordinated. If testers are not coordinated they cannot know the transition order which yields the controllability and observability problem and GTS cannot be applied properly.

Assume a GTS $tr_1.tr_2 \dots tr_m$, then the execution of IUT conforms to GTS if the following conditions are satisfied for $i, j = 1,2 \dots m-1$ and $j \leq i$

- tr_{i+1} is executed after tr_i (Order Constraint)

- If tr_{i+1} has an EC $(c \sim x)$ and tr_j is the last transition that resets c then delay between execution times of tr_{i+1} and tr_j must satisfy the $\sim x$. Delay between execution times of tr_{i+1} and tr_j is measured by the clock c. It measures the time between two transitions.

Controllability can be defined from Test System view as capability of a Test System to force the IUT to receive inputs in the given order. Controllability problem arises when TS cannot guarantee that IUT will receive event of tr_i before event of tr_{i+1}. For distributed test architecture controllability problem arises when port of tr_i is different from port of tr_{i+1}.

Observability can be defined from Test System view as capability of a Test System to observe the outputs of the IUT and decide which input is the cause of each output. For distributed test architecture where a transition contains at most single output for each output, observability problem arises when two consecutive transitions tr_i and tr_{i+1} occurs on the same port p but only one of the transitions has an output in port p and the other one is an empty transition with no output. In this case TS cannot decide whether tr_i or tr_{i+1} is the cause of output.

To solve both controllability and observability problem of distributed test architecture, usage of order messages between local testers proposed by [6] is described in the next section.

4.7.1. Coordination Method for Order Constraints

Assume transitions tr_i and tr_{i+1} for which sites $s_i \neq s_{i+1}$ and $i = 1,2 \ldots m-1$. (For $s_i = s_{i+1}$, there is no need to consider about order constraints since both events occur at the same tester). Immediately after the execution of tr_i, $Tester^{s_i}$ should send an Order Message $O(i, \tau_i)$ to tester $Tester^{s_{i+1}}$. Here i denotes the transion ID and τ_i denotes the execution time of this transition.

- If tr_{i+1} is an input transition $Tester^{s_{i+1}}$ must wait for reception of $O(i,\tau_i)$ before sending input to IUT to respect order constraints of inputs where $Tester^{s_{i+1}}$ represents the local tester that tr_{i+1} occurs.
- If tr_{i+1} is an output transition $Tester^{s_{i+1}}$ does not wait for the reception of $O(i,\tau_i)$. After it receives output from IUT $Tester^{s_{i+1}}$ records the transition execution time τ_{i+1}. $Tester^{s_{i+1}}$ checks whether the order constraint $\tau_i < \tau_{i+1}$ is respected or not.

4.7.2. Coordination Method for Timing Constraints

Consider tr_{i+1} has an $EC\ (c{\sim}x)$ and tr_j is the last transition that resets c for $i,j = 1,2 \ldots m-1$ and $j \le i$. If tr_{i+1} is an input transition $Tester^{s_{i+1}}$ has to guarantee that delay between τ_j and τ_{i+1} satisfies $(\tau_{i+1} - \tau_j)\sim x$. If tr_{i+1} is an output transition $Tester^{s_{i+1}}$ should check that whether the delay between τ_j and τ_{i+1} satisfies $(\tau_{i+1} - \tau_j)\sim x$ or not.

To achieve this for $s_{ij} \neq s_{i+1}$, $Tester^{s_j}$ sends a Timing message $T(j,\tau_j)$ to $Tester^{s_{i+1}}$ immediately after execution of tr_j. τ_T is the instant when $Tester^{s_{i+1}}$ receives $T(j,\tau_j)$.

1. For tr_{i+1} is an input transition and $\sim\ \in \{>,\ge\}$; $Tester^{s_{i+1}}$ gets τ_j from $T(j,\tau_j)$ and chooses an instant τ_{i+1} such that $(\tau_{i+1} - \tau_j){\sim}x$ and sends input to IUT.
2. For tr_{i+1} is an input transition and $\sim\ \in \{=\}$;
   - if $(\tau_T - \tau_j) > x$ $Tester^{s_{i+1}}$ can not guarantee $(\tau_{i+1} - \tau_j){\sim}x$. $Tester^{s_{i+1}}$ misses the deadline for τ_{i+1}.
   - if $(\tau_T - \tau_j) \le x$ $Tester^{s_{i+1}}$ gets τ_j from $T(j,\tau_j)$ and chooses an instant τ_{i+1} such that $(\tau_{i+1} - \tau_j){\sim}x$ and sends input to IUT.
3. For tr_{i+1} is an input transition and $\sim\ \in \{<,\le\}$;
   - if $!((\tau_T - \tau_j){\sim}x)$ $Tester^{s_{i+1}}$ can not guarantee $(\tau_{i+1} - \tau_j){\sim}x$. $Tester^{s_{i+1}}$ misses the deadline for τ_{i+1}.
   - if $(\tau_T - \tau_j){\sim}x$ $Tester^{s_{i+1}}$ gets τ_j from $T(j,\tau_j)$ and chooses an instant τ_{i+1} such that $(\tau_{i+1} - \tau_j){\sim}x$ and sends input to IUT.

4. For tr_{i+1} is an output transition; $Tester^{s_{i+1}}$ knows τ_j after it receives $T(j,\tau_j)$ from $Tester^{s_j}$and τ_{i+1} after it receives output from IUT. $Tester^{s_{i+1}}$ checks whether $(\tau_{i+1} - \tau_j) \sim x$ is respected or not.

If $s_{ij} = s_{i+1}$ then there is no need to send $T(j,\tau_j)$.

For the IUT given in Figure 4-3 and distributed test architecture given in Figure 4-4 required coordination messages between local testers *Tester*^{1} and *Tester*^{2} for GTS: $(\,?\,a^1\!:-\!:\{c_1,c_2\}\,)\,.\,(\,!\,b^2\!:c_1 < 1\!:\{c_1\}\,)\,.\,(?\,x^1\!:(c_1 \geq 2,\ c_2 < 3)\!:-)\ (\,tr_1.tr_2.tr_4\,)$ can be listed as ;

- Since tr_1 executes on *Port*^{1} and tr_2 executes on *Port*^{2}, *Tester*^{1} should send an Order Message $O(1,\tau_1)$ to *Tester*^{2} immediately after execution of tr_1. Since tr_2 is an output transition *Tester*^{2} should not wait for the reception of $O(1,\tau_1)$.
- Since tr_2 executes on *Port*^{2} and tr_4 executes on *Port*^{1}, *Tester*^{2} should send an Order Message $O(2,\tau_2)$ to *Tester*^{1} immediately after execution of tr_2. Since tr_4 is an input transition *Tester*^{1} should wait for the reception of $O(2,\tau_2)$ to guarantee order constraints of input.
- For tr_2 enabling condition uses c_1 and tr_1 is the last transition that resets c_1. *Tester*^{1} should send a Timing Message $T(1,\tau_1)$ to *Tester*^{2}. Since tr_2 is an output transition *Tester*^{2} should not wait for the reception of $T(1,\tau_1)$.
- For tr_4 enabling condition uses c_1 and tr_2 is the last transition that resets c_1. *Tester*^{2} should send a Timing Message $T(2,\tau_2)$ to *Tester*^{1}. Since tr_4 is an input transition *Tester*^{1} should wait for the reception of $T(2,\tau_2)$. For tr_4 enabling condition uses c_2 and tr_1 is the last transition that resets c_2. Since tr_4 and tr_1 are on the same port *Port*^{1}, there is no need to send timing message. *Tester*^{1} should choose an instant τ_4 such that τ_4 must satisfy both of the following requirements;
    - ✓ Since EC of tr_4 for $c_1 \in \{>,\geq\}$, *Tester*^{1} should choose an instant τ_4 such that $\tau_4 - \tau_2 \geq 2$ to guarantee delay between tr_4 and tr_2.
    - ✓ To guarantee EC of tr_4 for c_2 *Tester*^{1} should choose an instant τ_4 such that $\tau_4 - \tau_1 < 3$.

4.8. Algorithm for Distributing Global Test Sequence on to Local Testers

An algorithm is introduced in [6] to determine local test sequence (LTS) of local testers from a given GTS. Coordination methods mentioned in 4.7.1 and 4.7.2 are used in this algorithm to coordinate testers. Each tester must execute its local test sequence correctly for overall test system to execute GTS correctly.

Definiton 4.8.1. N_s is the number of testers in the test system and also the number of ports on the IUT.

Definiton 4.8.2. $+X_{p \to q}$ stands for X sent by *Tester*^{p} is received by *Tester*^{q}

Definiton 4.8.3. $-X_{p \to q}$ stands for X is sent from *Tester*^{p} to *Tester*^{q}

Definiton 4.8.4. For a is an any kind of event and τ is a time instant

- $a_{(=\tau)}$ means a occurs at instant τ
- $a_{(<\tau)}$ means a occurs at an instant $< \tau$
- $a_{(\leq\tau)}$ means a occurs at an instant $\leq \tau$
- $a_{(>\tau)}$ means a occurs at an instant $> \tau$
- $a_{(\geq\tau)}$ means a occurs at an instant $\geq \tau$

By using these notations time constraints of an event can be displayed. For $i = 1, \dots k$, $\sim_i \in \{<, >, \leq, \geq, =\}$ and τ_i as an instant; $a_{(\sim_i \tau_i)}$ denotes all time constraints of transition for $i = 1, \dots k$.

Definiton 4.8.5. For events a and e,

- $a_{(>e)}$ means a occurs after e
- $a \mid e$ means a occurs immediately after e
- $a \parallel e$ means a and e occur independent of each other.

Definiton 4.8.6. S is a sequence of events where each event can have different sites. *Projection* of S on to $Site\ p$ means removing all other events from S occurring on other sites. Projection of S contains transitions which occur on $Site\ p$.

Definiton 4.8.7. $LocRec^p$ is a specification for $Tester^p$ which specifies coordination messages that must be received by $Tester^p$. Coordination messages in the $LocRec^p$ are independent of each other and there is no time constraint between them. They can be received by $Tester^p$ in any order. An execution of $Tester^p$ conforms with $LocRec^p$ if all of the coordination messages are received by $Tester^p$ at the end of execution.

Definiton 4.8.8. $LocSeq^p$ is in the form of $A_1.A_2 \ldots A_n$ and each A_i is in the form of $a_{(\sim_1 u_1)(\sim_1 u_1)\ldots(\sim_k u_k)} \mid v_1 \mid v_2 \mid \ldots \mid v_m$. Each a is an event either output sent by IUT to $Tester^p$ or input sent by $Tester^p$ to IUT through port p. Each event a has two kinds of constraints as discussed in 4.7: order constraints and timing constraints. u_i is either an instant or coordination message reception. If u_i is the reception of a coordination message then $\sim_i$ is $>$ which corresponds to order constraint. If u_i is an instant $\sim_i \in \{<, >, \leq, \geq, =\}$ which corresponds to timing constraint of event a. v_i is the sending of coordination message immediately after the execution of a. $LocSeq^p = A_1.A_2 \ldots A_n$ represents execution of $A_1, A_2, \ldots A_n$ in the same order.

Execution of A_i denotes,

1. Execution of event a with the constraints defined by $(\sim_1 u_1)(\sim_2 u_2) \ldots (\sim_k u_k)$. There are two type of constraints: If u_i is the reception of a coordination message then $\sim_i$ is $>$ which corresponds to order constraint and If u_i is an instant $\sim_i \in \{<, >, \leq, \geq, =\}$ which corresponds to timing constraint of event a
2. v_i executes immediately after a. Coordination messages are immediately sent after execution of a. Coordination messages can include both order messages and timing messages.

The following figure presents the *"Distribution of a GTS into N_s LTS"* algorithm given in [6] in the form of a pseudo code first and the detailed explanation of the steps in the pseudo code follows next.

"Distribution of a GTS into N_s LTS" Algorithm Pseudocode
Input: $GTS = tr_1.tr_2 \ldots$ **Outputs:** $LocRec^1, LocRec^2 \ldots LocRec^{N_s}$ $LocSeq^1, LocSeq^2 \ldots LocSeq^{N_s}$
1. $CoordinationMessages = \epsilon$; ($CoordinationMessages$ is intially empty) **2.** $GTS' = tr_1.tr_2 \ldots tr_m$ is derived from GTS by removing last transition if last transition is empty transition. **3.** $\boldsymbol{FOR}\ i = 1, \ldots m:\ act_i = ev_i\ \boldsymbol{END\ FOR}$ **4.** $\boldsymbol{FOR}\ i = 1, \ldots m - 1$ **5.** $\boldsymbol{If}\ (s_i \neq s_{i+1})$ **6.** $act_i = act_i \| - O(i, \tau_i)_{s_i \to s_{i+1}}$ **7.** $\boldsymbol{If}\ ev_{i+1}$ is an input event **8.** $act_{i+1} = act_{i+1_{(>+O(i,\tau_i)_{s_i \to s_{i+1}})}}$ **9.** $\boldsymbol{else}$ **10.** $act_{i+1} = act_{i+1_{(>\tau_i)}}$ **11.** $\boldsymbol{endif}$ ($\boldsymbol{Line}$ **7**) **12.** $CoordinationMessages = CoordinationMessages \parallel +O(i, \tau_i)_{s_i \to s_{i+1}}$ **13.** $\boldsymbol{endif}$ ($\boldsymbol{Line}$ **5**) **14.** $\boldsymbol{for\ each}\ clock\ c\ in\ Z\ of\ tr_i$ **15.** $\boldsymbol{for\ each}\ tr_j\ (j > i)\ with\ EC(c \sim_{tr_j} k)$ $and\ tr_i\ being\ the\ last\ transition\ that\ resets\ c$ **16.** $\boldsymbol{If}\ (s_i \neq s_j)$ **17.** $act_i = act_i \| - T(i, \tau_i)_{s_i \to s_j}$ **18.** $\boldsymbol{If}\ ev_j$ is an input event **19.** $act_j = act_{j_{(>+T(i,\tau_i)_{s_i \to s_j})(\sim_{tr_j}(\tau_i + k))}}$ **20.** $\boldsymbol{else}$

```
21.                 act_j = act_j(~tr_j(τ_i+k))
22.             endif (Line 18)
23.             CoordinationMessages = CoordinationMessages ∥
                +T(i,τ_i)_{s_i→s_j}
24.         else
25.             act_j = act_j(~tr_j(τ_i+k))
26.         endif (Line 16)
27.     end foreach (Line 15)
28.   end foreach (Line 14)
29. END FOR (Line 4)
30. SEQ = act_1 act_2 ... act_m
31. FOR i = 1, ... N_s
32. LocRec^i = Projection of CoordinationMessages on to Tester^i
33. LocSeq^i = Projection of SEQ on to Tester^i
34. END FOR (Line 31)
35. if last transition of GTS is empty transition.
36.   FOR i = 1, ... N_s
37.     LocSeq^i = LocSeq^i.!ϵ
38.   END FOR (Line 36)
39. endif (Line 35)
```

Figure 4-5 2p-"Distribution of a GTS into N_s LTS" Algorithm Pseudocode

Explanation of the algorithm is as follows;

- GTS is global test sequence that will be executed by test system
- $LocRec^i$ is the record of coordination messages that must be received by each tester in the test system
- $LocSeq^i$ is the sequence of input and output actions on port *i* or the sending of coordination messages.

- $CoordinationMessages$ is a set that holds the reception of all coordination messages (Order and Timing messages) by testers in the test system independent of tester.
- **Line 1 -** Initially $CoordinationMessages$ is empty.
- **Line 2 -** $GTS' = tr_1.tr_2 \dots tr_m$ is derived from GTS by removing last transition if last transition of GTS is an empty transition. The last transition of a GTS can be an empty transition like $(!\,\varepsilon; -; -)$. For IUT to conform GTS, after correctly execution of $tr_1.tr_2 \dots tr_m$, TS must receive nothing from the IUT.
- **Line 3 -** act_i is an event list that holds sequence of input and output actions and the sending of coordination messages for tr_i. For tr_i, ev_i is set to be the first event in the event list act_i.
- **Line 5-13 -** $\boldsymbol{FOR}\ i = 1, \dots m-1$ It is checked whether subsequent transitions tr_i and tr_{i+1} occurs on different sites. If they occur on different sites, coordination problem occurs and it is solved by sending an order message from $Tester^{s_i}$ to $Tester^{s_{i+1}}$. If tr_{i+1} is an input transition then $Tester^{s_{i+1}}$ is set to wait for order message before sending input to IUT to guarantee that $\tau_i < \tau_{i+1}$. If tr_{i+1} is an output transition then $Tester^{s_{i+1}}$ is set to check whether $\tau_i < \tau_{i+1}$ after receiving output from IUT and coordination message from $Tester^{s_i}$.
- **Line 14-28 -** For each transition tr_i, clocks to be reset are processed iteratively. First transition in the global test sequence that uses clock as guard condition is determined as tr_j $(j > i)$. If tr_i and tr_j occurs on different sites timing message $T(i, \tau_i)_{s_i \to s_j}$ is sent by $Tester^{s_i}$ to $Tester^{s_j}$. If tr_j is an input transition then $Tester^{s_j}$ is set to wait both for reception of timing message $T(i, \tau_i)_{s_i \to s_j}$ and until the instant when EC $(c \sim_{tr_j} k)$ becomes $True$ before sending input to IUT. If tr_j is an output transition then $Tester^{s_j}$ is set to check whether EC $(c \sim_{tr_j} k)$ at the instant when $Tester^{s_j}$ receives output from IUT. If tr_i and tr_j occurs on same site then there is no need to use timing messages. If tr_j is an input transition then $Tester^{s_i} = Tester^{s_j}$ is set

to send input to IUT at an instant when EC ($c\sim_{tr_j}k$) becomes $True$. If tr_j is an output transition then $Tester^{s_j}$ is set to check whether EC ($c\sim_{tr_j}k$) at the instant when $Tester^{s_j}$ receives output from IUT.

- **Line 32 -** $CoordinationMessages$ is projected on to local testers and $LocRec^i$ is created for each local tester.
- **Line 33 -** SEQ is projected on to local testers and $LocSeq^i$ is created for each local tester.
- **Line 35-39 -** If last transition of GTS is empty transition then $!\varepsilon$ is added to each $LocSeq^i$. Termination of $LocSeq^i$ with $!\varepsilon$ means after execution of $LocSeq^i$, $Tester^i$ must receive nothing from the IUT for conformance to GTS. (If $LocSeq^i$ does not terminate with $!\varepsilon$, $Tester^i$ just ignores outputs send by IUT after execution of $LocSeq^i$)

For the IUT given in Figure 4-3 and distributed test architecture given in Figure 4-4 distribution of GTS into 2 local testers $Tester^1$ and $Tester^2$ for GTS: $(\,?a^1{:}-{:}\{c_1,c_2\}\,)\,.\,(\,!b^2{:}c_1<1{:}\{c_1\}\,)\,.\,(?x^1{:}(c_1\geq 2,\ c_2<3){:}-)$ $(tr_1.tr_2.tr_4)$ can be summarized as follows;

- $N_s = 2$ and there are 2 local testers since there are 2 ports of the IUT.
- GTS: $(\,?a^1{:}-{:}\{c_1,c_2\}\,)\,.\,(\,!b^2{:}c_1<1{:}\{c_1\}\,)\,.\,(?x^1{:}(c_1\geq 2,\ c_2<3){:}-)$ $(tr_1.tr_2.tr_4)$
- $LocRec^1 = (+O(2,\tau_2)_{2\to1} \parallel +T(2,\tau_2)_{2\to1})$
- $LocSeq^1 =$ $(\,?a^1 \mid -O(1,\tau_1)_{1\to2} \mid -T(1,\tau_1)_{1\to2}\,).(?x^1{}_{(>+O(2,\tau_2)_{2\to1})(<3+\tau_1)(\geq2+\tau_2)})$
- $LocRec^2 = (+O(1,\tau_1)_{1\to2} \parallel +T(1,\tau_1)_{1\to2})$
- $LocSeq^2 = (\,(\,!b^2{}_{(>\tau_1)(<1+\tau_1)}) \mid -O(2,\tau_2)_{2\to1}\,) \mid -T(2,\tau_2)_{2\to1}\,)$

CHAPTER 5

IMPLEMENTATION OF IEEE 1588 PRECISION TIME PROTOCOL ON A DISTRIBUTED TEST SYSTEM

5.1. IEEE 1588 PRECISION TIME PROTOCOL

In [6], the problem of having clocks on different sites in a distributed system is not taken into consideration. Distributed testers' clocks must be synchronized in order to check timings of outputs correctly. If clocks are not synchronized timestamps are not consistent with each other. In [6] the author does not discuss effects of unsynchronized clocks on the analysis of the timing of the outputs. For example; to check that the delay between two transitions Tr_1 and Tr_2 is between the range of [0,2], each local tester records the transition time by using its local clock. Assume that the local clocks of testers are not synchronized and $LocalClock_2 = LocalClock_1 + 3$. Suppose that Tr_1 executes at 50 for $LocalClock_1$. At the same time $LocalClock_2$ is 53. Then $LocalClock_2$ will be greater than 52 whether Tr_2 executes in required time delays and this requirement will fail although it is satisfied by the IUT.

IEEE 1588 precision time protocol is a new synchronization standard with very high accuracy that is particularly proposed for embedded industrial communication systems. In this thesis, we implement the synchronization with the timing messages similar to IEEE 1588 timing messages. Hence, rather than implementing the standard completely, we have a partial implementation. The two primary synchronization problems that must be overcome in a distributed test architecture

are *oscillator drift* and *time transfer latency* (*offset*) between testers. On different sites, oscillators may (when running free) not have exactly the same
frequency and the frequency of each site's oscillator may vary over time due to environmental conditions causing oscillator drift. Offset problem occurs if there is an offset between local clocks of different sites due to lack of global clock. Oscillator drift can be solved by using higher quality oscillators [7]. The time transfer latency (offset) problem is more difficult. IEEE 1588 Precision Time Protocol provides a means for networked computer systems to agree on a master clock reference time and a means for slave clocks to estimate their offset from master clock time. PTP solves the synchronization problem of master and slaves by precisely estimating send and receive times (time stamps) of messages exchanged between master and slaves. Using specialized hardware interfaces in the physical layer of the network increases the precision of the timestamps. Implementation of PTP without using specialized hardware is called software-only implementation. Software-only implementation of PTP introduces much more non-determinism on time stamp latencies since time stamping operation is executed on higher layers of the network. PTP can be described in brief as follows;

A. **Masters and Slaves:** The master provides the reference time for one or more slave clocks by exchanging messages over network.
B. **Sync Messages:** Sync messages are sent by the master to the slaves. Master time stamps the send time of Sync messages as t_0 and slaves time stamp the receipt time of Sync message as t_1. Difference between t_0 and t_1 is named master to slave delay d_{m2s}.
$d_{m2s} = t_1 - t_0$
C. **Delay Request Messages:** Delay Request Messages are sent by the slaves to the master. A slave time stamps the send time of Delay Request messages as t_2 and the master time stamps the receipt time of Sync message as t_3. Difference between t_2 and t_3 is named as slave to master delay d_{s2m}.
$d_{s2m} = t_3 - t_2$

D. TimeStamp Indication Message: TimeStamp Indication Messages are sent by the master to the slaves. The master sends t_0 and t_3 timestamps to the slaves for slaves to be able to calculate d_{m2s} and d_{s2m} values.

E. One-Way Delay: Message propagation delay is estimated by PTP. Master-to-slave and slave-to-master propagation delays are assumed to be symmetric. Average of master-to-slave and slave-to-master delays cancels the time offset between master and slave. Message propagation delay (d_{prop}) that is also called one-way-delay is calculated as: $d_{prop} = (d_{m2s} - d_{s2m})/2$.

F. Offset From Master: Time difference between master and slave clocks are estimated by PTP and referred as offset from master.

$Offset = d_{m2s} - d_{prop}$

Calculation of offset [7] using PTP is depicted in Figure 5-1.

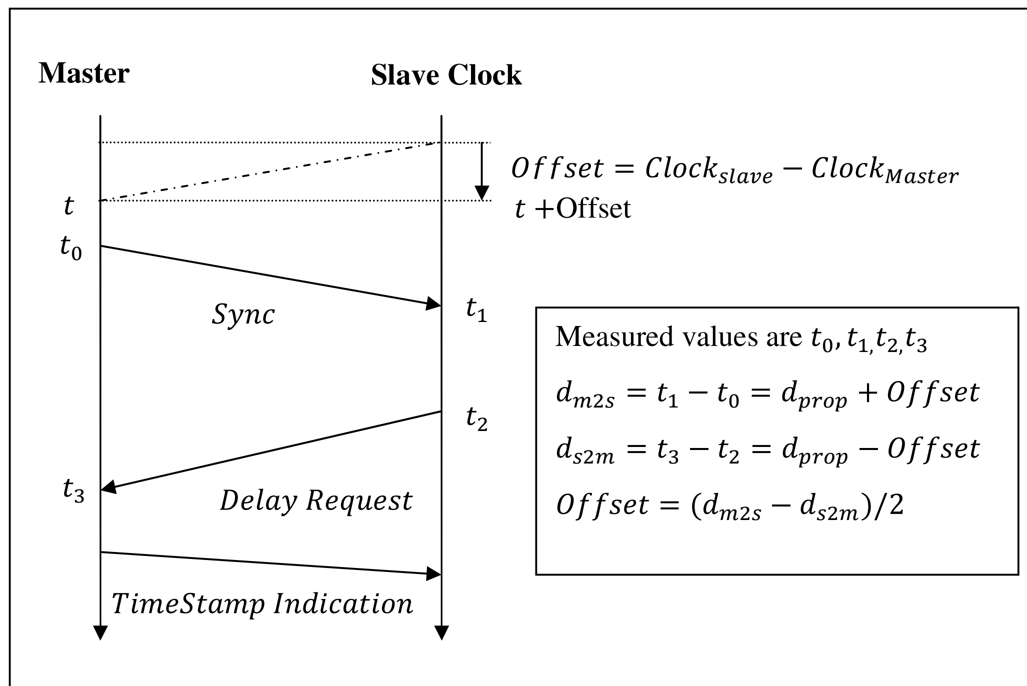

Figure 5-1 Offset Calculation with IEEE 1588 PTP

5.2. Implementation of PTP on a Distributed Test Architecture

By applying PTP on a distributed test system synchronization of local tester clocks can be achieved. In this thesis software-only implementation of PTP has been considered since it was not possible to reach hardware interfaces in the physical layer of the network and we have implemented PTP partially to solve our synchronization problem. Oscillator drift will be ignored since high quality oscillators can be used to overcome this problem.

Since there is no master slave relationship between local testers, one of the testers is chosen as master and the rest as slaves. PTP is applied between master and each slave before execution of LTSs. Each local tester adjusts its local clock using the $offset$ value that is calculated by PTP. Local testers synchronize their local clocks with the master and begin to execution of LTSs. Local Test Sequence execution with PTP implementation is depicted in Figure 5-2.

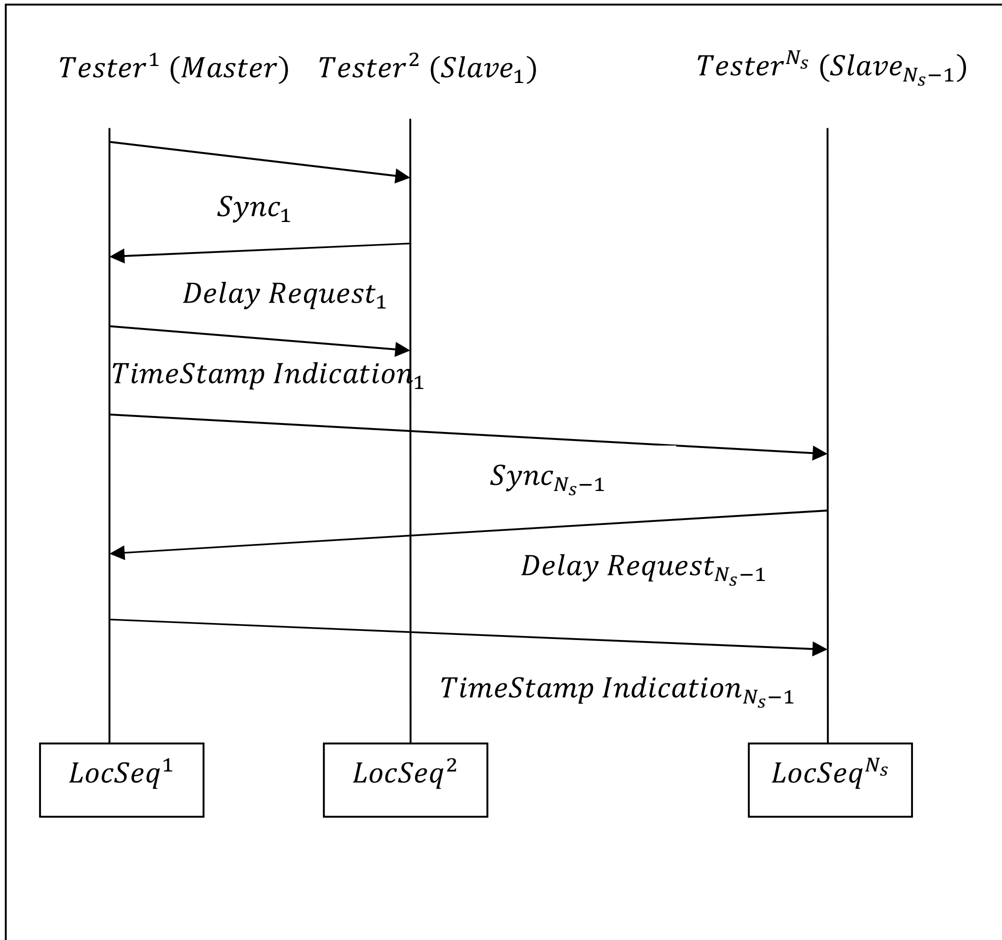

Figure 5-2 2p Local Test Sequence Execution with PTP Implementation

CHAPTER 6

IMPLEMENTATION

To the best of our knowledge, there is no implementation of testing distributed real-time systems using a distributed test approach in the literature prior to this study. Unfortunately, much of the work done in industry is not in the public domain. This study will contribute to the literature for applicability of this technique by explaining how the approach was implemented on sample projects.

In this chapter, a distributed real-time system (IUT) is introduced first that will be tested using distributed approach and nP-Timed Automata model of the IUT is presented. Then the implementation details of developed TS for testing IUT and the run time environment of the IUT and TS are explained, results of software test execution are presented and comparison with the previous tests held in the company is given to evaluate the performance of the technique.

6.1. IUT Architecture and Target System

IUT is a distributed real-time application with real time requirements that it has to implement. IUT consist of four nodes that are connected to each other. IUT runs on a target system that contains two PowerPC 7410 daughtercards and four 400 MHz MPC7410 PowerPC microprocessor computing nodes. Each PowerPC 7410 daughtercard includes two 400 MHz MPC7410 PowerPC microprocessor computing nodes. Computing nodes communicate with each other across the RACEway [19], high-speed fabric interconnect. Distributed parts of IUT runs on computing nodes and communicate with each other to perform overall IUT requirements.

6.1.1. PowerPC 7410 Daughtercard

As stated in [14] the RACE (Real-time Asynchronous Compute Environment) Series PowerPC daughtercard is the computational engine of RACE multicomputer systems. Each PowerPC 7410 daughtercard contains two compute nodes that each compute node consists of a 400 MHz MPC7410 PowerPC microprocessor, L2 cache, local SDRAM memory, and an ASIC that acts as both an advanced memory controller and the interface to the RACE switch-fabric interconnect. Each compute node on the 400-MHz PowerPC 7410 daughtercard has a dedicated fabric interface at 267 MB/s and the maximum memory speed at 133 MHz.

Mercury's RACE technology is a data link protocol that can be used to interconnect large numbers of computers, input and output interfaces, and other hardware devices into a communications fabric called a RACEway. RACE interconnect enables increased communication speed, more richly connected topologies, and augmented adaptive routing. These properties yield significantly higher bisection bandwidth and lower latency suitable for the most challenging real-time problems [19]. Figure 6-1 shows a photo of the PowerPC 7410 daughtercard on which two distinct computing nodes can be seen.

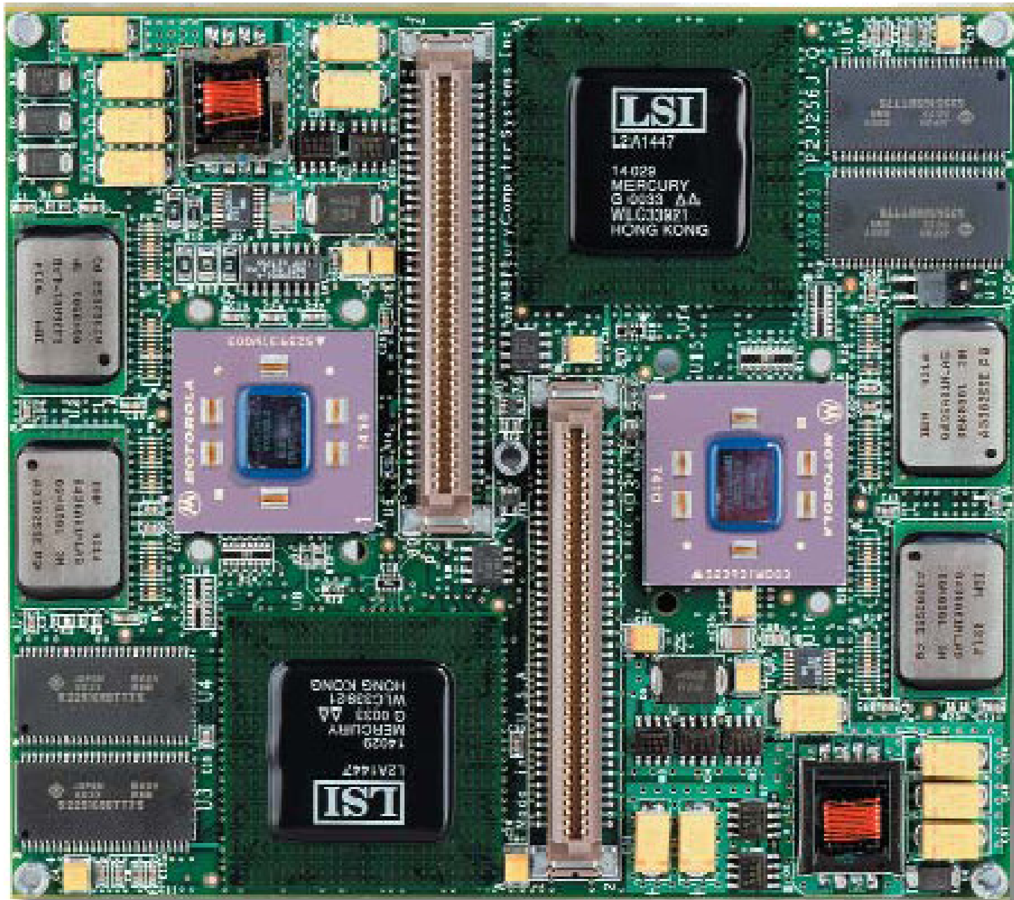

Figure 6-1 PowerPC 7410 Daughtercard

Based on the specifications of the board, it can be said that The PowerPC 7410 daughtercard is suitable for real-time applications. A detailed description of the board specifications can be found in Appendix A.

6.1.2.IUT Architecture

IUT is in brief a signal processing software that has an interface with other applications. Main functionality of IUT is getting orders and parameters from other applications, processing data accordingly and sending results to other applications. IUT has real-time requirements like processing data in a determined time and

sending results to another application. Each node communicates with other applications through a socket connection. Messages are predetermined data structures used by applications to exchange information. Messages are sent through socket connections between IUT and neighborhood applications that IUT communicates. Messages are implemented both by IUT and interfacing applications in order to exchange information correctly between them. Interfacing applications also run on 400 MHz MPC7410 PowerPC microprocessor computing nodes that are located on PowerPC 7410 daughtercards. IUT has totally four input/output connections to communicate with other applications so it can be modeled as 4p-Timed Automata. Architecture of IUT and the interaction with the neighborhood applications is depicted in Figure 6-2.

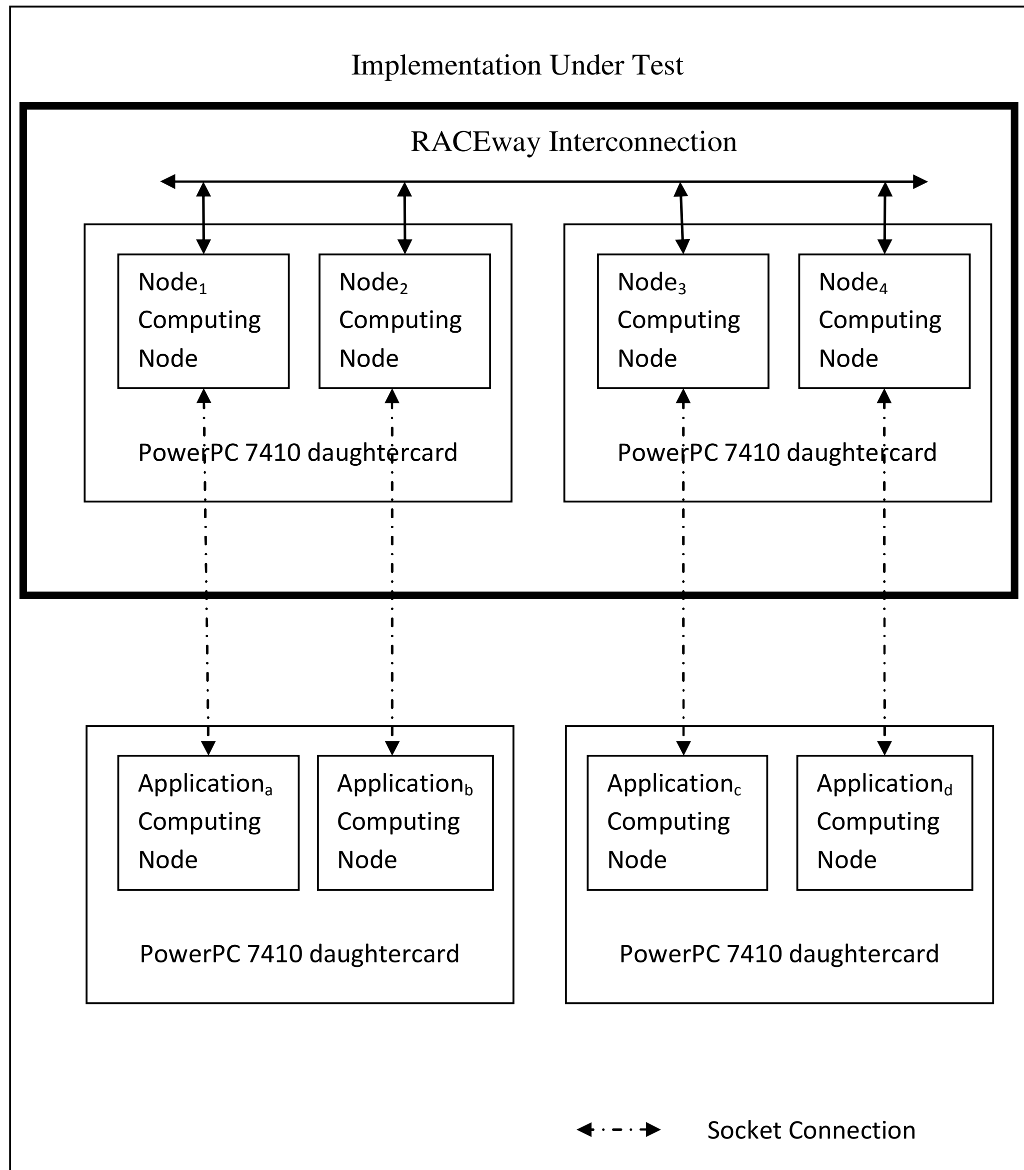

Figure 6-2 IUT Architecture

6.2. 4P-Timed Automata Model of the IUT

In this chapter, 4p-Timed Automata model of the IUT is explained as described in 4.1. Since IUT has 4 input/output interfaces it must be modeled as a four port

Timed Automata to be able to express all requirements in the model. IUT has 4 connections and each connection is with a separate application. Number of ports for modeling IUT is determined by number of input/output channels we want observe and model the system. Number of applications that IUT communicates is not important since IUT can communicate with an application through more than one channel.

While building the 4p-Timed Automata model of the IUT requirements of the IUT are used. Main points that must be taken into account are listed below

- Internal structure of the IUT must not be modeled.
- IUT must be modeled with respect to its inputs and outputs.
- Each transition must have an input or output event.
- Locations must be defined such that there must be a single event at every transition.

4p-Timed Automata model of IUT is given in Figure 6-3.

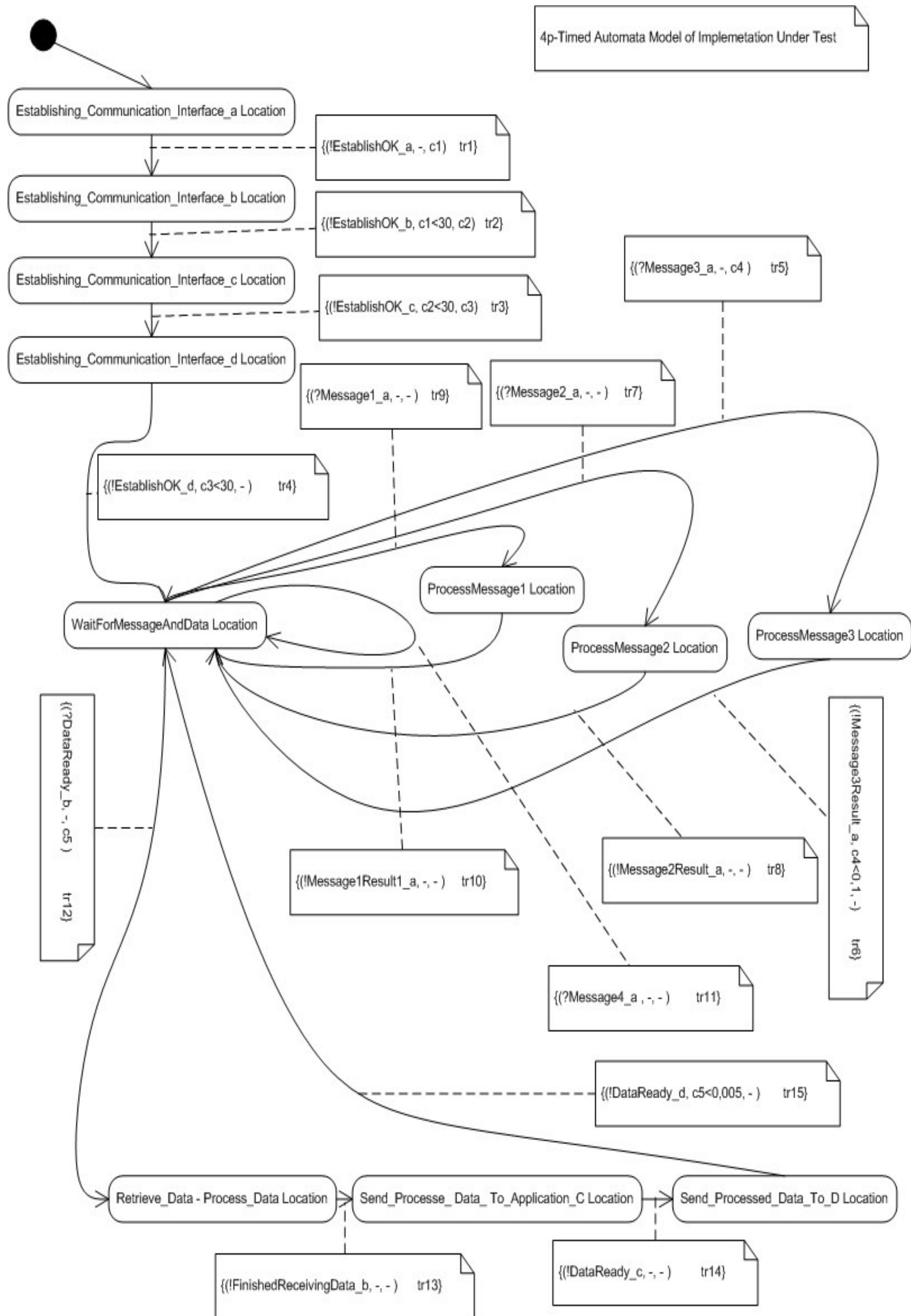

Figure 6-3 4p-Timed Automata Model of IUT

Explanation of the 4p-Timed Automata of IUT can be listed as;

- When IUT establishes communication it must send an output to each application in the order of Application_a, Application_b, Application_c and Application_d.
- Delay between communication establishments must not be larger than 30 second.
- Delay between input event $Message3^a$ send by Application_a to IUT and output event $Message3Result^a$ send by IUT to Application_a must not be greater than 0,1 second.
- Delay between input event $DataReady^b$ send by Application_b to IUT and output event $DataReady^d$ send by IUT to Application_d must not be greater than 0,005 second.

6.3. Developed Tool for Automatic Local Test Sequence Generation

It is time consuming to manually generate Local Test Sequences from a given Global Test Sequence and it also leads to faulty Local Test Sequence. A tool is developed for automatic generation of Local Test Sequences from a given Global Test Sequence.

Tool was developed using Visual Studio .NET 2003 and C# language. It implements the algorithm given in 4.8. It takes GTS transitions as input and produces Local Test Sequences as an output. Transitions are specified by;

- Event Name
- Event Site
- Event Type
- Reset Clocks
- Enabling Conditions

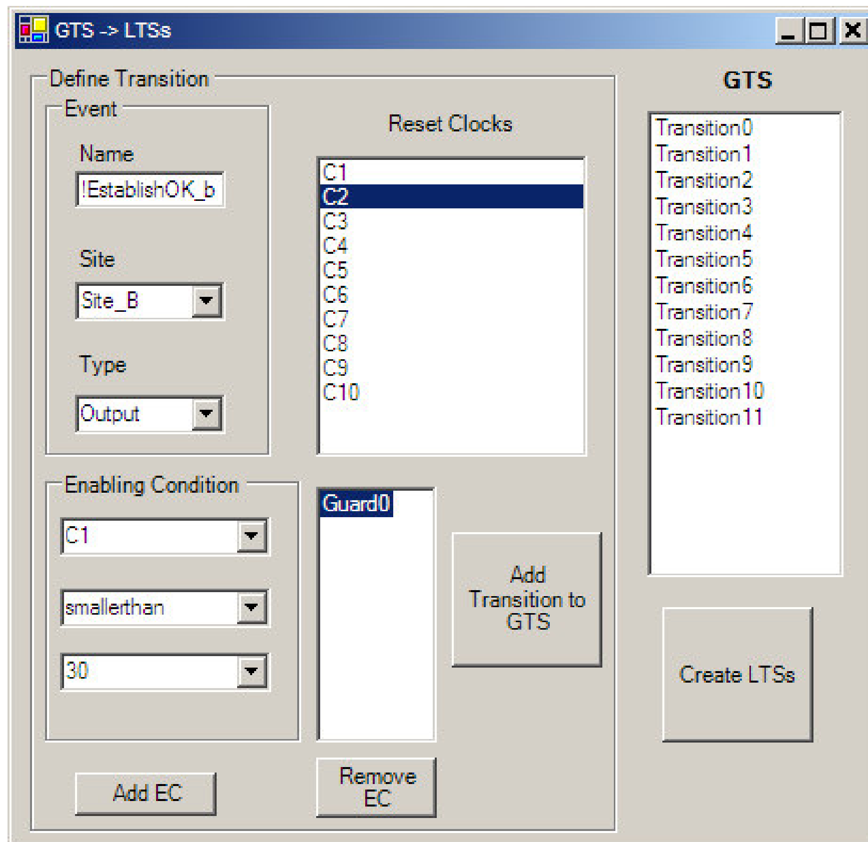

Figure 6-4 Developed Tool for LTS Generation

Figure 6-4 shows the interface of the developed tool. To generate LTSs, GTS transitions are specified using this tool. Each transition can have only one event. Event Name, Event Type (input/output) and Event Site (on which site this event occurs) fields are filled according to transition event. Enabling Conditions for each transition are defined by entering the clock name, condition type and time value. Each transition can have more than one enabling condition. After entering all transitions of GTS in the GTS order LTS are generated in a text file. Generated LTS are used when developing Local Testers.

6.4. Global Test Sequence and Local Test Sequences

As it can be seen from the figure 6.3, it is possible to construct more than one GTS. $(tr_1.tr_2.tr_3.tr_4).(tr_5.tr_6).(tr_7.tr_8).(tr_9.tr_{10}).(tr_{11}).(tr_{12}.tr_{13}.tr_{14}.tr_{15})$ shows the transitions that occur consequently. Transitions in the same bracket must occur after each other. $(tr_5.tr_6)$, $(tr_7.tr_8)$, $(tr_9.tr_{10})$, (tr_{11}), $(tr_{12}.tr_{13}.tr_{14}.tr_{15})$ can follow each other in any sequence so it is possible to generate 5!=120 different GTS. Following GTS sequences are possible,

$GTS\ 1$	$(tr_1.tr_2.tr_3.tr_4).(tr_5.tr_6).(tr_7.tr_8).(tr_9.tr_{10}).(tr_{11}).(tr_{12}.tr_{13}.tr_{14}.tr_{15})$
$GTS\ 2$	$(tr_1.tr_2.tr_3.tr_4).(tr_5.tr_6).(tr_7.tr_8).(tr_9.tr_{10}).(tr_{12}.tr_{13}.tr_{14}.tr_{15}).(tr_{11})$
$GTS\ 3$	$(tr_1.tr_2.tr_3.tr_4).(tr_5.tr_6).(tr_7.tr_8).(tr_{12}.tr_{13}.tr_{14}.tr_{15}).(tr_9.tr_{10}).(tr_{11})$
$GTS\ 4$	$(tr_1.tr_2.tr_3.tr_4).(tr_5.tr_6).(tr_{12}.tr_{13}.tr_{14}.tr_{15}).(tr_7.tr_8).(tr_9.tr_{10}).(tr_{11})$
....	

Following GTS is used to cover all transitions of 4p-Timed automata model given in Figure 6-3.

$GTS = tr_1.tr_2.tr_3.tr_4.tr_5.tr_6.tr_7.tr_8.tr_9.tr_{10}.tr_{11}.tr_{12}.tr_{13}.tr_{14}.tr_{15}$

Test system will check whether the IUT conforms to following transition sequence.

$GTS =$

$(!\,EstablishOK^a, -, c_1).(!\,EstablishOK^b, c_1 < 30, c_2).$

$(!\,EstablishOK^c, c_2 < 30, c_3).(!\,EstablishOK^d, c_3 < 30, -).$

$(?\,Message3^a, -, c_4).(!\,Message3Result^a, c_4 < 0.1\,, -).$

$(?\,Message2^a, -, -).(!\,Message2Result^a, -, -).$

$(?\,Message1^a, -, -).(!\,Message1Result^a, -, -).$

$(?\,Message4^a, -, -).(?\,DataReady^b, -, c_5).$

$(!\,FinishedReceivingData^b, -, -).(!\,DataReady^c, -, -).$

$(!\,DataReady^d, c_5 < 0.005, -)$

Test System will have 4 testers since there are 4 ports (a, b, c and d) of IUT. Local testers will be named as $Tester^a, Tester^b, Tester^c \; and \; Tester^d$. Specifications of local testers must be derived from GTS to be able to implement local testers. Local Test Sequences must guarantee that GTS is executed correctly if each tester executes correctly. LTSs of local testers are derived from GTS by using the tool introduced in 6.3 and LTS of each tester is listed below.

$LocSeq^a =$
$(\,!EstablishOK^a \mid -O(1,\tau_1)_{a\to b} \mid -T(1,\tau_1)_{a\to b}\,).$
$(?Message3^a{}_{(>+O(4,\tau_4)_{d\to a})}).(!Message3Result^a{}_{(<\tau_5+0.1)}).$
$(?Message2^a).(!Message2Result^a).(?Message1^a).(!Message1Result^a).$
$(?Message4^a \mid -O(11,\tau_{11})_{a\to b})$

$LocSeq^b =$
$(!EstablishOK^b{}_{(>\tau_1)(<\tau_1+30)} \mid -O(2,\tau_2)_{b\to c} \mid -T(2,\tau_2)_{b\to c}).$
$(?DataReady^b{}_{(>+O(11,\tau_{11})_{a\to b})} \mid -T(12,\tau_{12})_{b\to d}).$
$(!FinishedReceivingData^b \mid -O(13,\tau_{13})_{b\to c})$

$LocSeq^c =$
$(!EstablishOK^c{}_{(>\tau_2)(<\tau_2+30)} \mid -O(3,\tau_3)_{c\to d} \mid -T(3,\tau_3)_{c\to d}).$
$(!DataReady^c{}_{(>\tau_{13})} \mid -O(14,\tau_{14})_{c\to d})$

$LocSeq^d =$
$(!EstablishOK^d{}_{(>\tau_3)(<\tau_3+30)} \mid -O(4,\tau_4)_{d\to a}).$
$(!DataReady^d{}_{(>\tau_{14})(\tau_{12}+0.005)})$

6.5. Method for Analysis of Test Results

As stated previously, for IUT to conform to GTS each local tester must execute its LTS successfully. [6] does not discuss how to analyze test results. For conformance

of IUT to GTS, outputs of IUT must be analyzed. A method for analyzing test results has been adopted within this study as follows;

- Each local tester must execute its local test sequence and it must not be blocked for receiving output event from IUT eternally. Local testers are blocked for receiving output from IUT which means no other operation is performed by local testers until the expected output from IUT is received. Output event must be received in definite time duration for LTS to continue execution. If a tester is blocked eternally for receiving an output from IUT this means there is a fault in the IUT.
- After execution of LTSs, timing constraints of output events must be analyzed to see whether IUT has respected timing constraints specified by Timed Automata model.
- After execution of LTSs, order constraints of output events must be analyzed to see whether IUT has respected order constraints specified by GTS.

6.6. Implementation of Local Testers

Local testers implement both IEEE 1588 PTP and Local Test Sequences. $Tester^a$ behaves as a master, $Tester^b$, $Tester^c$ and $Tester^d$ behaves as a slave for PTP implementation. Testers wait in a blocking state for the reception of output send by IUT and record the reception time of that event. In order to respect order constraints testers also wait in a blocking state for the reception of $Order$ messages if the event related with that transition is input transition. On the other hand, Testers are not blocked for receiving $Order$ message from other testers when the event related with that transition is output transition. Order constraints related with outputs are controlled at the end of local test sequences. $Timing$ messages from other testers are not blocking, timing constraints are controlled at the end of local test sequences.

Figure 6-5, Figure 6-6, Figure 6-7 and Figure 6-8 introduces the pseudecode of each local tester. Firstly, each tester establishes communication with other testers. After establishing communication, $Tester^a$ implements PTP as a master and rest of the testers implement PTP as a slave. After implementing PTP protocol, each tester establishes communication with IUT. Each tester implements its local test sequence $LocSeq^i$. After the occurrence of each event in the LTS, each tester checks whether the order and timing constraints are satisfied.

$\boldsymbol{Tester^a}$ **Pseudocode**
1. Initializations **2.** Communication establishment with $Tester^b$ **3.** Communication establishment with $Tester^c$ **4.** Communication establishment with $Tester^d$ **5.** Send $Synch_b$ to $Tester^b$ and record send time as t_0 **6.** Wait For $Delay\ Request_b$ from $Tester^b$ and record receipt time as t_3 **7.** Send $Time\ Stamp\ Indication_b$ to $Tester^b$ with t_0 and t_3 timestamp values **8.** Send $Synch_c$ to $Tester^c$ and record send time as t_0 **9.** Wait For $Delay\ Request_c$ from $Tester^c$ and record receipt time as t_3 **10.** Send $Time\ Stamp\ Indication_c$ to $Tester^c$ with t_0 and t_3 timestamp values **11.** Send $Synch_d$ to $Tester^d$ and record send time as t_0 **12.** Wait For $Delay\ Request_d$ from $Tester^d$ and record receipt time as t_3 **13.** Send $Time\ Stamp\ Indication_d$ to $Tester^d$ with t_0 and t_3 timestamp values **14.** Communication establishment with IUT through $port\ a$ **15.** Wait for reception of $!EstablishOK^a$ from IUT and record receipt time as τ_1 (If not received for a specified time, $\boldsymbol{Tester^a_{Constraint1}}$ **FAIL**) **16.** Send $O(1, \tau_1)_{a \to b}$ to $Tester^b$ **17.** Send $T(1, \tau_1)_{a \to b}$ to $Tester^b$

18. **18.** Wait for reception of $O(4, \tau_4)_{d \to a}$ from $Tester^d$

19. Send ? $Message3^a$ to IUT and record the send time as τ_5

20. Wait for reception of ! $Message3Result^a$ from IUT and record receipt time as τ_6 (If not received for a specified time, $\boldsymbol{Tester^a_{Constraint2}}$ **FAIL**)

21. Send ? $Message2^a$ to IUT and record the send time as τ_7

22. Wait for reception of ! $Message2Result^a$ from IUT and record receipt time as τ_8 (If not received for a specified time, $\boldsymbol{Tester^a_{Constraint3}}$ **FAIL**)

23. Send ? $Message1^a$ to IUT and record the send time as τ_9

24. Wait for reception of ! $Message1Result^a$ from IUT and record receipt time as τ_{10} (If not received for a specified time, $\boldsymbol{Tester^a_{Constraint4}}$ **FAIL**)

25. Send ? $Message4^a$ to IUT and record the send time as τ_{11}

26. Send $O(11, \tau_{11})_{a \to b}$ to $Tester^b$

27. **IF** ($\tau_6 < \tau_5 + 0.1$) PASS **ELSE** $\boldsymbol{Tester^a_{Constraint5}}$ FAIL

Figure 6-5 Pseudocode of Tester[a]

$\boldsymbol{Tester^b}$ **Pseudocode**
1. Initializations **2.** Communication establishment with $Tester^a$ **3.** Communication establishment with $Tester^c$ **4.** Communication establishment with $Tester^d$ **5.** Wait For $Synch_b$ from $Tester^a$ and record receipt time as t_1 **6.** Send $Delay\ Request_b$ to $Tester^a$ and record send time as t_2 **7.** Wait For $Time\ Stamp\ Indication_b$ from $Tester^a$ and get the values t_1 and t_2

8. Using t_0, t_1, t_2, t_3 and equation $Offset=(d_{m2s} - d_{s2m})/2$ given in 5.1, calculate offset between $Tester^a$ and $Tester^b$. Adjust local clock of $Tester^b$ by using $Clock_{slave} = Offset + Clock_{Master}$.
9. Communication establishment with IUT through $port\ b$
10. Wait for reception of $!EstablishOK^b$ from IUT and record receipt time as τ_2 (If not received for a specified time, $\boldsymbol{Tester^b_{Constraint1}}$ **FAIL**)
11. Send $O(2, \tau_2)_{b \to c}$ to $Tester^c$
12. Send $T(2, \tau_2)_{b \to c}$ to $Tester^c$
13. Wait for reception of $O(11, \tau_{11})_{a \to b}$ from $Tester^a$
14. Send $?DataReady^b$ to IUT and record the send time as τ_{12}
15. Send $T(12, \tau_{12})_{b \to d}$ to $Tester^d$
16. Wait for reception of $!FinishedReceivingData^b$ from IUT and record receipt time as τ_{13} (If not received for a specified time, $\boldsymbol{Tester^b_{Constraint2}}$ **FAIL**)
17. Send $O(13, \tau_{13})_{b \to c}$ to $Tester^c$
18. **IF** ($\tau_1 < \tau_2$) PASS **ELSE** $\boldsymbol{Tester^b_{Constraint3}}$ FAIL (Get τ_1 from $O(1, \tau_1)_{a \to b}$ sent by $Tester^a$)
19. **IF** ($\tau_2 < \tau_1 + 30$) PASS **ELSE** $\boldsymbol{Tester^b_{Constraint4}}$ FAIL (Get τ_1 from $T(1, \tau_1)_{a \to b}$ sent by $Tester^a$)

Figure 6-6 Pseudocode of Tester^b

$\boldsymbol{Tester^c}$ **Pseudocode**
1. Initializations **2.** Communication Establishment With $Tester^a$ **3.** Communication Establishment With $Tester^b$

4. Communication Establishment With $Tester^d$
5. Wait For $Synch_c$ from $Tester^a$ and record receipt time as t_1
6. Send $Delay\ Request_c$ to $Tester^a$ and record send time as t_2
7. Wait For $Time\ Stamp\ Indication_c$ from $Tester^a$ and get the values t_1 and t_2
8. Using t_0, t_1, t_2, t_3 and equation $Offset = (d_{m2s} - d_{s2m})/2$ given in 5.1, calculate offset between $Tester^a$ and $Tester^c$. Adjust local clock of $Tester^c$ by using $Clock_{slave} = Offset + Clock_{Master}$.
9. Communication establishment with IUT through $port\ c$
10. Wait for reception of $!EstablishOK^c$ from IUT and record receipt time as τ_3 (If not received for a specified time, $\boldsymbol{Tester^c_{Constraint1}}$ **FAIL**)
11. Send $O(3, \tau_3)_{c \to d}$ to $Tester^d$
12. Send $T(3, \tau_3)_{c \to d}$ to $Tester^d$
13. Wait for reception of $!DataReady^c$ from IUT and record receipt time as τ_{14} (If not received for a specified time, $\boldsymbol{Tester^c_{Constraint2}}$ **FAIL**)
14. Send $O(14, \tau_{14})_{c \to d}$ to $Tester^d$
15. **IF** ($\tau_2 < \tau_3$) PASS **ELSE** $\boldsymbol{Tester^c_{Constraint3}}$ FAIL (Get τ_2 from $O(2, \tau_2)_{b \to c}$ sent by $Tester^b$)
16. **IF** ($\tau_3 < \tau_2 + 30$) PASS **ELSE** $\boldsymbol{Tester^c_{Constraint4}}$ FAIL (Get τ_2 from $T(2, \tau_2)_{b \to c}$ sent by $Tester^b$)
17. **IF** ($\tau_{13} < \tau_{14}$) PASS **ELSE** $\boldsymbol{Tester^c_{Constraint5}}$ FAIL (Get τ_{13} from $O(13, \tau_{13})_{b \to c}$ sent by $Tester^b$)

Figure 6-7 Pseudocode of Tester^c

$\boldsymbol{Tester^d}$ **Pseudocode**
1. Initializations **2.** Communication Establishment With $Tester^a$ **3.** Communication Establishment With $Tester^b$ **4.** Communication Establishment With $Tester^c$ **5.** Wait For $Synch_d$ from $Tester^a$ and record receipt time as t_1 **6.** Send $Delay\ Request_d$ to $Tester^a$ and record send time as t_2 **7.** Wait For $Time\ Stamp\ Indication_d$ from $Tester^a$ and get the values t_1 and t_2 **8.** Using t_0, t_1, t_2, t_3 and equation $Offset$=$(d_{m2s} - d_{s2m})/2$ given in 5.1, calculate offset between $Tester^a$ and $Tester^d$. Adjust local clock of $Tester^d$ by using $Clock_{slave} = Offset + Clock_{Master}$. **9.** Communication establishment with IUT through $port\ d$ **10.** Wait for reception of $!\,EstablishOK^d$ from IUT and record receipt time as τ_4 (If not received for a specified time, $\boldsymbol{Tester^d_{Constraint1}}$ **FAIL**) **11.** Send $O(4, \tau_4)_{d \to a}$ to $Tester^a$ **12.** Wait for reception of $!\,DataReady^d$ from IUT and record receipt time as τ_{15} (If not received for a specified time, $\boldsymbol{Tester^d_{Constraint2}}$ **FAIL**) **13. IF** $(\tau_3 < \tau_4)$ PASS **ELSE** $\boldsymbol{Tester^d_{Constraint3}}$ FAIL (Get τ_3 from $O(3, \tau_3)_{c \to d}$ sent by $Tester^c$) **14. IF** $(\tau_4 < \tau_3 + 30)$ PASS **ELSE** $\boldsymbol{Tester^d_{Constraint4}}$ FAIL (Get τ_3 from $T(3, \tau_3)_{c \to d}$ sent by $Tester^c$) **15. IF** $(\tau_{14} < \tau_{15})$ PASS **ELSE** $\boldsymbol{Tester^d_{Constraint5}}$ FAIL (Get τ_{14} from $O(14, \tau_{14})_{c \to d}$ sent by $Tester^c$) **16. IF** $(\tau_{15} < \tau_{12} + 0.005)$ PASS **ELSE** $\boldsymbol{Tester^d_{Constraint6}}$ FAIL (Get τ_{12} from $T(12, \tau_{12})_{b \to d}$ sent by $Tester^b$)

Figure 6-8 Pseudocode of Tester^d

Each Local Tester was implemented using C language that is widely used for real time applications using MC/OS APIs. The Multicomputer Operating System (MC/OS) is a real-time operating system developed by Mercury for use on the distributed, heterogeneous hardware of a RACE multicomputer system. Each local tester runs on MC/OS. Mercury uses MC/OS rather than a standard operating system because commercial operating systems such as UNIX and Windows are designed for use in single-processor computers. Such operating systems have two types of drawbacks relative to multicomputing:

- They lack capabilities that multicomputer applications typically require.
- They provide capabilities that multicomputer applications do not need.

Mercury developed MC/OS to avoid these drawbacks and to provide exactly those capabilities that RACE multicomputer applications need. MC/OS is optimized to give fast, predictable performance when used to perform multicomputing over the RACEway.

6.7. Test Execution and Analysis of Test Results

Software tests were performed for 3 times for different versions of IUT using the developed Test System. First version of IUT was tested previously for the functional correctness with the method explained in 3.2. Local testers' constraints are regarded as constraints of IUT. As a result of test execution a verdict is produced for each software constraint. Software test results for the first version of IUT are given in the Table 6-1.

Table 6-1 Test Results for First Version of IUT

Software Constraint	Software Test Result
$Tester^{a}_{Constraint1}$	PASS
$Tester^{a}_{Constraint2}$	FAIL
$Tester^{a}_{Constraint3}$	COULD NOT BE PERFORMED
$Tester^{a}_{Constraint4}$	COULD NOT BE PERFORMED
$Tester^{a}_{Constraint5}$	COULD NOT BE PERFORMED

Table 6-1 cont'd

$Tester^{b}_{Constraint1}$	PASS
$Tester^{b}_{Constraint2}$	COULD NOT BE PERFORMED
$Tester^{b}_{Constraint3}$	COULD NOT BE PERFORMED
$Tester^{b}_{Constraint4}$	COULD NOT BE PERFORMED
$Tester^{c}_{Constraint1}$	PASS
$Tester^{c}_{Constraint2}$	COULD NOT BE PERFORMED
$Tester^{c}_{Constraint3}$	COULD NOT BE PERFORMED
$Tester^{c}_{Constraint4}$	COULD NOT BE PERFORMED
$Tester^{c}_{Constraint5}$	COULD NOT BE PERFORMED
$Tester^{d}_{Constraint1}$	PASS
$Tester^{d}_{Constraint2}$	COULD NOT BE PERFORMED
$Tester^{d}_{Constraint3}$	COULD NOT BE PERFORMED
$Tester^{d}_{Constraint4}$	COULD NOT BE PERFORMED
$Tester^{d}_{Constraint5}$	COULD NOT BE PERFORMED
$Tester^{d}_{Constraint6}$	COULD NOT BE PERFORMED

As seen from the table $Tester^{a}_{Constraint1}$, $Tester^{b}_{Constraint1}$, $Tester^{c}_{Constraint1}$ and $Tester^{d}_{Constraint1}$ constraints are satisfied which means outputs are received from IUT. $Tester^{a}_{Constraint2}$ failed because output $!Message3Result^{a}$ could not be received. Due to this fault, execution of LTSs terminated and remaining constraints could not be tested. This fault could not be found by the method discussed in 3.2 since this output is not observable.

After fixing the fault found in the first version of IUT, second version of IUT was developed. For the second version of the IUT software tests were held and the results are given in the Table 6-2.

Table 6-2 Test Results for Second Version of IUT

Software Constraint	Software Test Result
$Tester^{a}_{Constraint1}$	PASS
$Tester^{a}_{Constraint2}$	PASS
$Tester^{a}_{Constraint3}$	PASS
$Tester^{a}_{Constraint4}$	PASS
$Tester^{a}_{Constraint5}$	FAIL
$Tester^{b}_{Constraint1}$	PASS
$Tester^{b}_{Constraint2}$	PASS
$Tester^{b}_{Constraint3}$	PASS
$Tester^{b}_{Constraint4}$	PASS
$Tester^{c}_{Constraint1}$	PASS
$Tester^{c}_{Constraint2}$	PASS
$Tester^{c}_{Constraint3}$	PASS
$Tester^{c}_{Constraint4}$	PASS
$Tester^{c}_{Constraint5}$	PASS
$Tester^{d}_{Constraint1}$	PASS
$Tester^{d}_{Constraint2}$	PASS
$Tester^{d}_{Constraint3}$	PASS
$Tester^{d}_{Constraint4}$	PASS
$Tester^{d}_{Constraint5}$	PASS
$Tester^{d}_{Constraint6}$	FAIL

As seen from the Table 6-2, $Tester^{a}_{Constraint5}$ and $Tester^{d}_{Constraint6}$ failed. Both of these constraints are timing constraints for IUT. These faults could not be found by the method discussed in 3.2 since outputs are not observable, it is not possible to control inputs of IUT and it is not possible to check temporal behavior of IUT.

After fixing the faults found for second version of IUT, third version of IUT was developed. Test results of IUT for its third version are given in the Table 6-3. Since TS enables us to reproduce GTS it is possible check whether these faults are recovered or not by applying GTS to IUT.

Table 6-3 Test Results for Third Version of IUT

Software Constraint	Software Test Result
$Tester^{a}_{Constraint1}$	PASS
$Tester^{a}_{Constraint2}$	PASS
$Tester^{a}_{Constraint3}$	PASS
$Tester^{a}_{Constraint4}$	PASS
$Tester^{a}_{Constraint5}$	PASS
$Tester^{b}_{Constraint1}$	PASS
$Tester^{b}_{Constraint2}$	PASS
$Tester^{b}_{Constraint3}$	PASS
$Tester^{b}_{Constraint4}$	PASS
$Tester^{c}_{Constraint1}$	PASS
$Tester^{c}_{Constraint2}$	PASS
$Tester^{c}_{Constraint3}$	PASS
$Tester^{c}_{Constraint4}$	PASS
$Tester^{c}_{Constraint5}$	PASS
$Tester^{d}_{Constraint1}$	PASS
$Tester^{d}_{Constraint2}$	PASS
$Tester^{d}_{Constraint3}$	PASS
$Tester^{d}_{Constraint4}$	PASS
$Tester^{d}_{Constraint5}$	PASS
$Tester^{d}_{Constraint6}$	PASS

6.8. Test Results for Different Global Test Sequences

In addition to developing test architecture for $GTS = tr_1.tr_2.tr_3.tr_4.tr_5.tr_6.tr_7.tr_8.tr_9.tr_{10}.tr_{11}.tr_{12}.tr_{13}.tr_{14}.tr_{15}$ some other Global Test Sequences were also considered. Test architectures were developed and test executions were performed for following Global Test Sequences.

$$GTS2 = tr_1.tr_2.tr_3.tr_4.tr_7.tr_8.tr_9.tr_{10}.tr_{11}.tr_{12}.tr_{13}.tr_{14}.tr_{15}.tr_5.tr_6$$

$$GTS3 = tr_1.tr_2.tr_3.tr_4.tr_{12}.tr_{13}.tr_{14}.tr_{15}.tr_5.tr_6.tr_9.tr_{10}.tr_7.tr_8.tr_{11}$$

Complexity of test architecture and local test sequences differ for different Global Test Sequences since number of coordination messages differ for different Global Test Sequences. Same test results of $GTS1$ were obtained for $GTS2$ and $GTS3$. Same faults were found for different Global Test Sequences. This proves that 4p-Timed Automata Model of IUT is correct and there is no missing state or extra state for the IUT since we have found the same faults for different global test sequences.

6.9. Evaluation of Test Results

Comparison of test approach used in the company and distributed test approach can be listed as;

- In most cases it is not desirable to use real environment during testing. This is due to safety and cost considerations since the confidence in correctness of real time system is low. Sometimes it is not possible to use real environment since it is not available. The approach discussed in 3.2 needs real environment for testing but the proposed distributed test approach does not need the real environment for testing.
- Rare event situations which occur only rarely in the real world cannot be created easily with the approach discussed in 3.2. Often it is either difficult or unsafe to obtain these situations from the real environment. By using the proposed distributed test approach it is much easier to create rare event situations. In our implementation $Message1^a$ is sent to IUT only when a failure occurs in the hardware of the system. It is hard to create this failure

on real system but with distributed test approach we can easily feed this input to IUT.

- It is not possible achieve test repeatability with the approach discussed in 3.2. Temporal behavior of distributed real-time system depends on the timing of events and it is not possible to control timing of inputs with this approach. However the proposed distributed test approach guarantees test repeatability by controlling timing and order of inputs.

- Controllability of IUT is not possible with the approach used in the company since inputs of IUT are not controlled. On the other hand, the proposed distributed test approach achieves controllability by controlling the applied inputs.

- It is not possible to observe input and output of IUT with the approach used in the company since IUT has interfaces with the rest of the system that we cannot observe. On the other hand observability is possible with the proposed distributed test approach by observing ports of local testers.

It is not possible to test temporal behavior of IUT with the approach used in the company due to observability problem. On the other hand it is possible to test temporal behavior of IUT with the proposed distributed test approach since this method achieves observability.

CHAPTER 7

CONCLUSION

Distributed real-time systems are mostly safety critical systems that are widely used nowadays for many critical applications. Software testing of safety critical distributed real-time systems is crucial since the failure of these systems result in catastrophic consequences. Most of the available software testing techniques support only software testing of sequential programs that do not have timing issues. For testing of distributed real time systems distributed behavior and timing issues of the application must be taken into account. There is a limited number of studies available in the literature on testing distributed real time systems which propose a complete method for test case generation, test architecture and test execution and few studies report the implementation results of these techniques.

In this thesis study, following a literature search on available software testing techniques for testing of distributed real time systems, the technique proposed in [6] was considered since it proposes a complete method for test case generation, test architecture and test execution. Software testing technique proposed in [6] has been implemented on sample projects, test architecture was developed and results of this study have been reported in this study. A software tool is developed for distributing the Global Test Sequence to Local Testers during the course of implementation.

Synchronization of local testers' clocks is a crucial issue of test architecture and test execution for testing of timing issues related with distributed real time systems. In [6] it is not discussed how to synchronize local testers within the test architecture and test execution. In this study IEEE 1588 Precision Time Protocol is partially implemented for synchronization of local testers' clocks.

The implementation results show that distributed test architecture presents certain advantages for testing of distributed real-time systems. Test repeatability, controllability of IUT, observability of IUT and ability to test temporal behavior of IUT are major benefits of distributed test architecture compared to method used in the company previously. Fault coverage of distributed architecture increases for finding timing faults and order faults when compared to previous method used in the company.

For each different GTS, distributed test architecture requires development of different local testers. Furthermore it is very costly to develop this test architecture since it is very complex. In spite of complexity of distributed test architecture, distributed test architecture can be used for testing critical systems for which failure of system results in serious hazards.

During test execution Test System determines the timing of inputs sent by TS to IUT. TS respects timing constraints at each execution of GTS but timing of inputs can differ at each test execution. This issue has to be considered as a future study in the implementation of test architecture.

REFERENCES

1. A. En-Nouaary, F. Khendek, and R. Dssouli, "Testing embedded real-time systems," in *seventh International Conference on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 417 - 424.

2. A. En-Nouaary, F. Khendek, and R. Dssouli, "Fault coverage in testing real-time systems," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference* , Dec. 1999, pp. 150 - 157.

3. A. Khoumsi, "A temporal approach for testing distributed systems," *IEEE Transactions on Software Engineering*, Vol. 28, No. 11, November 2002 .

4. A. En-Nouaary, F. Khendek, A. Elqortobi, and R. Dssouli, "Timed Test Cases Generation Based on State Characterization Technique," in *19th IEEE Real-Time Systems Symposium (RTSS'98)*, Dec. 1998.

5. A. Khoumsi, "A new method for testing real time systems," in R*eal-Time Computing Systems and Applications*, Dec. 2000, pp. 441 - 450.

6. A. Khoumsi, "Testing distributed real time systems using a distributed test architecture," in *Computers and Communications, 2001. Proceedings. Sixth IEEE Symposium,* July 2001, pp. 648 - 654.

7. K. Correll, N. Barendt, and M. Branicky, "Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol," In P*roc. Conf. on IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, October 2005.

8. W. Schütz, "A test strategy for the distributed real-time system mars," In *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pp. 20–27, 1990.

9. "IEEE Standard Glossary of Software Engineering Terminology," *IEEE standards collection*, *IEEE std 610.12*, September 1990.

10. H. Thane, and H. Hansson, "Towards Systematic Testing of Distributed Real-Time Systems," in *Proceedings of the 1999 IEEE Real-Time Systems Symposium*.

11. H. Thane, and H. Hansson, "Handling Interrupts in Testing of Distributed Real-Time Systems," in *6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Dec. 1999.

12. I. Ho, and J. Lin, "Generating Test Cases for Real-Time Software by Time Petri Nets Model," In *Proceedings of the 8th Asian Test Symposium*.

13. IEEE Std 1588-2002 (http://ieee1588.nist.gov)

14. *Mercury PowerPC 7410 Daughtercard Data Sheet.* [Online]. Available: http://www.mc.com/uploadedfiles/PPC7410_DS_4P_11.pdf, Last accesed date April 2008

15. H. Thane, A. Pettersson, and D. Sundmark, "The asterix real-time kernel," In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*.

16. A. J. Whittaker, "What Is Software Testing? And Why Is It So Hard?," *Journal IEEE Software,* January/February 2000.

17. H. Ural, and D. Whittier, "Distributed testing without encountering controllability and observability problems," *Information Processing Letters 88, 2003,* pp. 133-141.

18. W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," *Computing Surveys*, Vol. 14, No. 2, June 1982.

19. "Raceway Interlink functional specification", from http://www.mc.com/uploadedFiles/racewayintrlnk-spec.pdf, Last accesed date April 2008

20. Z. Wang, J. Wu, and X. Yin, "Towards interoperability test generation of time dependent protocols: a case study," in *Proceedings of the Global Telecommunications Conference*, 2004.

21. B. Broekman, and E. Notenboom, "Testing Embedded Software," Addison-Wesley, 2003, pp. 111-134.

22. G. Gönenç, "A Method for the Design of Fault Detection Experiments," *IEEE Transactions on Computers, June 1970,* pp. 551-558.

APPENDIX A

POWERPC 7410 DAUGHTERCARDS SPECIFICATIONS

P2J128J

RACEway ports: 2

Processor frequency: 400 MHz

Compute nodes: 2

Memory frequency: 133 MHz

SDRAM per CN: 128 MB

SDRAM per daughtercard: 256 MB

L2 cache frequency: 266 MHz

L2 cache per CN: 2 MB

Weight: 0.42 lb*

Dimensions: 5.0 in x 4.435 in

Power consumption**: 16.6W

Daughtercards per MCJ6***: 2

Daughtercards per MCJ9: 9

Q1P2J256J-Q1

RACEway ports: 2

Processor frequency: 400 MHz

Compute nodes: 2

Memory frequency: 133 MHz

SDRAM per CN: 256 MB

SDRAM per daughtercard: 512 MB

L2 cache frequency: 266 MHz

L2 cache per CN: 2 MB

Weight: 0.43 lb*

Dimensions: 5.0 in x 4.435 in

Power consumption**: 16.6W

Daughtercards per MCJ6***: 2

Daughtercards per MCJ9: 9

* Rugged version weighs an additional 0.01 lb.

** Maximum typical power consumption measured with concurrent FFTs and I/O.

*** Requires 5-row connectors on MCJ6 and VME backplane.

Commercial Environmental Specifications

Operating temperature: 0ºC to 40ºC up to an altitude of 10,000 ft
(inlet air temperature at motherboard's recommended minimum airflow)

Storage temperature: -40ºC to +85ºC

Relative humidity: 10% to 90% (non-condensing)

As altitude increases, air density decreases, hence the cooling effect of a particular CFM rating decreases. Many manufacturers specify altitude and temperature ranges that are not simultaneous. Notice that the above operating temperature is specified simultaneously with an altitude. Different limits can be achieved by trading among altitude, temperature, performance, and airflow.