Timed Test Suite Generation Based on Test Purpose Expressed in MSC

Gang Liu

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

February 2004

ABSTRACT

Timed Test Suite Generation Based on Test Purpose Expressed in MSC

Gang Liu

When testing real time system, Automating timed test suite generation has much advantages over manual test suite generation. Formal models are usually used to describe the complex system behaviours, such as TIOA (Timed Input Output Automaton) and MSC (Message Sequence Charts). Therefore, test suite can be generated from the formal model of the specification. Exhaustive test are preferred to cover all faults in a test, but it is almost impossible. Test purpose represents the partial requirements to be tested.

In this thesis, we present a new method for automatically generating timed test suite based on the test purpose expressed in MSC, and the specification expressed in TIOA. A set of integrated algorithms is provided to process input test purpose and specification to generate timed test suite. In this method, MSC is transformed to TIOA, and the test purpose and the specification are synchronized as one single product. This single product is then sampled to construct a grid automaton. Finally, a traversal algorithm is applied to the grid automaton to generate test suite. We implemented this method and experimented it with different examples. Comparing with other methods, test cases generated by our method have a smaller number, and they are self-evident and can be easily represented by TTCN (Tree and Tabular Combined Notation).

ACKNOWLEDGEMENTS

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

I also want to thank all teachers who have taught me in my degree program at Concordia University.

Special thanks go to my colleges and friends, Yi Qin and LiangGang Yu, for their warm-hearted suggestions and help.

Finally I dearly thank my wonderful family, my parents and my sister. This thesis is dedicated to them, for their forever encouragement and supports.

TABLE OF CONTENTS

List of Figures viii

CHAPTER 1 1

Introduction 1

1.1 Testing Real-Time Software Systems 2

1.2 Automatic Test Suite Generation 3

1.3 Guideline of This Thesis 4

CHAPTER 2 6

Overview of Conformance Testing 6

2.1 Conformance Testing and the Standard 6

2.1.1 Conformance Testing Process 7

2.1.2 Test Methods 10

2.2 Formal Specification Methods 14

2.3 Conformance Testing and Specification 16

2.4 Test Hypotheses and Test Purpose 17

2.5 Fault Model 18

2.6 Test Suite Generation 20

2.7 Conclusion 21

CHAPTER 3 23

Timed Test Suite Generation Based On Test Purpose 23

3.1 Definitions 23

3.1.1 Timed Input Output Automaton (TIOA) 24

3.1.2 Message Sequence Chart (MSC) 25

3.1.3 Synchronous Product (SP) 28

3.1.4 Regions Graph and Grid Automaton 29

3.2 Framework 31

3.3 Algorithms 34

3.3.1 Converting MSC to TIOA 34

3.3.2 Clock Minimizing 40

3.3.3 Synchronous Product Construction 45

3.3.4 Sampling - Deriving Grid Automaton 47

3.3.5 Traversal 50

3.4 Time-Related Fault Model 54

3.5 Fault Coverage 55

3.6 Conclusion 57

CHAPTER 4 58

Implementation of Algorithms 58

4.1 Development Tools and Environment 58

4.1.1 Bison and Flex 59

4.2 Activities Diagrams 61

4.3 Class Diagrams 61

4.3.1 TIOA Classes 62

4.3.2 MSC and MSC Translator Classes 65

4.3.3 Clock Minimizing Classes 66

4.3.4 Sampling Class 67

4.4 Case Studies 68

4.4.1 An Detailed Example 68

4.4.1.1 Specification 68

4.4.1.2 Test Purpose 70

4.4.1.3 Results 72

4.4.2 Other Examples 76

4.4.2.1 10-Digit Telephone System 76

4.4.2.2 Multimedia System 77

4.4.2.3 Synchronization Protocol 78

4.4.3 Related Works 80

4.5 Conclusion 81

CHAPTER 5 82

Summary and Future Work 82

5.1 Summary 82

5.2 Future Work 84

Bibliography 87

APPENDIX A. SIMPLE MSC Grammar 94

APPENDIX B. TIOA Grammar 99

APPENDIX C. Lex and Yacc Files for SIMPLE MSC 103

APPENDIX D. Lex and Yacc Files For TIOA 115

APPENDIX E. Output Test Cases for Telephone System 122

LIST OF FIGURES

Figure 2-1: Conformance Testing Framework 8

Figure 2-2: Local Single Layer Test Method 11

Figure 2-3: Distributed Single Layer Test Method 12

Figure 2-4: Remote Single Layer Test Method 13

Figure 2-5: Coordination Single Layer Test Method 13

Figure 2-6. Software Life Cycle and Testing 14

Figure 2-7. Conformance Relationship 17

Figure 3-1. Timed Test Case Generation Framework 31

Figure 4-1 Activity Diagram 60

Figure 4-2. Package Diagram 61

Figure 4-3. Class Diagram of TIOA 64

Figure 4-4. Classes of MSC 65

Figure 4-5. Classes of MSC Translator 66

Figure 4-6. Class Diagram of ClockMinimize 67

Figure 4-7 Class Diagram of GACreator 67

Figure 4-8. The TIOA Specification of Telephone System 69

Figure 4-9. A Test Purpose in MSC for Telephone System 71

Figure 4-10. Test Purpose as TIOA before Clock Minimizing 72

Figure 4-11. Test Purpose as TIOA after Clock Minimizing 73

Figure 4-12. Synchronous Product 73

Figure 4-13. Grid Automaton 74

Figure 4-14 Some Resulting Test Cases 75

Figure 4-15 Specification of 10-digit Telephone System 76

Figure 4-16 Test Purpose of 10-digit Telephone System 77

Figure 1-17 Specification of Multimedia System 77

Figure 4-18 Test Purpose of Multimedia System 78

Figure 4-19 Specification of Media Synchronization Protocol 78

Figure 4-20 Test Purpose of Media Synchronization System 79

Figure 5-1. Multi-processes Testing 85

CHAPTER 1

Introduction

Testing plays a key role in software life cycle. This is because in recent years, software systems are much more complex, and distribution and parallel calculation have become more usual. In software engineering, a specification specifies the functionalities, performance, execution environments and other issues regarding software products. The implementation of the software must conform to the specification. Testing is one of the approaches for checking the conformance between an implementation and its specification.

In testing, test suite is applied to the implementation under test (IUT), and the output interaction traces from IUT are observed and analyzed. From this analysis, a verdict is reached to conclude whether the implementation is correct or not, with respect to the specification.

Test Purposes are usually used to specify what to test, because it is almost impossible to cover all the faults in a test. Test Purposes represent partial conformance requirements in the specification. Another benefit of using test purposes is the reduction of the number of test cases.

When testing real time systems, there exist particular difficulties due to the system behaviors limited by time constraints. A real time system has to react to the stimuli from the environment within a limited or specified period. Therefore, the correctness of the system functionalities depends on not only its actions, but also the time constraints on these actions.

In this thesis, we discuss the issues of testing real time software systems. Especially, we focus on how to automatically generate timed test suite based on test purpose with the given specification.

1.1 Testing Real-Time Software Systems

Real-time software is used to control safety critical systems, such as medical device monitoring, air traffic control, power plant control and emerging multimedia-over-internet applications. Real time applications are harder to test because they require not only the correct output signals, but also the signals occurring at the correct time. To ensure the correctness of the implementation, the output signals and the time should both be checked upon testing.

The most efficient means of precisely describing complicated system behaviors is to use formal models. Various well-known models are widely used, such as Finite State Machine (FSM), Petri-nets and Input Output Automata. Most of proposed formal models for real-time systems are time enrichment of those traditional models, with time constraints and additional time labels, such as Timed Finite State Machine (TFSM) and Timed Input Output Automata (TIOA)([14][17]). To test real time system described by the formal models, the obvious means for lowering test cost is deriving test suite from the models automatically. The observed output traces from the IUT should also be analyzed against the models.

1.2 Automatic Test Suite Generation

Automating test suite generation is greatly preferred. Because compared to manual t est c ase g eneration, i t h as the a dvantages o f h igher f ault c overage p ower a nd lower test cost.

Formal models based on state machines describe what is the system's state, and how a system changes its state, as well as when and where a system accepts inputs and sends outputs. Different methods ([21] [26] [27] [29] [31]) have been developed to derive test suite from the specification as formal models. However, these methods have the problem of resulting in large number of test cases. That could be unacceptable in real-world testing, especially for real time systems with complicated behaviors. To solve these problems, one of the approaches is covering only partial functionalities of the system in a test. These partial functionalities under test are described by Test Purposes. The test cases are only used to ensure the correctness of the functionalities described by the test purpose. Hence, the number of test cases could be limited. To verify the correctness of IUT, it is necessary to verify if the IUT conforms to both the specification and test purpose.

Test purpose is also represented by formal methods. This provides the benefit of automated processing of generating test suite. Our approach to automatically generate timed test suite is based on test purpose, which is represented by Message Sequence Chart (MSC)[2].

1.3 Guideline of This Thesis

Although there are already some methods for automatically deriving test suite, the problems of those methods still exist. First, the number of generated test cases is too high, because of state space explosion problems. Second, the time constraints are seldom considered. Third, most of the proposed methods only consider partial works of the whole test suite generation procedure.

In this thesis, we will provide a new method of automatic timed test suite generation based on test purpose. We use TIOA to describe the specification, and MSC to describe the test purpose. We synchronize the specification and test purpose into a single product, and use Region Graph and Grid Automaton, as well as clock minimizing algorithm, to analyze the time constraints and reduce the state space of the automaton. We develop an integrated set of algorithms for the whole processing from input specification and test purpose to output test suite. We also implemented this test suite generation method as a tool.

The remainder of this thesis is organized as follows: Chapter 2 is an overview of testing activities. The well-known test methodology is introduced, as well as the framework of conformance testing and test methods. We also discuss the conformance relation between reference specification and IUT, and how test case generation is related to formal models. The c hapter also introduces the concepts of test hypothesis and test purpose, as well as fault coverage and fault model.

Chapter 3 presents a method and framework for timed test suite generation based on test purpose expressed in MSC, and specification expressed in TIOA. First, the formal definitions of related concepts are given. Then, each step of test suite generation in the framework is explained. The detailed algorithms for each step are provided. Also the fault coverage of our method is discussed.

Chapter 4 provides the implementation of the method we introduce in Chapter 3. Object-oriented designing is used in the design phase. We provide the activities diagrams and class diagrams in UML. Examples of applying our implementation are discussed.

Chapter 5 comes with the summary of this thesis and the future work. We will discuss the possible future work related to test suite generation in that chapter.

CHAPTER 2

Overview of Conformance Testing

In this chapter we will introduce the background concepts and knowledge about testing. A standard conformance testing methodology and framework is introduced in Section 2.1. Sections 2.2 and 2.3 introduce formal specification methods and their role in conformance testing. Section 2.4 discusses test hypotheses and test purpose, as well as their importance in testing. Section 2.5 provides the FSMs fault model. Several well-known test suite derivation methods are introduced in Section 2.6.

2.1 Conformance Testing and the Standard

The distributed real time systems communicate with the environment by exchanging messages. The communication behaviors are based on communication protocols. A p rotocol i mplementation m ust b e g uaranteed t o b e c ompatible w ith o ther implementations. To achieve the compatibility, a detailed specification is necessary, and therefore all implementations have to conform to the specification. To ensure the conformance, one approach is to test the implementation. This is referred to as conformance testing.

There are various reasons that cause implementations to fail to interact with each other. First, the developers of the implementation could introduce errors. Second, the specification could be incomplete, so that different implementations could have different behaviors for the incomplete part of the specification. Third, the specification could provide a range of choices that results in incompatibilities between implementations. Finally, the implementation could adapt a different interpretation of the same specification from others.

Conformance testing is the activity of checking whether a new implementation conforms to a specification or not. However, different test developer could have different principles when deciding if the implementation conforms to specification. Moreover, the same product could be tested more than once by different test developers. Therefore, a general principle and test procedure for conformance testing are necessary, so that the repeated conformance testing for the same system can be minimized. International Organization for Standardization (ISO) has developed a standard for conformance testing, that is ISO IS-9646, "OSI Conformance Testing Methodology and Framework"([1]).

2.1.1 Conformance Testing Process

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

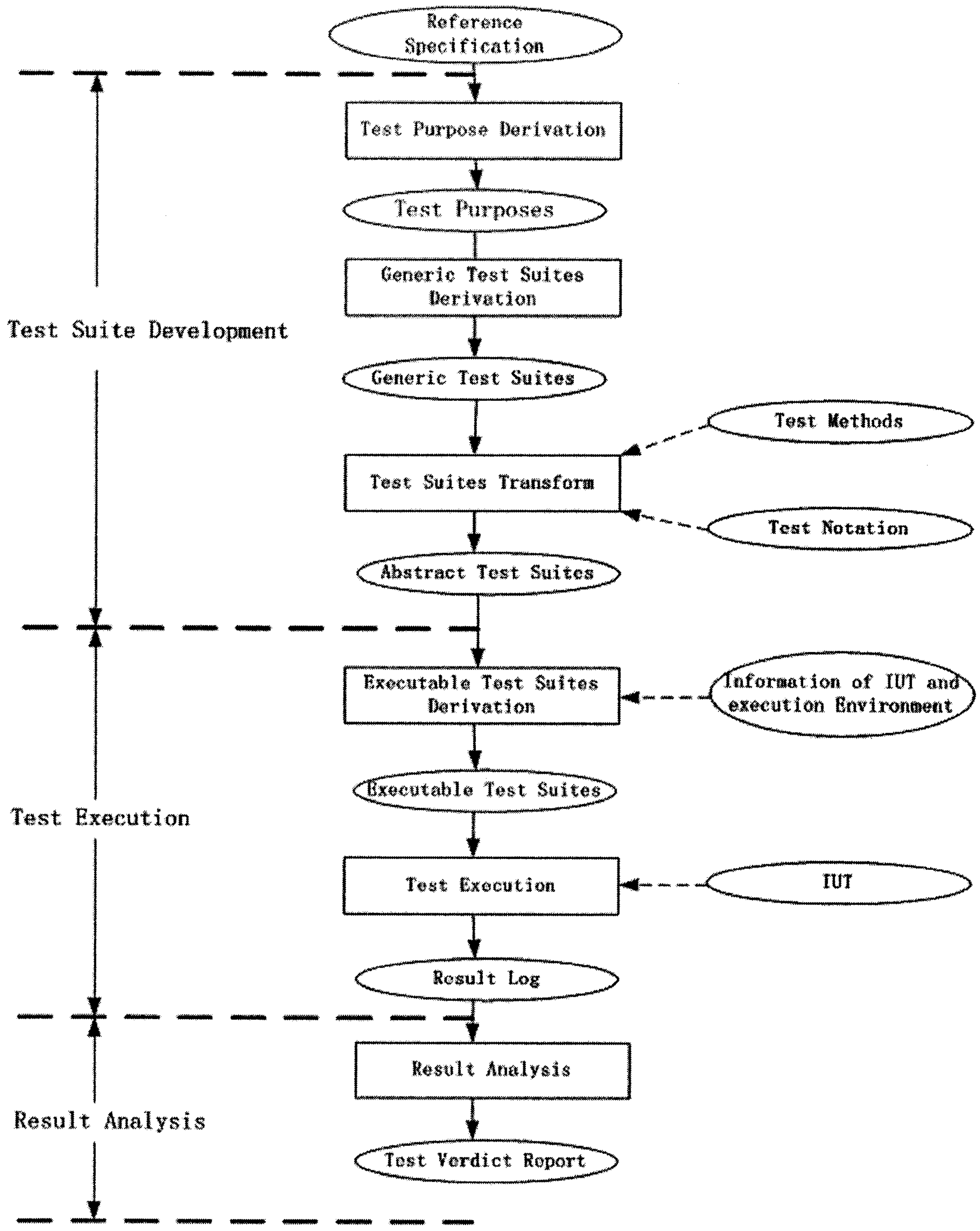

Figure 2-1: Conformance Testing Framework

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

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

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

Finally, an abstract test suite is derived from each generic test suite. The derivation is made by considering a particular test method and the constraints of the applied test environment. The resulting abstract test suite is independent of any implementation. The test cases in an abstract test suite are represented in a well-defined test notation.

A semi-formal language to specify abstract test suite is suggested in the standard, which is TTCN, the Tree and Tabular Combined Notation [18].

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

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

Result analysis. The recorded reactions of IUT are compared with the reactions specified in the test suite, and a verdict report is created for the certification of the final product. A verdict is assigned to each test case according to the recorded outputs from IUT. A verdict is either PASS, INCONCLUSIVE or FAIL. If the outputs indicate that the implementation conforms to the specification and the test purpose, a PASS verdict is concluded. Otherwise, if the implementation fails to conform to the specification, a FAIL verdict is concluded. INCONCLUSIVE verdict is concluded if the implementation conforms to the specification but the test purpose is not achieved.

2.1.2 Test Methods

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

As it is depicted in Figure 2-2, in LS-method, an Upper Tester (UT) provides the observation and c ontrols on the upper service boundary of IUT. A L ower Tester (LT) provides the observation and controls on the lower service boundary of IUT. There are two PCOs in LS-method: The upper PCO is used by UT to send and receive signal to and from the implementation, and the lower PCO is used by LT.

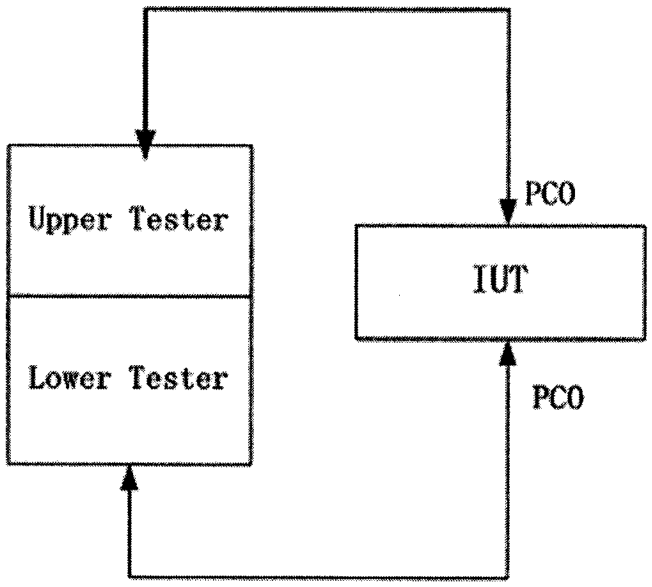

Figure 2-2: Local Single Layer Test Method

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

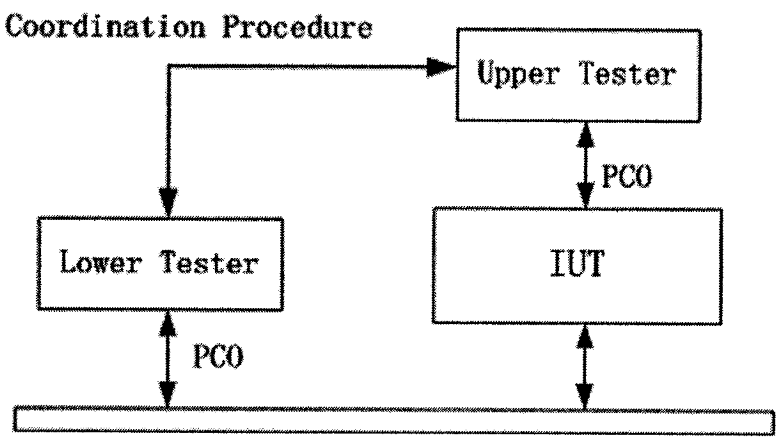

Figure 2-3: Distributed Single Layer Test Method

There exist particular difficulties when testing with DS-method. First, the independence between UT and LT implies possible fault coverage limitations. Second, because of remote testing, queuing delays could cause remote tester occupy incorrect time-related information from IUT. Third, the separated testers could face synchronization problems. Finally, the external coordination between UT and LT could also face failures since the external environment is not guaranteed to be error free.

In the RS-method (Figure 2-4), there is no upper tester, and only one PCO is available for the remote tester.

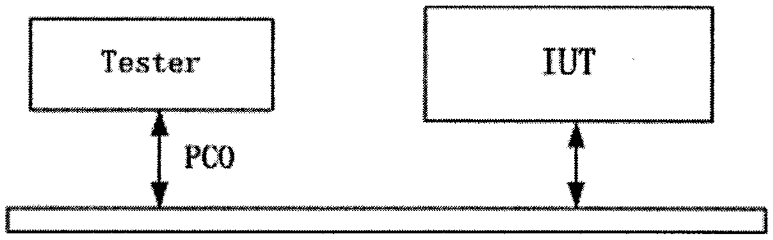

Figure 2-4: Remote Single Layer Test Method

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

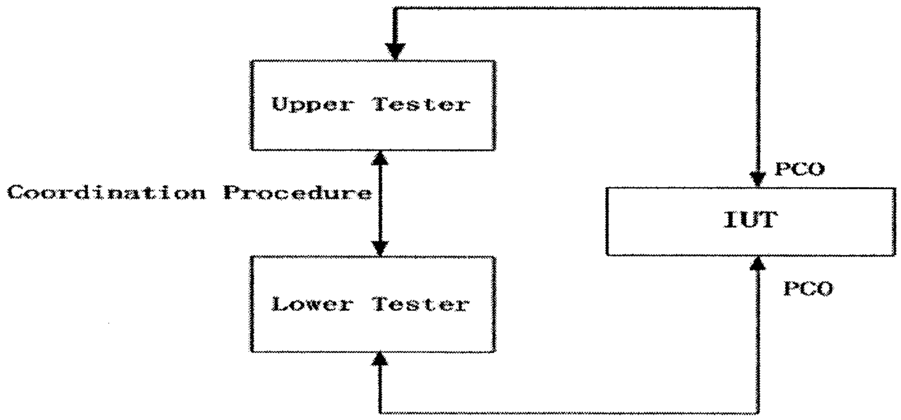

Figure 2-5: Coordination Single Layer Test Method

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

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

In the following chapters, we will introduce a new method to generate test suite based on the test purpose and the specification. According to ISO9646 framework, our method focuses on the scope of test suite generation phase.

2.2 Formal Specification Methods

In the software life cycle, the specification, implementation and testing have a very close relationship with each others. Figure 2-6 depicts how they can impact each others.

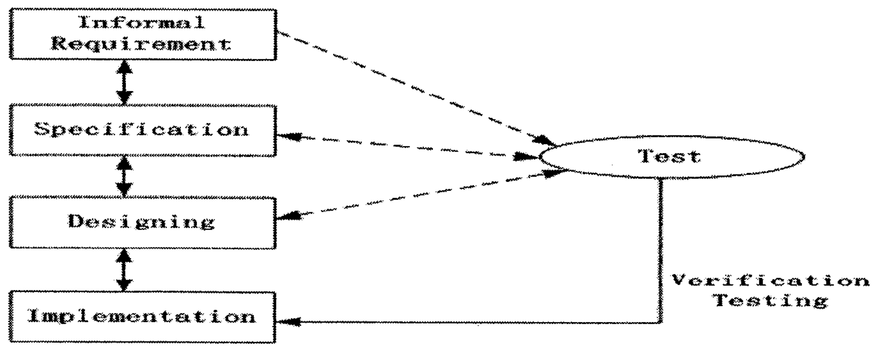

Figure 2-6. Software Life Cycle and Testing

Software life cycle starts with requirement engineering. Requirements are usually described in an informal natural language such as English, or semi-formal languages such as UML. Requirements could be imprecise and inconsistent. Then, the requirements are transferred to the specification, which gives the detail of what the user needs and how the system works. The specification specifies the user requirements in a precise and unambiguous way, but also is abstract enough and not touching the irrelevant system's structure and implementation details.

Due to the increasing complexity of distributed real time system, a precise and unambiguous specification becomes more difficult to be well defined. Using formal methods to describe the complicated behaviors of the system is necessary, since formal methods provide the way to define a precise and unambiguous specification, and make it possible to reason about the complexity of system behaviors. Moreover, formal methods also make it possible to automate the process of generating functions code and test suite.

The design comes after the specification. In this phase, the system's internal structure and detailed functions are designed. Designing should follow the requirements and specification. Under the conduction of the detailed design, the implementation is realized. Executable code, software components are created.

The final implementation must be checked to verify if it fulfils the user's requirement. In fact, not only in the implementation phase, in every phase of software life cycle, checking must be done to guarantee that the final product really does what the user needs it to do. Usually, two techniques are used to verify the correctness: validation and testing.

Validation is used to check if the specification and design represent indeed what the user needs, or whether they respect the requirement or not. Testing is a kind of experiment, the system under test is executed and observed. The tester controls and observes the system under test, and verifies its correctness. Tests can be derived from the requirement, specification and design, and then applied to the implementation. The observed results are analyzed against the requirements, specification and design. Furthermore, testing the implementation could also find out the possible incorrectness of the specification and design, and help in specification and design validation.

If the specification is described by the means of formal methods, the testing can be done as early as the specification phase. Formal models can be created to explain the system, and they are executable. The formal model's internal behaviors can be verified, e.g., in a finite state machine model, it can be checked if there exist deadlock scenarios.

2.3 Conformance Testing and Specification

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

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

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

Conformance testing is black-box testing. The implementation is required to conform to the reference specification. The test suite are derived from the specification, and then applied to the implementation. In this way, the implementation is tested against the specification, and the conformance between the implementation and the specification is verified (as shown in Figure 2-7). While the specification is specified by the means of formal models, it is possible to generate test suite automatically from formal models by some kind of generation algorithms. Compared to the manual test suite generation, automatic generation algorithm from formal models has obvious advantages, such as no human introducing errors, more fault coverage, and lower test costs.

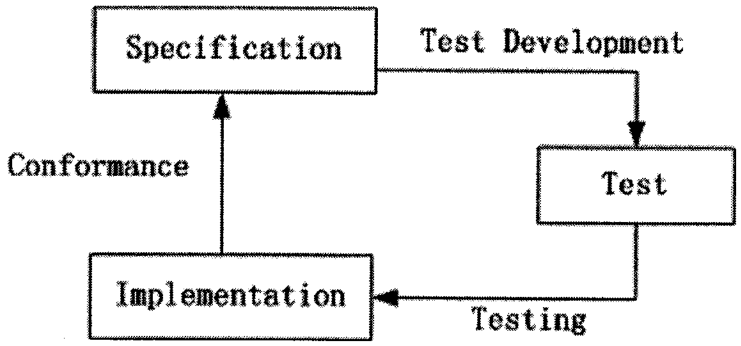

Figure 2-7. Conformance Relationship

2.4 Test Hypotheses and Test Purpose

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

The purpose to use test hypotheses is to reduce the possible implementations and test cases. One way to achieve that purpose is by using test purpose. Test purpose specifies what functionalities of the implementation to be tested. Single or partial requirements and functions specified in the specification are derived to be tested. Therefore, not all the requirements, but only a finite set of functions are considered. As we have discussed in section 2.1, ISO conformance testing framework advises that the test suites are created from the test purposes.

2.5 Fault Model

Since we cannot cover all faults by a test, how can we say a test is good or not? It is necessary to define a specific criteria that can evaluate the quality of a test. Fault Model does the job. A fault model is used for describing how faults affect the behaviors of the implementation. In a conformance testing, a test suite is said to have a complete fault coverage with respect to a given fault model, when it either satisfies the conformance relation, or there exists a test case in it which results in a verdict FAIL[16].

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

Most formal models used to describe the behaviors of distributed and real time systems are based on Finite State Machines (FSMs) or Input Output Automaton (IOA)([14]). The fault model for the FSM has the general meaning when testing is based on FSM models. The FSM fault model includes [9]:

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

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

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

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

The fault model for real-time systems will be discussed in Chapter 3.

2.6 Test suite Generation

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

Transition Tour (TT-method)([26]). By this method, the FSM is taken from the initial state and every transition is traversed at least once. This method detects all output faults; there is no guarantee that all transition errors are detected.

Distinguishing Sequence (DS-method)([36]). A distinguishing sequence is an identifying state sequence. When a distinguishing sequence is applied to a FSM, it results in different outputs for each given state. This method can detect all output and transition errors, but it cannot be guaranteed that a DS indeed exists for a given FSM.

Unique Input Output (UIO-method)([28]). In this method, there exists a different input sequence for each given state. When the unique input sequence is applied to the given state, the resulting output sequence is different from the one of other states. Again, this method cannot guarantee full fault coverage.

W-method([35]). This method includes two sets of input sequences: W-set, which consists of input sequences that can distinguish between every pair of states; and P-set, which consists of input sequences that take the machine from the initial state to a given state for a given transition. Test sequences are formed by concatenating of these two sets; It provides a way to test all misbehaviors of the implementation machine. The logic of w-method is that the prefix input sequences bring the FSM to an identified state, and then the specific transition from the state is tested. This method guarantees the detection of all faults, but it cannot guarantee the existence of w-set for a given FSM.

Wp-method([30]). Wp-method has the same detection power as W-method. The main advantage of the Wp-method over W-method is that the length of test suite is reduced. Instead of using a w-set to check each reachable state, only a subset of w-set is used. Another advantage is that Wp-method is always applicable.

Those test derivation methods are based on FSM as formal model. In conformance testing, the formal models of the specification are used to derive test suites. However, these methods have well known problem of state space explosion. The length of test sequence and the number of test cases could be too high so that the test cost is too high. That makes real world testing very difficult. Test purposes can be used to solve this problem by testing only partial specifications, and only specific faults are targeted to be covered. To make test generation automatic, test purposes are also required to be expressed in formal models. For timed test suite generation, a summary of the existing methods is given in Chapter 4.

2.7 Conclusion

Most of formal models used to describe real time system are state machines and automata with time enhancement, such as Timed Finite State Machine (TFSM), and Timed Input Output Automaton (TIOA). Specific time related faults should be covered when testing real time software. In the following chapter, We will focus on automatic timed test case generation, and we will introduce a new method of timed test suite generation based on test purpose expressed in MSC.

CHAPTER 3

Timed Test suite Generation Based On Test Purpose

A new method of deriving timed test suite is presented in this chapter. This method is based on the timed formal models used for expressing the specification and test purpose. We assume that TIOA is used for the specification and MSC is used for the test purpose. Section 3.1 introduces the definition of those formal models. Section 3.2 describes the timed test case g eneration framework step by step. Section 3.3 gives the detailed algorithms of every step. Finally, the time-related fault model and fault coverage of this method is discussed in Section 3.4 and 3.5.

3.1 Definitions

In this section, we will introduce the concepts and the formal methods that are used in our method. TIOA and MSC, as well as other theoretical ingredients: Synchronous Product, Regions Graph and Grid Automata, are introduced, so that the subsequent contents in our framework can be easily understood.

3.1.1 Timed Input Output Automaton (TIOA)

A TIOA is a tuple (I, O, L, l^0, C, T), where:

-- *I is a finite set of input actions, each input action begins with "?".*

-- *O is a finite set of output actions. Each output action begins with "!".*

-- *L is a finite set of locations.*

-- *$l^0 \in L$ is the initial location.*

-- *C is a finite set of clocks all initialized to 0 in l^0.*

-- *T is the set of transitions*

A transition can be denoted as $l \xrightarrow{\{?,!\}a,G,\lambda} l'$. Here a transition consists of a source location l, a destination location l', an input action $?a$ or an output action $!a$, a clock guard G, and a set of clocks to be reset λ. A clock guard G is a set of the time constraints on the execution of transition. The reset clocks in the set λ are set to zero when the transition is fired.

Each clock has a value, so called clock valuation, which is denoted as v. v is a non-negative real number assigned to the clock. The clock valuation of the set of clocks can be denoted as a vector $(v_1, v_2, v_3, \ldots)$ or $V(C)$. When time elapses by d time units (where d is also a non-negative real number) for any clock valuation $v \in V(C)$, the reached clock valuation is $v + d$. When the clock valuation satisfies a clock guard G, it is denoted as $v \models G$. The symbol λ represents that a subset of clocks $X \in C$ is reset to zero, which means that each clock valuation $v \in V(X)$ is assigned the value 0, also denoted as $[X:=0]v$.

We assume that the transitions are instantaneous and all clocks in C has a bounded domain $[0, max] \cup \{\infty\}$, where max is the largest value for clocks in C. Any value larger than max should be represented as ∞. All clocks are set to zero in the initial location.

TIOA is an abstract model; it can be executed by input actions, or by time elapsing. Such execution is called operational semantics: it starts at l^0, where all clocks are set to zero, then the values of clocks increase synchronously. At any time, TIOA can make a transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$, provided that the guard G is satisfied. The fired transition makes input or output a, clocks in λ are reset to zero, and finally it arrives at location l'.

Since TIOA can be run as time elapsing, its state should be carefully considered. The States of TIOA is formally defined as following:

Let $A=(I_A, O_A, L_A, l^0_A, C_A, T_A)$ be a TIOA,

-- A state of A is a pair (l, v) consisting of a location $l \in L_A$ and a clock valuation $v \in V(C_A)$.

-- The initial state of A is the pair (l^0_A, v_0), where $v_0 = 0$ for each clock $x \in C_A$. We denote the set of states of A by $S(A)$.

TIOA textual grammar is presented in Appendix B.

3.1.2 Message Sequence Chart (MSC)

MSC is widely used in modeling communication systems. MSC has enough power to describe the message exchanging and time related events. It is formally specified by ITU-T([2]). The latest specification is MSC-2000, whose first version is published in November 1999. The purpose of recommending MSC is ([2]) "to provide a trace language for the specification and description of the communication behavior of system components and their environment by means of message interchange" The communication between components can be presented in a very intuitive and transparent manner by MSC. Therefore, "the MSC language is easy to learn, use and interpret".

MSC is a formal language with the syntax definition in both textual and graphical representations. There are two kinds of MSC: basic MSC and High-level MSC (HMSC). Each basic MSC describes a scenario of the interaction between the instances. The whole MSC document is the collection of basic MSCs, which provides a clear system specification based on instances. HMSC provides a mean to combine basic MSCs.

The main elements in a MSC are instance, message, action, timer and gate. MSC is composed of interacting instances. An instance can be a process, a service or a system component. Instances interact with each other by sending or receiving messages. An instance can have internal actions. An action is an atomic event that could happen upon sending or receiving a message, or it can be an independent internal event.

Timer is used to calculate time elapsing. In MSC, the timer is related to the events of start-timer, time-out and timer-stop. A time-out or a timer-stop event is always subsequent to a start-timer event. In addition, MSC has time constraint syntax to support the notion of quantitative time, which is very useful for describing the sequence of events in time for a real time system. In MSC, events are instantaneous; time constraints can be specified in order to define the time interval within which events may occur.

A gate represents an interface between instance and environment. Any message exchanged between an instance and the environment must pass through a gate.

In our method, we only consider a simplified Message Sequence Chart, which is enough to achieve our goal of describing a test purpose. Therefore, based on the assumptions we make for using MSC as the representation of test purpose, we derive a simplified MSC grammar from the basic MSC defined in MSC-2000. This simplified MSC is called SIMPLE MSC. SIMPLE MSC is compatible with the original basic MSC, and its grammar is given in Appendix A.

The following is a mathematic definition of our simplified SIMPLE Message Sequence Chart:

A MSC M is a structure:

$M = (P, S, R, A, O, T, T_X, Y)$, where,

-- P is a finite set of instances;

-- S is a finite set of sending message events;

-- R is a finite set of receiving message events;

-- A is a finite set of local events, such as local actions, timer start, time-out, timer stop;

-- O is the ordering of S, R and A. We assume they have a total ordering relation among them, that means we know which event happens first and which happens next by O;

-- T is a finite set of timers;

-- T_X associates each timer related event with its timer;

-- Y associates each pair of dependent events with its timing restriction, and associates each action with its duration.

The instances in MSC can be any system components that communicate with each other by sending or receiving message (S, R). For the aim of test case generation, we try to simplify the situations by considering only one instance to be tested in one test purpose. Other instances are regarded as part of implementation environment. We also assume that the ordering of events is fixed. Time restriction between events can be represented by time constraints variables or timer related events.

3.1.3 Synchronous Product (SP)

The implementation under tested must satisfy the specification and the test purpose at the same time. Therefore, we must compose a merged product, which represents both the specification and the test purpose, so that test suite can be generated from a single product. This product is called Synchronous Product. Synchronous Product is represented also as a TIOA. We first convert MSC to TIOA, then merge two TIOAs into a single one as Synchronous Product.

The definition of Synchronous Product is given as following:

Let $A = (I_A, O_A, L_A, l^0_A, C_A, T_A)$ *be a specification and*

$TP = (I_{TP}, O_{TP}, L_{TP}, l^0_{TP}, C_{TP}, T_{TP})$ *be a timed test purpose. The synchronous product of A and TP is the TIOA SP:*

$SP = (I_{SP}, O_{SP}, L_{SP}, l^0_{SP}, C_{SP}, T_{SP})$ *such that:*

— $I_{SP} = I_A \cup I_{TP}$ *and* $O_{SP} = O_A \cup O_{TP}$.

— $L_{SP} \subseteq L_A \times L_{TP}$.

-- $l^0_{SP} = (l^0_A, l^0_{TP})$.

-- $C_{SP} = C_A \cup C_{TP}$.

L_{SP} and T_{SP} are the smallest relations defined by the following two rules:

$$\_\_(l_1,l_2) \in L_{SP} \wedge l_1 \xrightarrow{\{?,!\}a,G_1,\lambda_1}{}_A l_1' \in T_A \wedge l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}{}_A l_2' \notin T_{TP} \Rightarrow$$
$$(l_1',l_2) \in L_{SP} \wedge (l_1,l_2) \xrightarrow{\{?,!\}a,G_1,\lambda_1}{}_A (l_1',l_2) \in T_{SP}$$

$$\_\_(l_1,l_2) \in L_{SP} \wedge l_1 \xrightarrow{\{?,!\}a,G_1,\lambda_1}{}_A l_1' \in T_A \wedge l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}{}_A l_2' \in T_{TP} \Rightarrow$$
$$(l_1',l_2') \in L_{SP} \wedge (l_1,l_2) \xrightarrow{\{?,!\}a,G_1\&G_2,\lambda_1\cup\lambda_2}{}_A (l_1',l_2') \in T_{SP}$$

The definition also illuminates the basic idea of how the Synchronous Product is constructed from two TIOAs. SP's input set and output set are the union of specification and test purpose input set and output set. In addition, the clock set is the union of clock sets of the specification and test purpose. The location set of SP are formed by the combination of locations of the two original TIOAs.

The two rules mentioned above determine the SP 's transitions; they are explained in section 3.3 where we will provide the detailed algorithm of constructing Synchronous Products. We can see that some transition executions in the specification are not allowed in the synchronous product, because the time constraints for these transitions in the specification and the corresponding one in the test purpose cannot be simultaneously satisfied.

3.1.4 Regions Graph and Grid Automaton

Firstly, the Regions Graph([15]) is defined as:

Let A = (I_A, O_A, L_A, l_A^0, C_A, T_A) be a TIOA. The regions graph of A is an automata $RG = (\sum_{RG}, S_{RG}, s^0_{RG}, T_{RG})$ *where:*

-- $\sum_{RG} = I_A \cup O_A \cup R^{>0}$

-- $S_{RG} = \{\langle l,[v]\rangle \mid l \in L_A \wedge v \in V(C_A)\}$

-- $s^0_{RG} = \langle l^0_A, [v_0]\rangle$, *where* $v_0(x) = 0$ *for all* $x \in C_A$,

-- *RG has a transition* $s \xrightarrow{\{?,!\}a} s'$, *from* $S = \langle l,[v]\rangle$ *to* $s' = \langle l',[v']\rangle$ *on action {?,!}a iff there is a transition* $l \xrightarrow{\{?,!\}a,G,\lambda} l'$ *such that* $v =\mid G$ *and* $v' = [\lambda := 0]v$.

-- *RG has a delay transition* $s \xrightarrow{d} s'$, *from* $s = \langle l,[v]\rangle$ *to* $s' = \langle l',[v']\rangle$ *on time increment* $d > 0$, *iff* $[v'] = [v + d]$

The regions graph of the TIOA describes the time behaviors of the automaton. The locations of the TIOA are divided by clock valuations. The states of the automaton can be described by two dimensions: where and when (represented by location and clock valuation).

Here an important property of regions graph is that each state of the regions graph has a delay transition labeled with the symbol *d*, which is the time delay between states. The value of *d* is in the interval *]0, 1[*. The number of the states of a simple regions graph for a TIOA could be very large if the number of clocks in the set C_A is a little high. For example, 3 clocks can make the number of regions graph's states rise to almost one thousand. However, we can derive a sub-automaton by sampling regions graph with a delay $\frac{1}{n+1}$, where n is the number of the clocks in the set C_A. The resulting sub-automaton is called Grid Automaton([16][8][15]). Grid automaton is a sampled region graph that has delay transitions labeled with the time delay $\frac{1}{n+1}$. The proof of the existence of the grid automaton is given in [15].

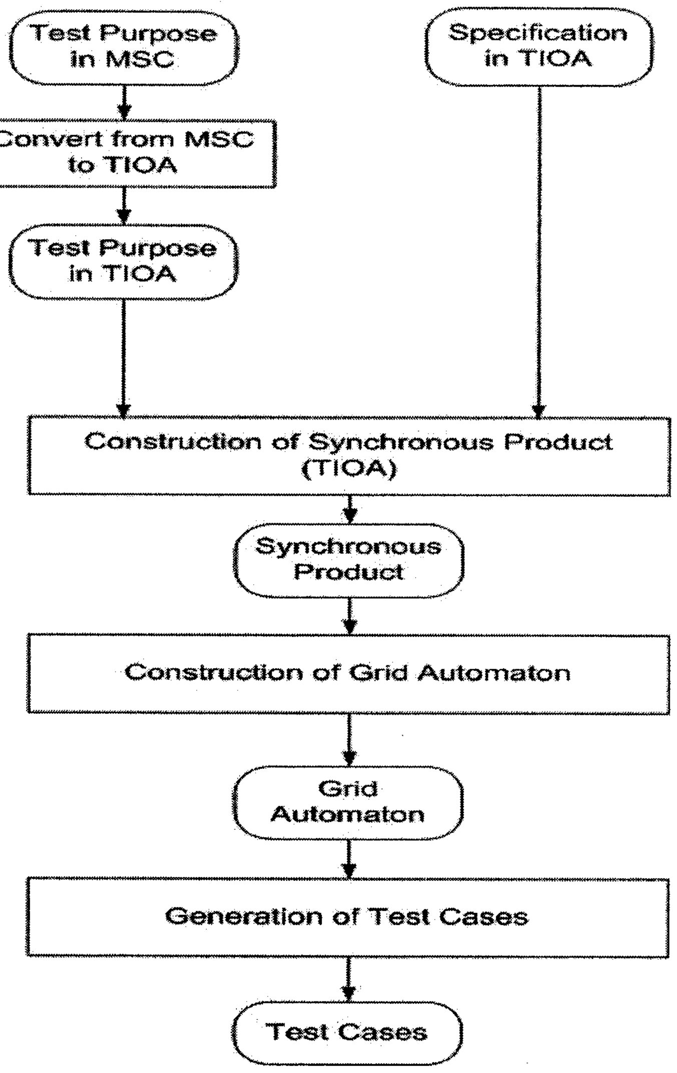

Figure 3-1. Timed Test Case Generation Framework

3.2 Framework

We present a framework for timed test suite generation based on the test purpose expressed in MSC and the specification expressed in TIOA[42]. The Figure 3-1 depicts the framework. There are four main phases distinguished in the process: premium TIOAs generation, synchronous product construction, sampling and test suite generation.

Premise TIOAs generation. At first, the MSC of test purpose and TIOA of specification are given in textual representation. The first step in this phase is parsing the MSC and TIOA text, and transferring them to internal structure representations, so that they can be understood and analyzed by the program. Parsing is done by using Flex and Bison.

Once MSC is transferred to an internal structure, it is converted to a TIOA. The basic idea of converting is that each receiving message in MSC can be translated to an input action in TIOA, and each sending message can be translated to an output action; the state between each pair of exchanging messages can be identified as the location in TIOA. The timer events and the time constraints in MSC can be described by the replacing clocks of TIOA. A critical problem of the resulting TIOA is that the number of clocks could be unnecessary larger than what we need. Therefore a process is needed to minimize the number of clocks for the resulting TIOA.

At the end of this phase, the test purpose is represented also in TIOA, so that the specification and test purpose have the same representation and can be synchronously composed and analyzed.

Synchronous product construction. The implementation must conform not only to the specification but also to the test purpose. The synchronous product represents the requirements of both the specification and test purpose. As in the definition given in the last section, locations, input and output actions and the clocks of synchronous product are the combination of two TIOAs that represent both of test purpose and specification. The detailed algorithm is given in the next section.

Sampling. Once the synchronous product is ready, it is then transferred to a regions graph and the regions graph is sampled by the time granularity. The resulting product is a Grid Automaton. The basic process is as follows: time is regarded as an input of the automata. Then the first thing is to figure out the time delay length, which is made by sampling the time with a fixed granularity. The granularity used here is $\frac{1}{n+1}$, where n is the number of clocks in the TIOA. The initial state of GA is formed with the initial location of synchronous product TIOA, where the clock valuation is set to zero. Next step is decomposing all states reachable from the initial state with repeating $\frac{1}{n+1}$ delay transitions. For each state, we add all its related delay transitions. The same process is repeated starting from the target state of the transitions we just created, until no state left.

Test suite generation. In this last phase, the test suite are derived from the Grid Automaton. Note that the time has been transferred to the label of the transition. So, by traversing the GA, we get test cases with time delay included. We use a depth-first traversal to generate timed test suite. We have different traversal strategies to fulfill variant fault coverage power. We will discuss that in section 3.3.5 and 3.5.

The test suite generated by our method are executable and can be easily represented in TTCN([18]).

The sequence of a test case consists of input actions, time and output actions. When the IUT is tested with a timed test case, a usual activity is that the tester applies the input action, and observes that if the correct output comes from the IUT in a correct period. For example, when the test sequence "*?a, 1/3, 1/3, !b*" is applied to IUT, firstly, "*a*" is input, and after two 1/3 time units, the output "*b*" is expected. To conclude a verdict, if the outputs or final state does not satisfy the specification, the verdict is "fail"; if they satisfy both the specification and test purpose, the verdict is "pass". However, if they satisfy the specification but not the test purpose, the verdict is "inconclusive".

The algorithm for each phase is given in the next section.

3.3 Algorithms

3.3.1 Converting MSC to TIOA

MSC is converted to TIOA, so that a synchronous product, which is represented by TIOA, can be created. In section 3.1, we presented the definition of TIOA and MSC. The detailed description of converting algorithm based on those definitions is introduced as follows.

A SIMPLE MSC is derived from the formal specification of MSC-2000 by simplifying the grammar of MSC-2000. The aim of simplified MSC is to provide a base for describing the scenarios of test purposes, while not bothered by heavy and complicated syntax. The SIMPLE MSC is derived based on the following assumptions:

(1) We assume that the Implementation Under Test is a process instance in MSC, and it is deterministic, which means that all actions (events) are deterministic. Other instances communicating with the instance under test are regarded as test environment.

(2) All local actions, except timer events, are ignored in our algorithm.

(3) We only consider Receiving and Sending Message events and timer events. A timer can be started, stopped, and time-out. We initialize a clock when a timer is started.

(4) Time constraint can also be expressed by time interval syntax in MSC, which follows the event it's measuring. The syntax is like:
[alias name] <message event> [time <time interval> <alias name>]. Time intervals can be defined for any two events within an MSC document. The textual representation of an MSC indicates the potential pairs of events only. Such a pair of event is indicated by connecting the interval boundaries of two events. The algorithm following also concerns this kind of time interval.

To translate an SIMPLE MSC $M = (P, S, R, A, O, T, T_X, Y)$ to a TIOA $A = (I, O, L, l^0, C, T)$, the exchanging message events can be processed as follows: Received messages set R and sent messages set S can be renamed as TIOA input I and output O actions sets respectively.

Then, we can create an initial location l^0 before any event happens and a final location *lf* after all events finish in MSC. Next, for each event $e \in (S \cup R)$, we create a corresponding location l'. It means that after the event *e* happens, the process reaches the state l'. All of those l' form the set *L* in TIOA. Then, we examine the set *O* starting from location *l0*. Every event *e* included in the set *O* or *S* or *R* gives rise to a transition *t*. It is repeated until we arrive at the end of the instance. The *T* set of TIOA is formed by those *t*.

Next, we rename the timer set T in MSC to clock set C in TIOA. From T_X we know the timer related events. And again we scan the set O to determine the correspondence of timer events and the transitions. For each start-timer event, we create a location l', and a transition t from last event's corresponding location l to l'. The corresponding clock is set to zero at transition t (Note that at the initial location l^0, all clocks are set to zero). When a corresponding time-out event is reached, a location lx is created. For each event between the start-timer and corresponding time-out events, a transition tx arriving at lx is created for the corresponding location. We then label each tx with clock guard by considering timer's value. With the set Y, the time constraint on a single event or a sequence of events is also transformed to a clock in TIOA. The transition of each event within the time constraint is labeled with the corresponding clock guard.

The pseudo code is the following:

Input: *MSC* $M = (P, S, R, A, O, T, T_X, Y)$

Output: *TIOA* $A = (I, O, L, l^0, C, T)$

Definitions of Variables:

E : the set of all events in the instance of MSC;

Et : the set of all timer related events;

Ets : timer setting event;

Eto : timer time-out event;

Etp : timer stop event;

Label(e): the message event e 's parameter.

Last(L) : last created location in L;

Tc = (tmax, tmin) : the time constraint.

Y={(es,ee,tc)| es is the starting event, ee is end event, tc is the time constraint}: the association of two events and a time constraint.

TCL : the set of the transitions within time constraints.

TRL : the set of the locations under the monitoring of timer.

STR : the set of starting timers;

STEP0: Create Initial location

```
Create initial location $l_0$;
Add all event e to E in the deterministic ordering;
Add all timer related events to Et;
STR $\leftarrow$0;
```

STEP1: Construct transitions and locations

```
While E != 0 do
    read the next event $e \in E$ ;
    if ( the event $e \in R$ or $e \in S$ ) do
        create a location $l$ and a location $l'$;
        add $l$ to L;
        add $l'$ to L;
        create a transition t from $l$ to $l'$ ;
        create a reset clocks set $\lambda$ for t, $\lambda \leftarrow 0$;
        create a guard G for t, G$\leftarrow$0;
        label the transition t with label(e);
        add t to T;
end if

if (e == y.es) do
        create a clock c and reset clock v(c): = 0 ;
        add c to $\lambda$;
        Create a set TCLy $\leftarrow$0;
end if

if (e == y.ee) do
        for (each transition $t y \in TCLy$) do
                add $v(c) > y.tc.t_{\min}$ and $v(c) < y.tc.t_{\max}$ to G
        end for
        delete TCLy ;
end if

if (e $\in$ Et and e == ets) do
        create a location l ;
        add l to L;
        create a transition t from last(L) to l;
        label the transition t with $\Phi$;
        create a reset clocks set $\lambda$ for t;
        create a clock c, $v(c):=0$;
        add c to $\lambda$ and C;
        add timer rt to STR;
        add l to TRL;
        continue;
end if

if ($e \in Et$ and $e == eto$) do
        Create a location $l_x$;
        Add $l_x$ to L;
        For (each location $l \in TRL$) do
                Create a transition t from l to $l_x$;
                Add t to T;
                Label the transition t with "!timeout";
                create a clocks guard set G for t;
                add $v(c) > v(rt)$ to G;
        End for
        $STR := STR \backslash rt$;
        Delete TRL;
        Continue;
end if

If (STR != 0) do
```

```
        For each ( $rt \in STR$ ) do
            Add $v(c) < v(rt)$ to G of last(T);
        End for
    End if

    $E:=E \setminus e$;
End while
```

3.3.2 Clock Minimizing

A main problem in timed test suite generation is that timed automaton could be exponential with high number of clocks. One of the reasons of causing extra clocks is that the specification is usually written in high level language and later translated to timed automaton, the different timer events and time variables appear in the original language are translated to different clocks in timed automaton. However, these clocks are rarely active at the same time. Hence, the number of clocks can be reduced.

Another potential chance to reduce number of clocks is based on the fact that some clocks are simultaneously reset and therefore they are equal at some locations, since they all proceed at the same speed rate.

Based on these two situations, a method is proposed in [4] to reduce the number of clocks by combining two algorithms. The first algorithm is to detect active clocks: Since in some locations some clocks are never active, those clocks can be ignored. The second algorithm is to detect equal clocks: in the locations where equal clocks can be regarded as the same clock, the repeated clocks can be eliminated. After each of those detection, a renaming follows so that clocks are renamed in a new way by which the number of clocks decreases.

We will present the Renaming function at first, followed by Reducing Active Clocks Algorithm and Reducing Equal Clocks Algorithm. The Renaming function is used as a sub-function for the later two algorithms.

(1) Renaming function

Definitions of Variables:

CO: the set of clocks in original TIOA;

CR: the set of clocks replacing CO;

$cr(x,l)$ *: the clock replacing the clock x at the location l;*

$v(x)$ *: the valuation of clock x;*

cat(l): the set of clocks that appear in the guard G of the outgoing transition from location l;

Definition of Rule Renaming:

Rule(t) is defined for transition $t = l \xrightarrow{\{?,!\}a,G,\lambda} l'$ *:*

For $x, y \in CO$, *if* $cr(x,l') == cr(y,l')$ *then it is required that*

$x == y$ *or* $x, y \in \lambda$.

Definition of Renaming function Rename(CR):

for each (transition $t = l \xrightarrow{\{?,!\}a,G,\lambda} l'$*) do*

replace each $x \in cat(l)$ *with* $z = cr(x,l)$ *by following Rule(t);*

```
        for each ( $z \in CR$ ) do
            if ( $cr(y,l') == z$ ) do
                if ( $y \in \lambda$ ) do
                    set z to zero, z:=0;
                else
                    assign $v(cr(y,l'))$ to z, := $v(cr(y,l'))$;
                end if
            end if
        end for
    end for
```

(2) Reducing Active Clocks Algorithm

The algorithm of reducing active clocks is described as follows:

Definitions of Variables:

Act(l) : the set of the active clocks at location l;

AL : the set of all locations that own active clocks;

Cat(l) : the set of clocks that appear in the guard G of the outgoing transition from location l;

L : the set of the locations of the TIOA;

Maxact(AL): the largest number of active clocks in all locations;

STEP1: Initialization

```
For (all $l \in L$) do
    $act(l) := cat(l)$;
    if $act(l) \neq 0$ then
        add $l$ to AL;
    end if
end for
```

STEP2: Detecting active clocks

```
For (each $l \in AL$) do
    For (each outgoing transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$) do
        For (each clock $x \in act(l')$) do
            If $x \notin \lambda$ and $x \notin act(l)$ do
                Add $x$ to $act(l)$
            end if
        End for
    End for
End for
```

STEP3: Clocks Renaming

```
Calculate Maxact(AL);
Create renaming clocks set CR which owns Maxact(AL) clocks;
Apply renaming function Rename(CR) to TIOA;
```

(3) Reducing Equal Clock Algorithm

Definitions of Variables:

Equ(l) : the set of pairs of equal clocks at location l;

Maxequ(L) : the max number of equal pairs in all locations;

STEP1: Initialization

```
For (each $l \in L$) do
    $equ(l) := L \times L$;
end for
```

STEP2: Detecting Equal Clocks Pair

```
For (each transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$) do
    For (each pair $(x, y) \in equ(l')$) do
        If ($x \in \lambda$ and $y \notin \lambda$) then
            $equ(l') := equ(l') \setminus (x, y)$;
        end if
    end for
end for
For (each transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$) do
    For (each pair $(x, y) \in equ(l')$) do
        If ($(x, y) \notin equ(l)$ and $x, y \notin \lambda$) then
            $equ(l') := equ(l') \setminus (x, y)$;
        end if
    end for
end for
```

STEP3: Renaming Clocks

Calculate the Maxequ;

Create clock set CR which owns Maxequ clocks;

Renaming equal clock pair with the same clock by applying Rename(CR);

3.3.3 Synchronous Product Construction

When we test the implementation against test purpose, we also have to verify if the implementation satisfies the specification. The Synchronous Product represents both the requirements from the test purpose and specification. In our algorithm of Synchronous Product Construction, the inputs are two TIOAs, the output is a single TIOA as synchronous product:

INPUT: *A specification TIOA* $S = (I_S, O_S, L_S, l^0_S, C_S, T_S)$

A test purpose TIOA $TP = (I_{TP}, O_{TP}, L_{TP}, l^0_{TP}, C_{TP}, T_{TP})$

OUTPUT: *A synchronous product TIOA* $SP = (I_{SP}, O_{SP}, L_{SP}, l^0_{SP}, C_{SP}, T_{SP})$

Definitions of Variables:

RL is the set of reachable locations;

HL is the set of handled locations;

STEP1: Initialization

```
$l^0_{SP} \leftarrow (l^0_S, l^0_{TP})$.
Add $l^0_{SP}$ to $L_{SP}$.
$C_{SP} \leftarrow (C_S \cup C_{TP})$.
$RL \leftarrow l^0_{SP}$
$HL \leftarrow 0$
```

STEP2: Construct the Locations and Transitions of SP

```
While ($RL \backslash HL$ != 0)
    Get a location $l = (l_1, l_2)$ from $RL \backslash HL$.
    Add $l$ to $HL$.
    If ($l_1 \xrightarrow{\{?,!\}a,G_1,\lambda_1}_{S} l_1' \in T_S$ and $l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}_{TP} l_2' \notin T_{TP}$) then
        Add $(l'_1, l_2)$ to RL.
        Add $(l_1,l_2) \xrightarrow{\{?,!\}a,G_1,\lambda_1}_{SP} (l_1',l_2)$ to $T_{SP}$.
    End if
    If $l_1 \xrightarrow{\{?,!\}a,G_1,\lambda_1}_{S} l_1' \in T_S$ and $l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}_{TP} l_2' \in T_{TP}$ then
        Add $(l'_1, l'_2)$ to RL.
        Add $(l_1,l_2) \xrightarrow{\{?,!\}a,G_1\&G_2,\lambda_1\cup\lambda_2}_{SP} (l_1',l_2')$ to $T_{SP}$.
    end if
end while
```

Two steps are identified in this algorithm. The first step is initializing SP. The initial location of SP is constructed by combining the initial locations of the specification and test purpose TIOAs. The clock set of SP is the union of the clock sets of the specification and test purpose TIOAs.

The second step is to construct the locations and transitions of SP. Two sets are used to process the algorithm: the *RL* set stores the SP locations that are reached, and the *HL* set stores the locations that are already being handled. The *RL* set is initialized with the SP initial location l^0_{SP}, and *HL* set is empty. To construct the Synchronous Product, one of the locations in *RL* but not in *HL* is derived to create more locations and transitions, and then put in *HL*. This procedure is repeated until the *RL* and *HL* have no difference. The starting point is the initial SP location. The following two rules are used to create SP's transitions and locations:

(1) $l_1, l_1' \in L_A$, and $l_2, l_2' \in L_{TP}$, if the transition $l_1 \xrightarrow{\{?,!\}a,G_1,\lambda_1}{}_A l_1'$ exists in T_A and transition $l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}{}_A l_2'$ doesn't exist in T_{TP}, then there exists a transition $(l_1,l_2) \xrightarrow{\{?,!\}a,G_1,\lambda_1}{}_A (l_1',l_2)$ in T_{SP}.

(2) However, if $l_2 \xrightarrow{\{?,!\}a,G_2,\lambda_2}{}_A l_2'$ does exists in T_{TP}, then a transition $(l_1,l_2) \xrightarrow{\{?,!\}a,G_1\&G_2,\lambda_1\cup\lambda_2}{}_A (l_1',l_2')$ exists in T_{SP}. the result SP transition's guard is the union of G_1 and G_2 from the *A* and *TP* respectively, as well as the clock reset.

3.3.4 Sampling - Deriving Grid Automaton

A Grid Automaton (GA) is created from Synchronous Product. The system under test stays in a state for a finite or infinite time period. To test whether the system obeys the time constraint or not, we try to observe the time period during which the system is in a particular state by dividing the state with a delay clock value, so that when we generate test cases, the clock value can be obtained. The Regions Graph of Synchronous Product provides operational semantics. The GA is a sub-automaton of the Regions Graph; it consists of a chosen set of representatives for each state of the Regions Graph and accordingly instantiate the delay transition *d*.

The algorithm to construct a GA by sampling the Synchronous Product can be summarized as follows: First, the granularity value is calculated as $\frac{1}{K+1}$, where *k* is the number of clocks in TIOA. Then, an initial state is created with the initial location of input TIOA. All the clocks are set to zero at initial state. Finally, we create all reachable state from this initial state with repetitive $\frac{1}{K+1}$ delay transition. From each reachable state *(l, v)*, we create transition $(l,v)\xrightarrow{\{?,!\}a}(l',[\lambda := 0]v)$ for each transition $l\xrightarrow{\{?,!\}a,G,\lambda}l'$ in TIOA, if *v* satisfies *G*. Afterwards, we repeat the same process until all reachable states are covered.

INPUT: *A synchronous product* $SP=(I_{SP}, O_{SP}, L_{SP}, l^0_{SP}, C_{SP}, T_{SP})$.

OUTPUT: *A sub automata GA*

Definitions of Variables:

RS: the set of reachable states

HS: the set of handled states

granularity: the granularity value;

STEP1: Initialize the Variables

```
$s^0 \leftarrow (l^0_{SP}, 0)$

$RS \leftarrow l^0_{SP}$

$HS \leftarrow 0$

$granularity \leftarrow \frac{1}{K+1}$
```

STEP2: Creating the GA States and Transitions

```
While ($RS \backslash HS \neq 0$) do
    Get a state $s = (l, v)$   from RS/HS
    Add s to HS
    For (each transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$ in TIOA) do
        If ($v \models G$) then
            add $(l,v) \xrightarrow{\{?,!\}a} (l',[\lambda := 0]v)$ to GA if it does not exist; Add ($l'$, $[\lambda:=0]v$) to RS if it does not exist
        end if
    end for
    Add $(l,v) \xrightarrow{granularity} (l, v+granularity)$ to GA if it does not exist;
    Add ($l$, $v$+granularity) to RS if it does not exist
end while
```

3.3.5 Traversal

After the Regions Graph of Synchronous Product is sampled, a sub-automaton (GA) is derived. By traversing the resulting GA, the test cases can be obtained. There are many traversal algorithms with variant fault coverage power. They are described in section 3.5. Here, we provide two traversal algorithms: all-paths coverage traversal algorithm and all-states coverage traversal algorithm.

(1) All-paths coverage traversal algorithm. This traversal algorithm guarantees that all paths are covered. It also implies that all states and transitions of GA are covered. The algorithm begins with the GA as input, and ends with a set of test cases as outputs. The first step is initializing all the variables. *VS* is the set of visited states, *TC* is the set of test case, *NS* is the set of neighbor states of the being visited state, S_{SP} set stores all states of GA.

In the second step, a state is chosen from S_{SP} (at the beginning, the initial state is chosen). The chosen state *s* is then put into *VS*, indicating that it has been visited. Then, a set *NS* is created and all the neighbors of s are put into *NS*. For each neighbor of *s* in *NS*, the transition's label (it could be an input/output action *a*, or a time delay *d*) is concatenated with *TC*. If a state without outgoing transition is reached, a test case rises from all passed transition in *TC*. This procedure is executed recursively until all the paths have been covered.

INPUT: *A sub-automaton of the region graph of SP.*

OUTPUT: *A set of timed test cases.*

Definitions of Variables:

VS: the set of visited states

TC: the set of test cases

NS: the set of neighbor states

S_{SP}: the set of GA states

STEP1: Initialization

```
VS ← 0
TC ← 0
NS ← 0
S_SP ← all states of GA
```

STEP2: Traversing the GA

```
While ($S_{SP}$ \ VS != 0)
    /* at the first time, the initial state is chosen */
    Choose a state s from $S_{SP}$ \ VS ;
    Add s to VS;
    NS ← all the neighbors of s;
    While (NS $\neq$ 0)
        Choose and remove a state s1 from NS such that $s \xrightarrow{\{?,!\}a} s_1 \in T_{GA}$.
        Concatenate {?,!}a with TC.
        Add all the neighbors of s1 to NS.
        If (s1 has no outgoing transition) then
```

```
                    Print TC.
                    TC ← TC\{?,!}a.
                end if
        end while
end while
```

(2) All-states coverage traversal algorithm. This algorithm guarantees that all states of GA are covered. The difference from all-path coverage algorithm is that this algorithm doesn't cover all transitions paths, therefore it has much less coverage power. The advantage is that this algorithm generates a smaller number of test cases.

The algorithm is relatively simple. The first step is to initialize all the variables. The second step begins from initial state s0. The deep-first strategy is applied: a neighbor is chosen and the transition label is added to the test case. This is recursively repeated until an already visited state or no-outgoing-transition state is reached, and gives rise to a new test case. Then, the next neighbor state is chosen; the same process runs again until all states are visited.

INPUT: *A sub-automaton of the region graph of SP.*

OUTPUT: *A set of timed test cases.*

Definitions of Variables:

VS: the set of visited states

TC: the set of test cases

```
NS(s): the set of neighbor states of s
$S_{SP}$: the set of GA states

STEP1: Initialization
VS ← 0
TC ← 0
NS(s0) ← 0

STEP2: Traversing the GA
Choose initial state s0 from $S_{SP}$;
Add s0 to VS;
NS ← NS(s0);
While (NS $\neq$ 0)
    Choose and remove a state s1 from NS such that $s \xrightarrow{\{?,!\}a} s_1 \in T_{GA}$
    if ($S_{SP}$ \ VS == 0)
        return;
    end if
    Concatenate {?,!}a with TC;
    If (s1 has no outgoing transition or $s1 \in VS$) then
        Print TC.
        TC ← TC\{?,!}a.
        Continue;
    else
```

```
        Add s1 to VS;
        NS ← NS(s1);
        Go to while loop to do the recursion;
        TC ← TC\{?,!}a.
    end if
end while
```

3.4 Time-Related Fault Model

When testing real time systems, comparing to non-time systems, the extra task is to identify the time-related faults. In our method, the regions graph not only represents the system functional behaviors, but also provides an intuitive means to identify the possible time-related faults that the implementation can have. Besides the faults which could exist in the non-time related state machines (the output faults, transition faults, additional and missing transitions, and additional states faults, which we have described in Section 2.3), the extra time-related faults are given in [8]. Four types of time-related faults are identified: clock reset faults, time constraint restricted faults and time constraint widening faults. We give a simple description for each of them as follows:

Clock reset faults. There are two kinds of clock reset faults. When the implementation does not reset a clock that is reset in the specification, and the implementation reset a clock that is not reset in the specification. The reset faults change the order of clocks. When the clock is reset in the implementation while it is not allowed in the specification, the number of states of the implementation regions graph will increase, therefore some extra states can be reached. When the clock is not reset, but it is required to be reset in the specification, the number of states of the regions graph of the implementation will decrease.

Time constraint restricted faults. An implementation is said to be faulty if it refuses to receive an input under the time constraints specified by the specification. This fault will lead to reducing the number of the regions graph's states. However, if the implementation restricts the time constraint of the output, it is not considered as a fault, since the output is controlled by the implementation, and the restricted time constraint still falls in the interval of the time constraint given in the specification.

Time constraint widening faults. An implementation can have time constraint widening faults by increasing the upper bound or/and decreasing the lower bound of the time constraint given in the specification. Four types of these faults can occur: first, the transition is fired at the time greater than the fixed clock value given by the specification. Second, the transition is fired at the time smaller than the fixed clock value given by the specification. Third, the transition is fired at the time greater than the upper bound of the time constraint required by the specification; finally, the transition is fired at the time lower than the lower bound of the time constraint required by the specification. The widening faults leads to larger number of the implementation regions graph states than the one of the specification regions graph.

3.5 Fault Coverage

In section 2.6, we introduced various test case derivation methods. Those methods can also be applied as traversal algorithms, as what it is done in [8]. We try to cover all the paths of the grid automata in the traversal algorithm, which we presented in section

3.3. By covering all the paths, the number of test cases could be very high. Sometimes it is impossible to test every possible path. The solution depends on how powerful the fault coverage is needed. There are various coverage strategies with different coverage power. The criteria is described as follows:

(1) **Coverage of all paths**. Every path in grid automata is covered, this criteria has most powerful fault coverage because all possible situations are examined by test cases. The negative side is that the number of test cases is too high, so it is sometimes impossible to examine every path.

(2) **Coverage of all transitions**. This has less power of fault coverage than all paths coverage, because the paths traversed are the subgroup of (1), not every possible situation is examined by the resulting test cases.

(3) **Coverage of all states**. All states of GA are covered by the test cases. Again, this has less power of fault coverage than (1) and (2). Only states are covered, it is not guaranteed that all transitions are covered.

(4) **Coverage of paths to get all verdicts PASS**. This focuses on the paths to get verdict PASS, the verdicts FAIL and INCONCLUSIVE are ignored. The number of generated test cases could still be very high, and it is not guaranteed that every state can be covered. This method can be applied when the aim of test purpose is only to verify if the system under test can properly execute the normal tasks.

(5) **Coverage of only one path to get one verdict PASS**. When the test purpose is only to examine one particular property of the system, one path is chosen to be covered, so that a PASS verdict can be reached. This approach generates only one test case and has the least power of fault coverage.

(6) **Coverage of paths to get all verdicts FAIL or INCONCLUSIVE**. Only paths leading to FAIL or INCONCLUSIVE verdict are covered. It has the same power as (4).

(7) **Coverage of only one path to get verdict FAIL or INCONCOUSIVE**. Like (5), only one path is chosen to examine the situation when system under tested is fail or in an inconclusive state.

3.6 Conclusion

We provided a new method for timed test suite generation. This method is based on test purpose expressed in MSC and specification expressed in TIOA. The four main phases in the framework of this method are discussed. We also gave the detailed algorithms for each phase. The fault coverage criteria related to traversal algorithm is also discussed, as well as the timed fault model.

In next chapter, we will provide the detailed implementation of the algorithms of this method.

CHAPTER 4

Implementation of Algorithms

We have introduced the framework of our method for timed test suite generation based on test purpose expressed in MSC and specification expressed in TIOA. We have implemented this framework with C++ programming language. In this chapter, the design of the implementation is presented, and several examples are given to illustrate how our method works. More specifically, Section 4.1 provides the information about the development tools we used for developing the program. Section 4.2 presents the UML activities diagram, which depicts how the program is constructed internally. Section 4.3 presents the UML class diagrams. An example is given in Section 4.4.

4.1 Development Tools and Environment

We used object-oriented techniques to realize the implementation of our method. The program is designed with the Unified Model Language (UML)([23] [24]). The logical processing is depicted by the Activities Diagrams. The identified classes and their relations are described with Class Diagrams. The program is written in C++ language, which is a popular object-oriented language. Most of data structure operations utilize the Standard Template Library (STL), which now is a standard library included in formal C++ foundation libraries.

4.1.1 Bison and Flex

A specific parser is used to parse the source code in a specific language. We used the tools Bison and Flex to construct the parsers of MSC and TIOA languages.

Flex is a tool of generating programs that recognize the lexical text, so called scanner. The elements of the parsed language are depicted in regular expression language. Flex can recognize those elements and generate C programs by reading the pattern expression written in a lex file.

Bison is a grammar parser generator. It can convert a language grammar description into a C program that is used to parse the source file of that language. Usually Bison and Flex are used together. A lex file, which describes the language elements, and a yacc file, which describes the grammar, are needed by Flex and Bison respectively to generate program that parses the source code of a given language. Bison and Flex are the advanced tools that succeed the old popular parser generation tools yacc and lex.

Appendix C provides SIMPLE MSC lex and yacc files. TIOA lex and yacc files are provided in Appendix D. The grammar of SIMPLE MSC and TIOA are provided in Appendix A and Appendix B, respectively. We can see that the grammars described in yacc files of SIMPLE MSC and TIOA are the same as what they are in Appendices A and B.

Lex and yacc files are applied on Flex and Bison tools, and transformed to the language parser as *C* source code. For our program, two text files are needed as inputs, the test purpose file in MSC and the specification file in TIOA. The parsers generated by Flex and Bison parse these two files and transform them to internal *MSC* and *TIOA* classes (which are given in section 4.3) respectively. As an example, in section 4.4 we will provide the MSC and TIOA files for the test purpose and the specification of a telephone system.

The source code of the implementation is developed for Linux and Solaris operating systems. In the next section, we use activities diagram to describe the internal structure of the implementation program.

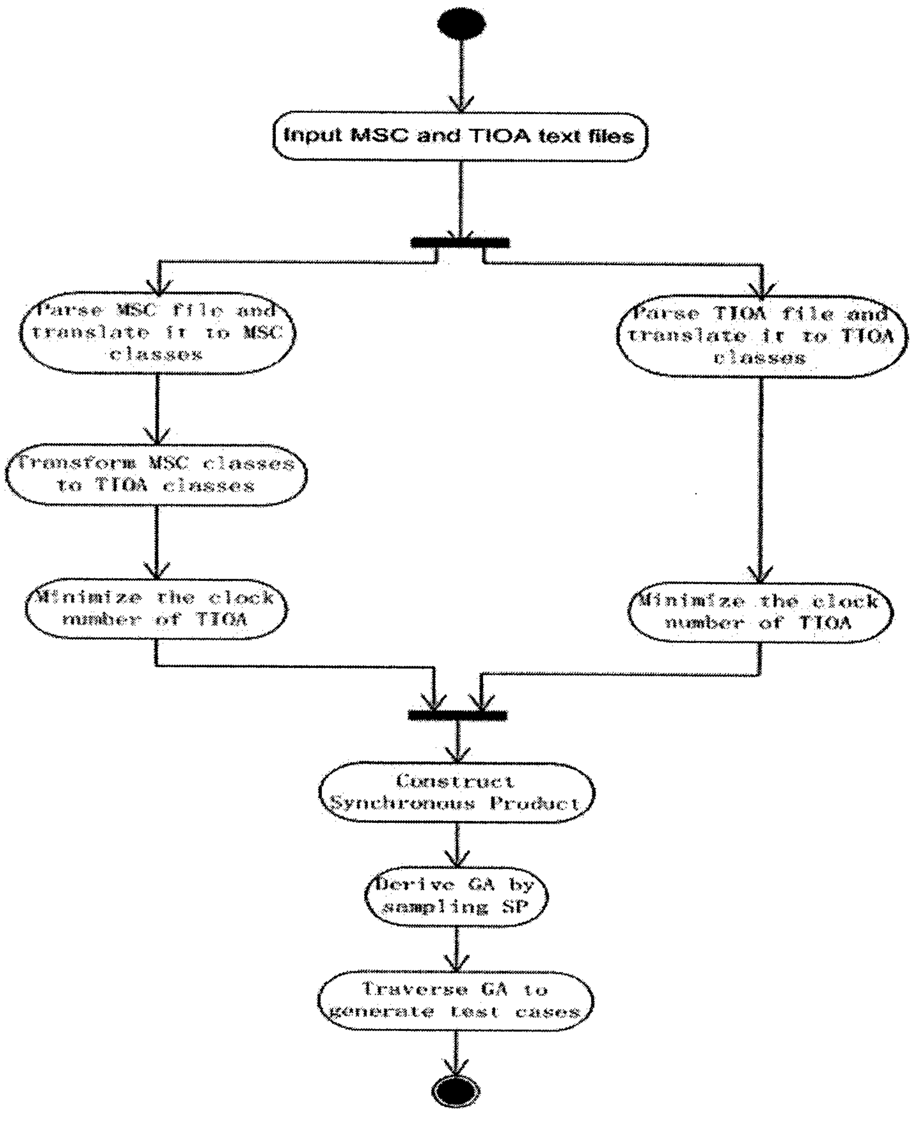
Input MSC and TIOA text files
Parse MSC file and translate it to MSC classes
Parse TIOA file and translate it to TIOA classes
Transform MSC classes to TIOA classes
Minimize the clock number of TIOA
Minimize the clock number of TIOA
Construct Synchronous Product
Derive GA by sampling SP
Traverse GA to generate test cases

Figure 4-1 Activity Diagram

4.2 Activities Diagrams

Figure 4-1 describes the activities of the program. It is derived from the framework we gave in chapter 3. MSC and TIOA text files are read by the program, and then transformed to the internal structures (C++ class). The structure representing MSC is then converted to TIOA. The two TIOAs are filtered by a clock minimizing algorithm to reduce the number of clocks. Then, they are merged and a synchronous product is constructed. It is followed by a sampling algorithm, which samples the synchronous product and results in a Grid Automaton. Finally, test suite are generated by traversing the Grid Automata.

The final output is a file depicting the generated test sequences. Appendix E gives the output test cases for the example of a telephone system introduced in Section 4.4.

4.3 Class Diagrams

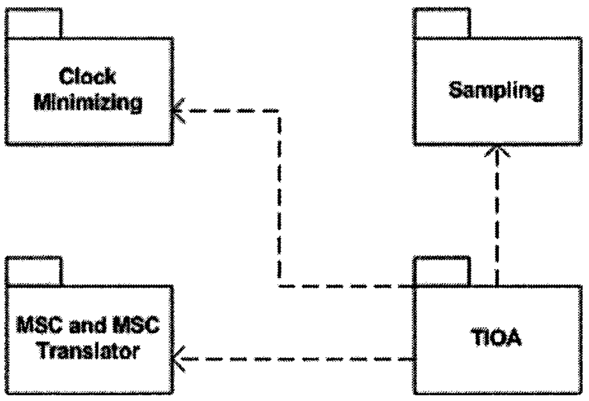

Figure 4-2. Package Diagram

As it is depicted by package diagram in Figure 4-2, the classes are organized as four packages: TIOA, MSC and MSC Translator, Clock Minimizing, and Sampling. The

Class Diagrams depict the structure of the classes and their relations. The following is the class diagrams for each of these packages.

4.3.1 TIOA Classes

Figure 4-3 depicts the TIOA classes hierarchy, which represents the internal structure of the Timed Input Output Automaton. The TIOA classes are mainly composed of three top classes: *TIOA, Location, Transition*. *TIOA* represents the automata, it is composed of *locations* of type *Location*. The transitions are included in the *locations* as the outgoing transitions of type *Transition*. For each of these three top classes, two sub-classes are created to represent the elements of Timed Input Output Automaton and Grid Automaton respectively. The detailed description is as follows:

TIOA class represents the whole TIOA, it has two child classes:

- **TIOA_States**: this class is for Timed Input Output Automaton; it has a list of type *Locations* member, which includes all the locations of the TIOA. The function *constructSynchProd* realizes the synchronous product construction algorithm.
- **TIOA_grid**: this class is for Grid Automaton. It has a list of *TIOAG_Lcation* member, which includes all the Grid Automaton's locations. The function *traverse* realizes the traversal algorithm.

Location class represents the locations in an automaton; it also has two child classes:

- **TIOA_Location:** this class represents the TIOA locations. It includes the members of a location name and a list of outgoing transitions.
- **TIOAG_Location**: this class represents Grid Automaton states. It has a list of clock valuation member, which represents the clocks values when the machine is in the given state. A list of outgoing transitions is also included in this class.

Transition class represents the transitions in an automaton. It has two child classes representing the transition of TIOA and GA respectively. The *Label* member in Transition class represents the input or output actions, or the GA's time delays. The two child classes of Transition are:

- **TIOA_Transition:** This class is for the transitions of TIOA. It includes the time guards and reset clocks members. Time guards are represented by the instances of class *TIOA_Constraint*. Reset clocks are represented by the list of the instances of class *Clock*. The class *Clock* represents the clocks in TIOA, its lower bound is 0 and upper bound is the value assigned in the member *domain*. The member *destination* is the transition's destination location in TIOA.
- **TIOAG_Transition:** This class is for the transitions of Grid Automaton. Because time guards and reset clocks are converted to locations and time delays in GA, there is no time constraint and clock in GA. The transition's destination state is represented by the member *destination*.

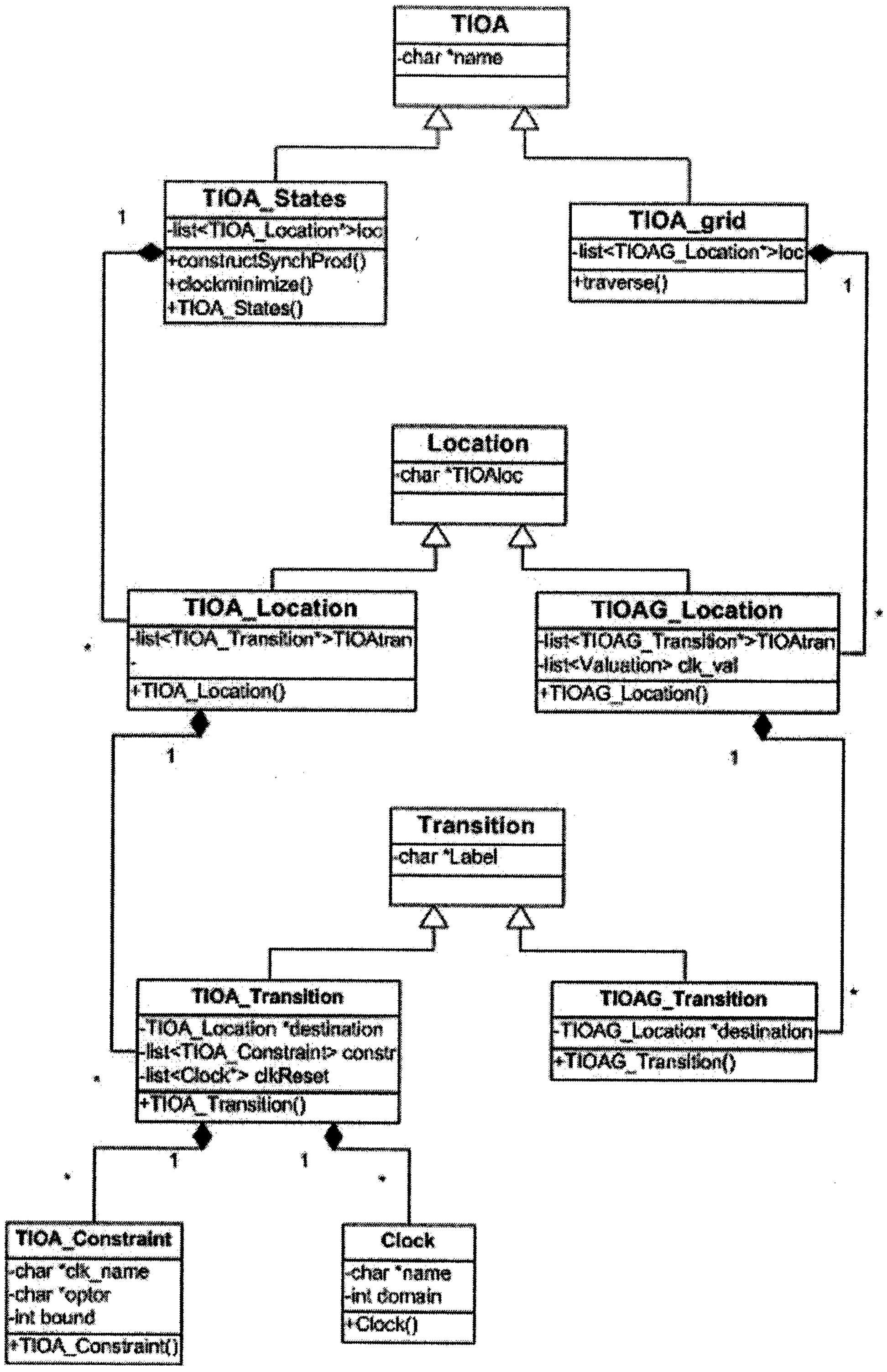

Figure 4-3. Class Diagram of TIOA

4.3.2 MSC and MSC Translator Classes

Figure 4-4 is the class diagram of MSC. The text MSC is transformed to an internal structure, which is represented by class *Instance* in our design. As we have described in section 3.1, the instance of MSC represents a process. The class *Instance* here is the same as what it means in MSC. The *Instance* class is composed of event list and time constraints. The events happened in MSC are classified into two categories: *MsgEvent* and *TimerEvent*. *MsgEvent* represents the message exchanging between the instance and environment. *TimerEvent* represents timer events. The time intervals and time constraints are identified as *TimeConstr* class, the member *beginevent* and *endevent* indicate the range of time interval.

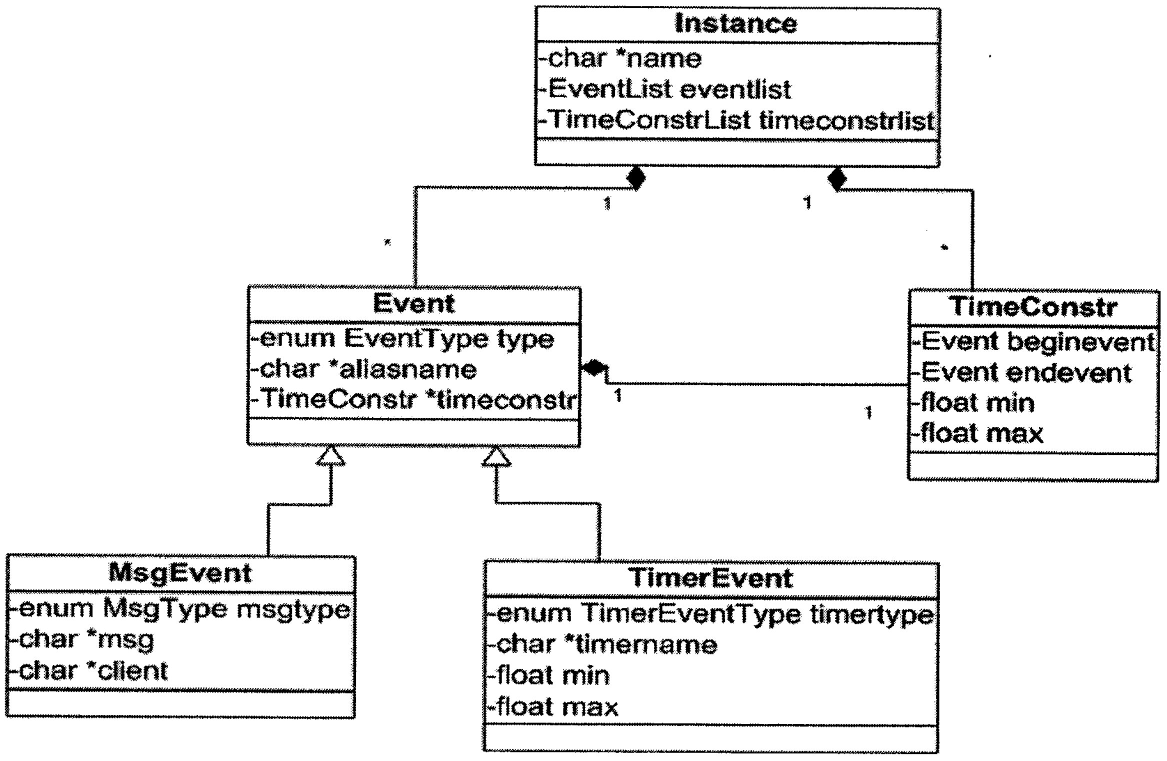

Figure 4-4. Classes of MSC

A class *MSCTranslator* is designed to realize the MSC-TIOA converting algorithm as it is depicted in Figure 4-5. The function *translate* translates the instance of *Instance* into an instance of *TIOA_States*. This class decouples the link between MSC and TIOA classes.

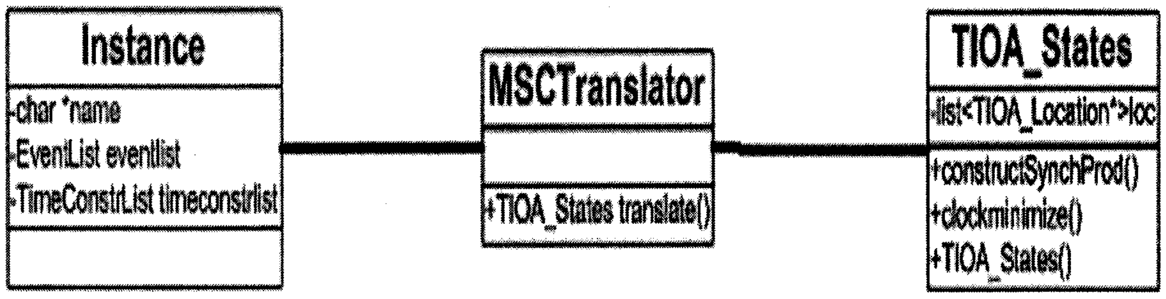

Figure 4-5. Classes of MSC Translator

4.3.3 Clock Minimizing Classes

As it is depicted in Figure 4-6, three classes are designed to do the job of minimizing TIOA clock, which are used by TIOA member function *minimizing*. The *ClockMinimizer* class has two subclasses to realize the clock minimizing algorithm:

- **ActiveMinimizer** realizes the active clocks reduction algorithm.
- **EquMinimizer** realizes the equal clocks reduction algorithm.

The static function *Minimizing* in class *ClockMinimizer* reads an instance of *TIOA_States* as input parameter, then uses the functions in *ActiveMinimizer* and *EquMinimizer* to reduce the number of clocks in TIOA.

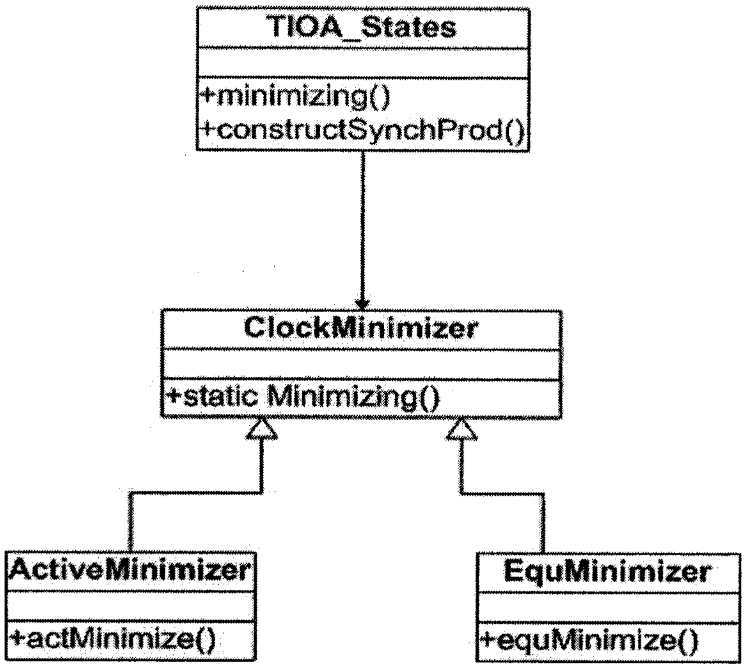

Figure 4-6. Class Diagram of ClockMinimize

4.3.4 Sampling Class

A class *GACreator* is designed to create Grid Automaton by sampling Synchronous Product TIOA. *GACreator* has one static function *sampling*, which realizes the sampling algorithm. It reads an instance of *TIOA_States* as input parameter and writes an instance of *TIOA_grid* as return. Figure 4-7 represents their relationship. The reason to create a standalone sampling class is to reduce the coupling between *TIOA_States* and *TIOA_grid.*

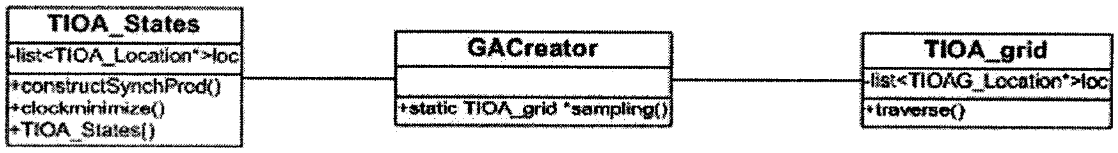

Figure 4-7 Class Diagram of GACreator

4.4 Case Studies

In this section, we will provide several examples on which our method is applied. These examples with different clock numbers, state space and test purposes illustrate how our method works and what elements have impacts on the resulting test cases. We also look at other works related to timed test suite generation, and compare them to our method.

4.4.1 An Detailed Example

This example is a telephone system ([3]), which accepts two digits dial and issues a connection. The specification and test purpose are given as TIOA and MSC respectively. The resulting test cases of applying our implementation on this example are provided in Appendix E.

4.4.1.1 Specification

Figure 4-8 is a graphic TIOA specification of the telephone system. The user of this system hangs off the phone and composes two digits, then the connection is setup by the system. The system has time constraints for every actions made by the user. When the user hangs off the phone, the first digit must be input within one time unit, and the second digit has to be issued also within one time unit after the first digit is input. Any action that fails to meet the time constraints will cause an error output and the clock is reset to zero, and the system is set to the initial state. After the two digits are input successfully, the system has to issue a "Connect" in two time units, then it is in the conversation state until the user hangs on the phone and the system goes back to the initial state. When the system enters the initial state, the clock is set to zero and the system is waiting for the user hanging off the phone. Once the phone is hung off, the clock is again reset to zero.

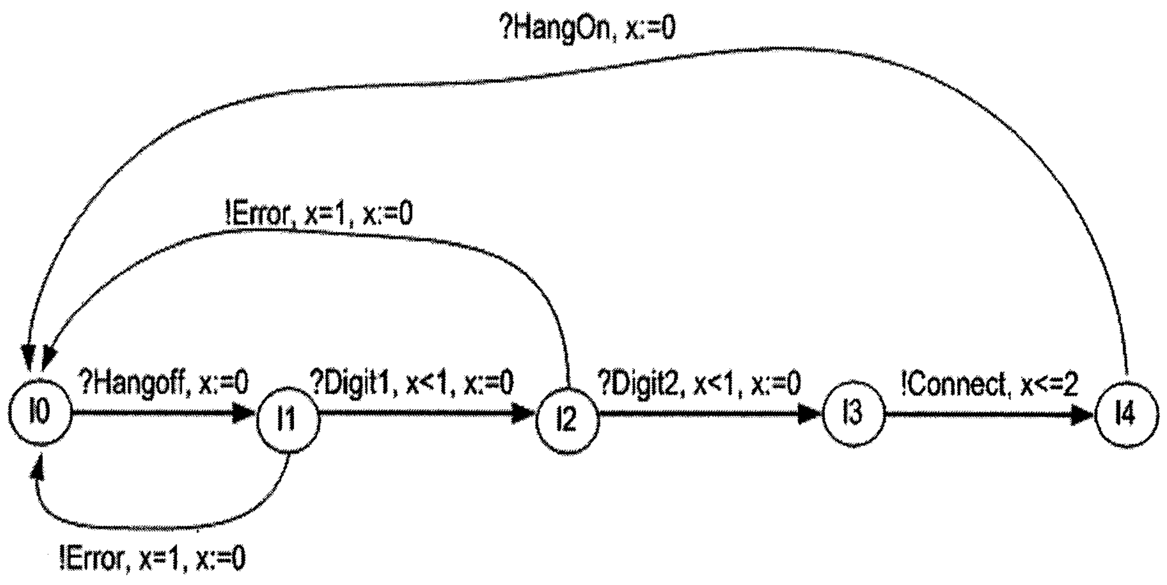

Figure 4-8. The TIOA Specification of Telephone System

The corresponding textual TIOA specification of the telephone system is as follows:

SYSTEM TELE;

CLOCKS

x 2

END CLOCKS;

TRANSITIONS

l0 ?HangOff (x:=0) l1

```
l1      ?Digit1         (x:=0) (x<1)  l2
l2      ?Digit2 (x:=0)(x<1)  l3
l3      !Connect        (x<=2)l4
l4      ?HangOn         (x:=0) l0
l2      !Error (x:=0) (x=1) l0
l1      !Error (x:=0) (x=1) l0
END TRANSITIONS;
END SYSTEM;
```

This textual TIOA representation of the specification is input to the program. The TIOA parser parses and transforms it to the internal *TIOA* class.

4.4.1.2 Test Purpose

Figure 4-9 depicts the graphic test purpose in MSC. The test purpose tries to test the normal actions of the telephone system. It is a subset of functions of the specification. The system clock is reset to zero when the phone is hung off, and the first digit has to be input within one time unit and the second digit has to be accepted at exactly one time unit. Then, the system clock is reset to zero. Within one time unit, a "Connect" must be output, meaning that a connection is setup by the system. In this test purpose, more restricted time constraints is applied to the system under test, and they are compatible with the specification. The differences with the specification are: 1) it is required that the second digit is dialed exactly at one time unit after the first digit is dialed; the specification requires that the second digit must be dialed within 1 time unit; 2) the connection must be issued in one time unit instead of two time units in the specification.

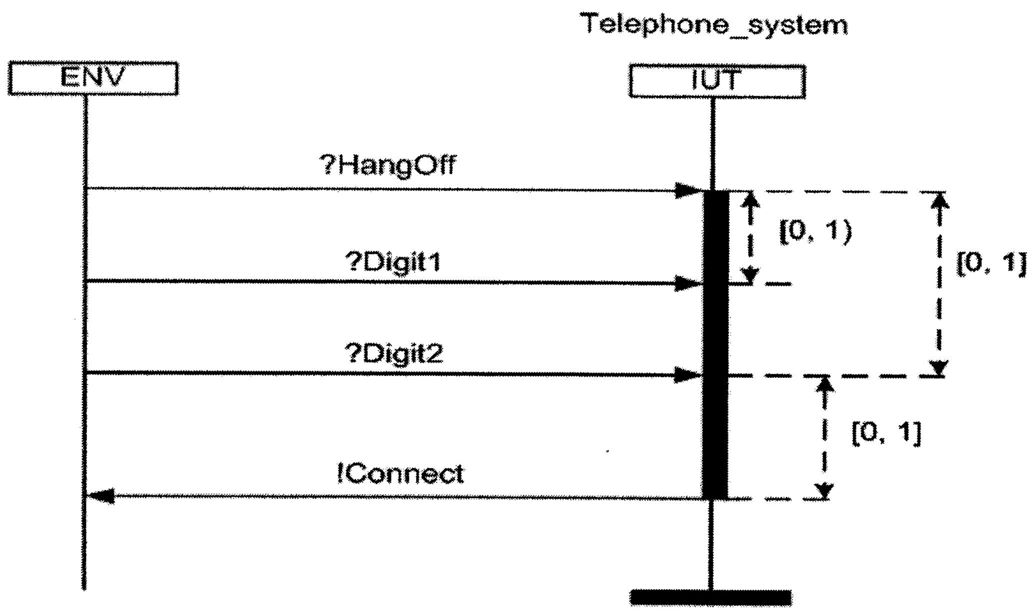

Figure 4-9. A Test Purpose in MSC for Telephone System

The corresponding textual MSC test purpose for the telephone system is as follows:

MSC Telephone_System;

INSTANCE IUT : SYSTEM Telephone ;

E1 IN HangOff FROM ENV time [1, 1] E3 ;

E2 IN Digit1 FROM ENV time [0, 1) E1 ;

E3 IN Digit2 FROM ENV time [0, 1] E4;

E4 OUT Connect TO ENV ;

ENDINSTANCE;

ENDMSC;

This textual MSC representation of test purpose is input to the program. The MSC parser generated by the Flex and Bison parses and transforms it to the internal MSC classes.

4.4.1.3 Results

The example is applied to our program. We will see how every steps works for this example.

First, the test purpose is transformed to a TIOA, which is shown in Figure 4-10. We can see that the transformation creates three clocks ("c0", "c1", "c2"), which are used to describe the three time constraints from the textual MSC representation. It is obvious that clocks "c0" and "c1" are reset at the same time at location *l0*. And after the transition is fired from *l2*, "c0" and "c1" are never active. In addition, "c2" is not active until the location *l3* is reached. Therefore, these three clocks are not all necessary and only one clock is required.

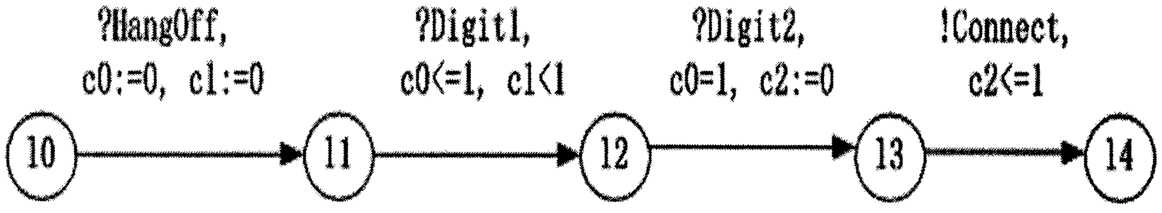

Figure 4-10. Test Purpose as TIOA before Clock Minimizing

Then, the clock minimizing algorithm is applied on the test purpose TIOA. The resulting one has only one clock *y* (Figure 4-11).

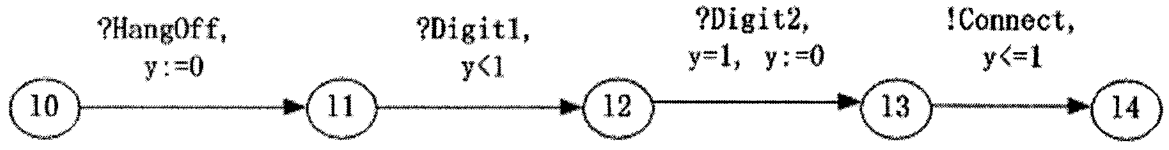

Figure 4-11. Test Purpose as TIOA after Clock Minimizing

The Synchronous Product represents the specification and test purpose requirements simultaneously. It is constructed based on two TIOAs. Figure 4-12 depicts the synchronous product of the telephone system. The clocks *x* and *y* are included in the resulting product, which represents the clock valuations from the specification and the test purpose respectively.

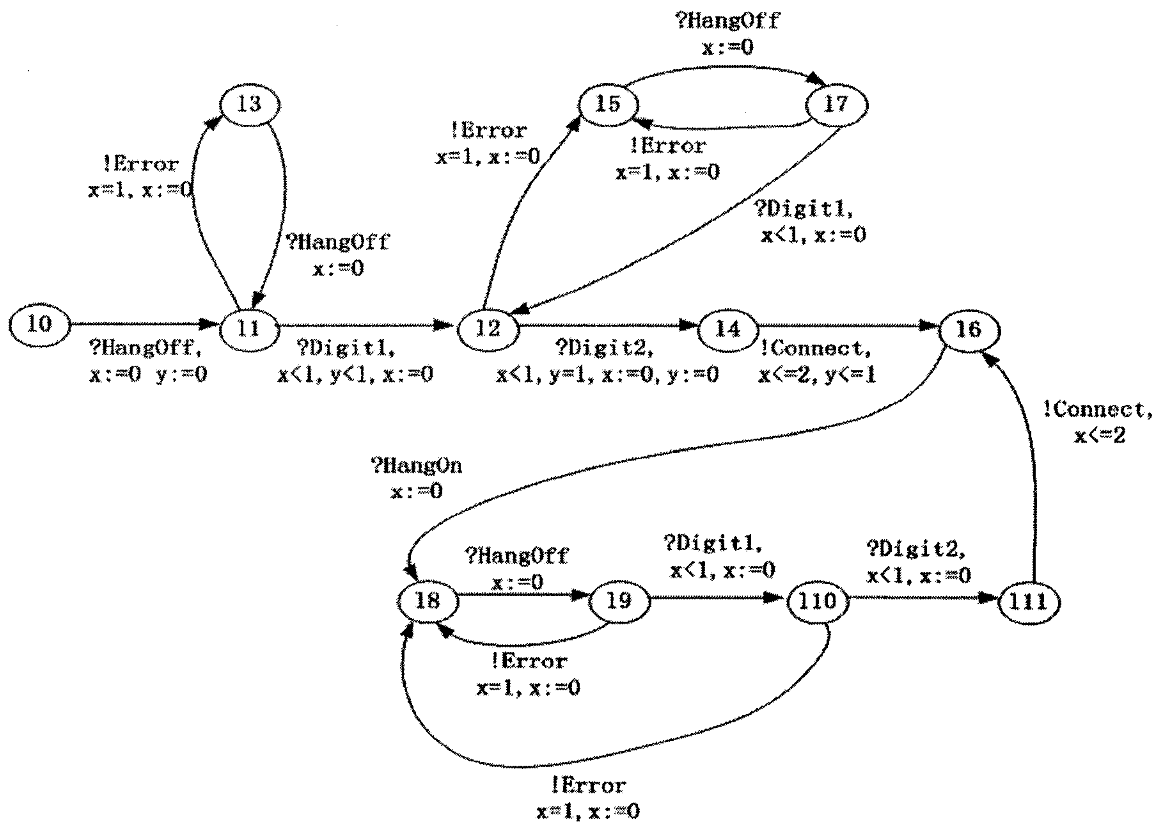

Figure 4-12. Synchronous Product

The Synchronous Product is then sampled by the sampling algorithm. The resulting Grid Automaton has totally 163 states. Part of the GA is depicted in Figure 4-13.

The locations in Synchronous Product are transformed to the states in GA; each state of the GA is described by the valuations of two clocks (where "–1" means the clock value is infinite, when the value is beyond the constraints). States are separated by the transition with the time granularity $\frac{1}{n+1}$ =*1/3*, where *n=2* is the number of clocks. The GA provides a intuitive view of the telephone system's behavior with time progressing.

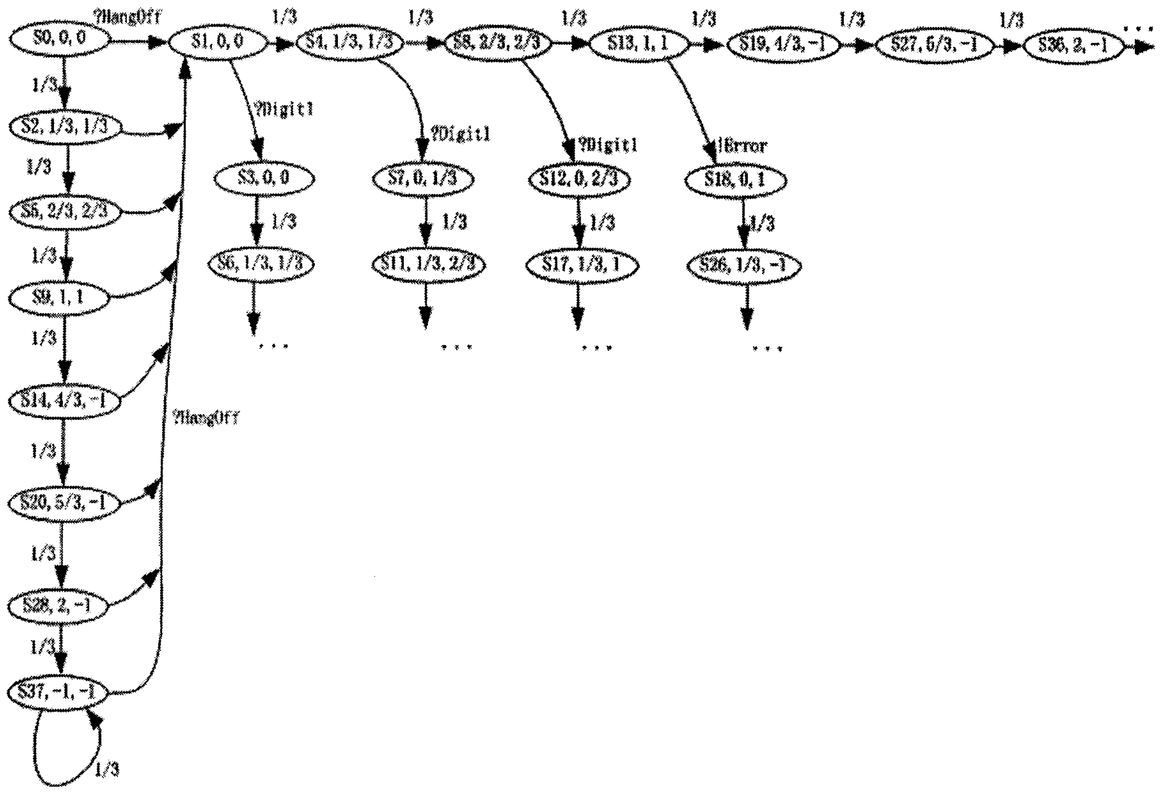

Figure 4-13. Grid Automaton

Finally, the GA is traversed with the traversal algorithm. Test cases are composed of the labels of the transition passed by. The criteria we discussed in section 3.4 of Chapter 3 gives the fault coverage power of different traversal strategies. Figure 4-14 provides partial test cases generated by using all-states coverage algorithm. All test cases for the example are presented in Appendix E.

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3.
?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3.
?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect.
?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3.
?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect.
?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
?Digit1. ?Digit2. !Connect. ?HangOn.
**************
... ...
```

Figure 4-14 Some Resulting Test Cases

The test cases generated by our method are self-evident. For example, the first test case *?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. !Error* in Figure 4-14 means that the phone is hung off and followed by an input *Digit1*; after three *1/3* time units (one time unit), an *Error* should be output by the system. Then, if a *Hangoff* is input at this time, an *Error* should be output by the system again. A verdict "fail" can be concluded if the system has misbehavior. For example, if after three *1/3* time units following input *Digit1*, there is no *Error* being output by the system, the system is considered to be fail. The total generated test cases are provided in Appendix E.

4.4.2 Other Examples

We will provide other examples which are experimented with our method. These examples have different numbers of clocks, clock valuation ranges and numbers of states.

4.4.2.1 10-Digit Telephone System

The 10-digit telephone system([39]) is basically the same as the previous 2-digit telephone system. The time from hanging off the phone to the dial tone output must be within 1 time unit. The time between each digit input is within 2 time units, and the total time to complete the 10-digit input is 12 time units. After the 10 digits are input, the connect should be setup within 3 time units. Any action missing the time constraints can cause an error output and the system is reset to initial state. Figure 4-15 illustrates the specification as a TIOA.

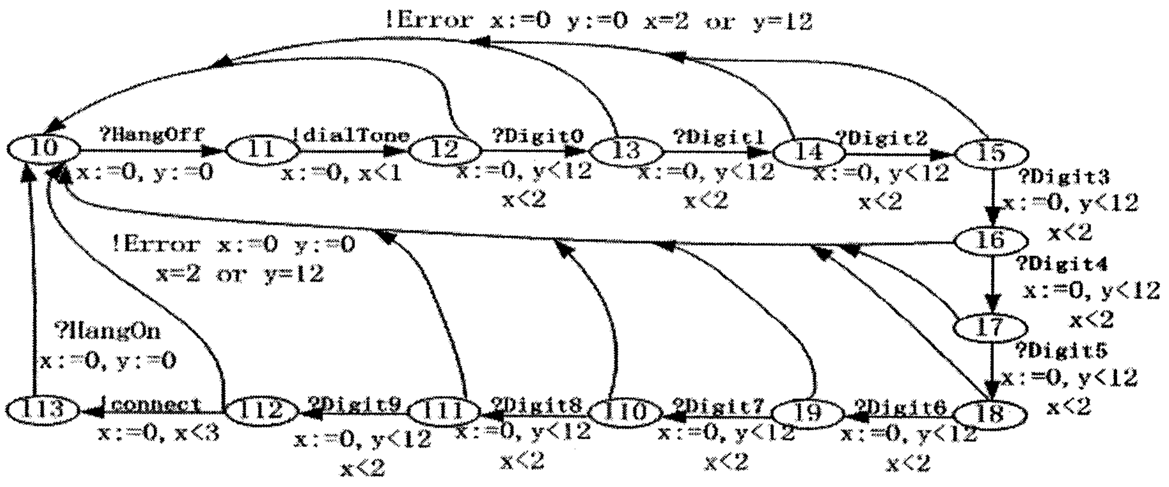

Figure 4-15 Specification of 10-digit Telephone System

The test purpose is depicted in Figure 4-16. The test purpose requires that the time between each digit dialed should be within 1 time unit, instead of 2 time units in the specification. This also implies that the total time of 10-digit dialing will be within 12 time units. The results by applying our method on this system are summarized in Table 4-1.

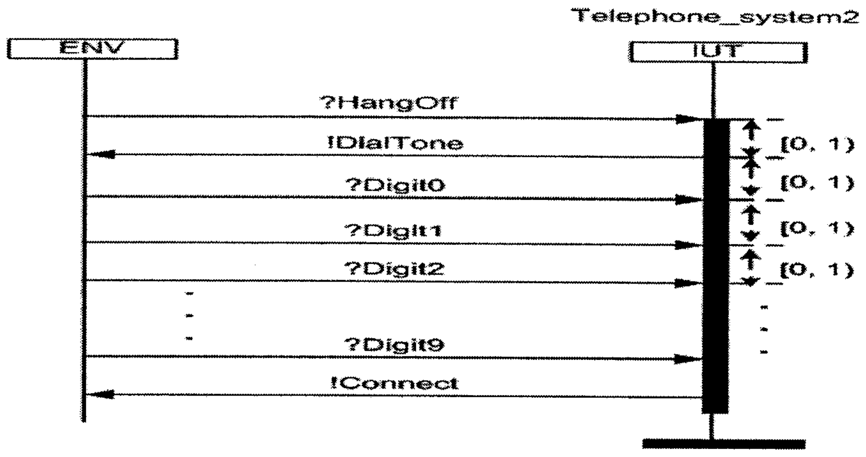

Figure 4-16 Test Purpose of 10-digit Telephone System

4.4.2.2 Multimedia System

The multimedia system([41]) is responsible to receive multimedia data and send back the acknowledge. Once the system obtains the image data, it resets all clocks and requires that the sound data is received within 2 time units and an acknowledge is sent out within 5 time units. Then, the system is again reset to initial state. Any misbehavior resets the system to initial state with an error output. Figure 4-17 illustrates the system behavior.

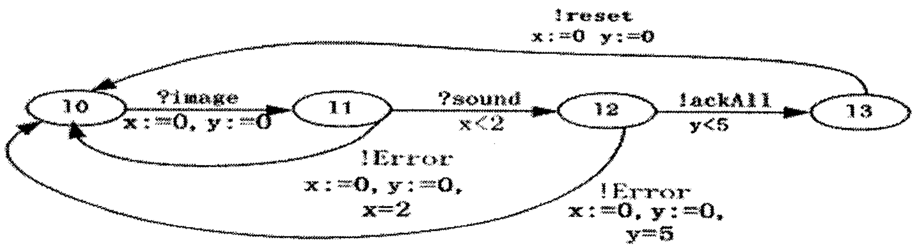

Figure 1-17 Specification of Multimedia System

The test purpose is to test if the system can complete the task within a more limited time constraints. Instead of 5 time units in the specification, the system is required to send out acknowledge within 2 time units in this test purpose, as it is depicted in Figure 4-18. The results are showed in Table 4-1.

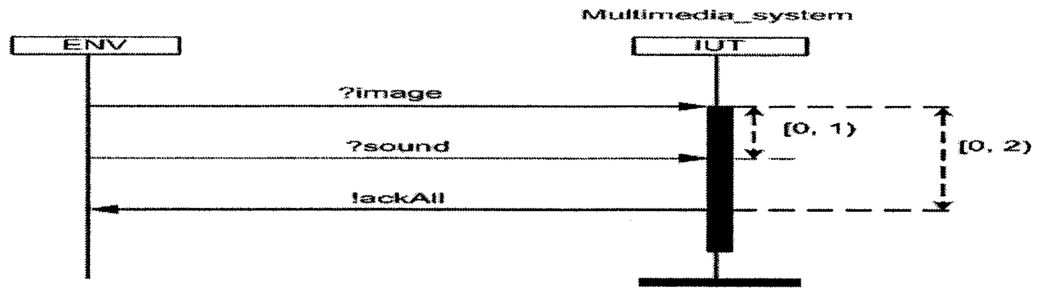

Figure 4-18 Test Purpose of Multimedia System

4.4.2.3 Synchronization Protocol

A simplified synchronization protocol([40]) is depicted in Figure 4-19 as TIOA. The protocol controls how the media data is received and displayed. Five clocks are used to express the time constraints. Since the data starts to be sent, the data should be ended within the duration [7, 11]; the first display should be started within the duration [8,10], and ended within [9, 12] . After the first display, the same cycle runs again, and once display is ended, it goes back to a state for waiting for the next media data.

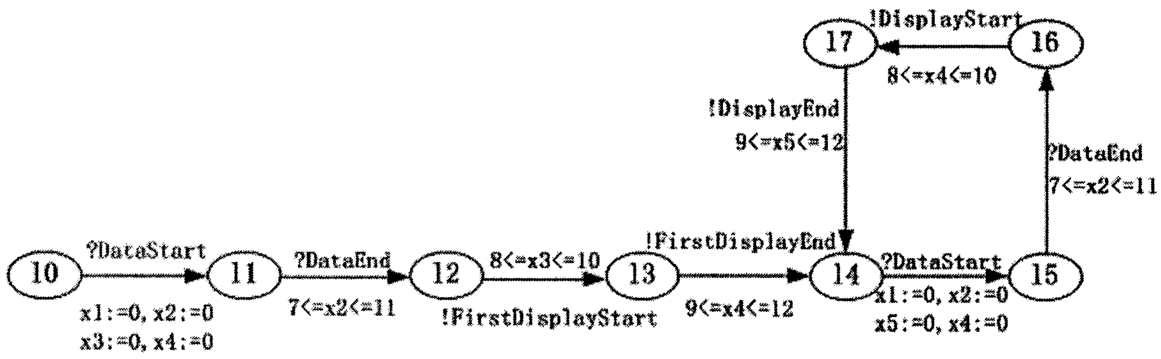

Figure 4-19 Specification of Media Synchronization Protocol

The test purpose is just to test the first synchronization cycle with a more limited time constraint. As it is depicted in Figure 4-20, the first data display should starts within the duration (8, 9), instead of [8,10] in the specification. The results are summarized in Table 4-1.

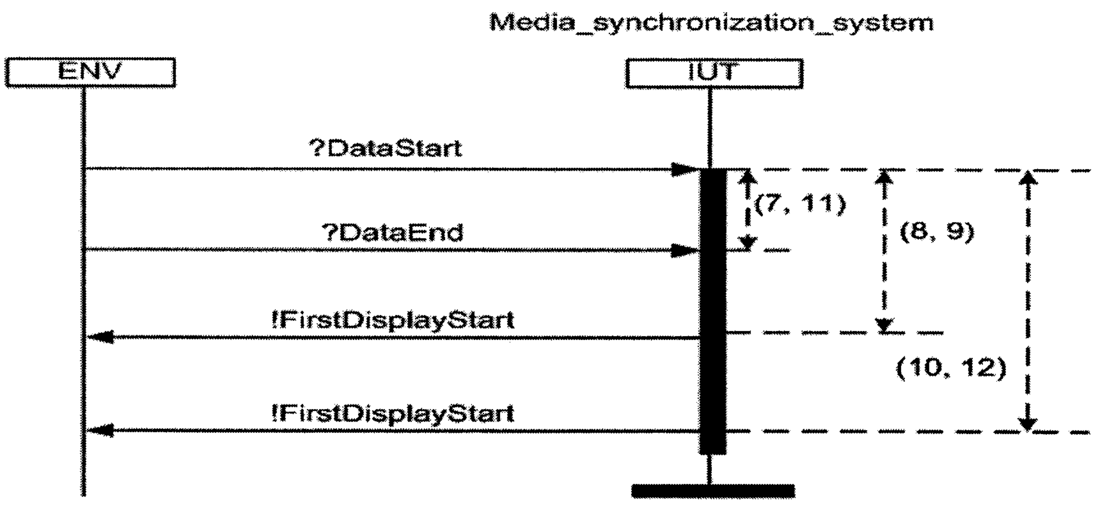

Figure 4-20 Test Purpose of Media Synchronization System

Table 4-1 depicts the results by applying our method on all these examples

System Name	Number of GA States	Number of Test Cases
2-digit Telephone System	163	138
10-digit Telephone System	3707	1563
Multimedia System	297	259
Synchronization System	208	89

Table 4-1: Summary of the Results

From the results table, we can see that these examples with different clock number, state space and clock valuation range can result in different number of GA states and the final test cases.

4.4.3 Related Works

Various research groups have proposed different approaches for testing timed systems based on finite state machine models. In this section, we will summarize some of their work and compare them to our approach.

The authors of [32] provide a theoretical framework for testing timed automaton. They use methods of untimed models to generate test suite from timed automaton. The region graph and GA are also used in their framework. But the clock values are restricted as integer values in timed automaton. Meanwhile, due to using granularity of 2^{-n} to derive GA, where n is the number of clock region composed of specification and implementation TIOA, the resulting test cases is too large to be executable.

The authors of [37][38] propose an architecture to test real time protocols implementation by TIOA. The approach firstly transforms the specification in TIOA to an equivalent untimed one by using timer operations. Then, the W_P–method is applied to the resulting TIOA to generate test suite. The region graph is also used to analyst the time domain constraints. The advantage of this approach is that it solves the state explosion problem. The drawback is that this approach could produce non-executable test suite, and could introduce undesired non-determinism.

The authors of [9] present a test suite generation method based on a restricted class of timed automaton. In this restricted timed automaton, an action is associated with a correspondent clock. Whenever a transition is executed on the action, the correspondent clock is reset to zero. The test suite generation of this approach is based on a equivalent class graph. This graph is a partition of the system time space. There are two steps in the test suite generation method. First, the equivalent class graph is constructed from the timed automaton. Then, the test cases is selected based on a symbolic reachability analyst. The advantage is that this method generates a small number of test cases. The drawback is that it doesn't ensure a full fault coverage.

The authors of [21] introduce a framework for testing timed system. The approach is based on a constraint graph(CG) as the description of specification. The test cases are generated by considering the satisfaction of some test criteria for real time system. However, this approach has the drawback that the constraint graph is not general since that it has the restrict on the duration of time delay. The time delay between input and out events is limited by a minimum and maximum value, following some classification of timing requirements. Moreover, the generated test suite do not cover all the potential faults in an implementation of CG.

Compared to those approaches, our method provides a more integrated framework, and fewer test cases are generated. Moreover, the test suite generation strategy are flexible. The tester could use different traversal algorithms to achieve various coverage power.

4.5 Conclusion

We provided the implementation for the processing of timed test suite generation. We used activities diagram and class and package diagrams to describe the design. Several examples were given to illustrate how our implementation works. We also looked at other researchers' works, and compared them to our approach. In next chapter, we will discuss the unsolved problems and possible improvements regarding our method.

CHAPTER 5

Summary and Future Work

5.1 Summary

The testing methodology described in ISO IS-9646 provides a unified framework to conduct the conformance testing process. In this thesis, we mainly focused on the scope of test case generation. While the exhaustive test is preferred for detecting all possible faults, it is impossible because the tests can be infinite and the test cost is too high.

One of the techniques to reduce the number of tests is Test Hypothesis. A test hypothesis indicates that we do know some facts about the implementation under test, and these facts imply that something about the implementation are true. The test purpose is based on the test hypothesis that the implementation does implement some behaviors that are required. Conformance testing is the activity of verifying if the implementation conforms to the specification. The test purposes in conformance testing describe partial conformance requirements in the specification.

To automatically generate test suite, it is necessary to use formal methods to describe the specification and test purpose in precise and unambiguous manner, especially for real time distributed systems. We provided a method to generate timed test suite based on the test purpose expressed in Message Sequence Chart and the specification expressed in Time Input Output Automaton. The reason why we adopt these two formal models is that MSC is a very intuitive language; it provides the natural means to describe the message exchanging behaviors of distributed systems. Also, MSC is widely used as a tool in the networking software engineering. Nevertheless, MSC lacks the power to describe the internal complex behaviors. The traditional well-known mathematic automaton model provides precise description of the complex behaviors. TIOA i s b ased o n a utomaton m odel w ith t he t ime e xpression, i t h as e nough p ower t o describe complex timed behaviors.

Several methods have been developed to derive test sequence by traversing the non-time automaton (we have described them in section 2.3). For timed automaton, regions graph and grid automaton are introduced in [14] and [16]. They provide a way to analyze the time-related fault models of the automaton. Moreover, the system's state is identified by the clock value, and the time progressing is expressed as transitions between the states. Therefore, it unifies the expression of the function and the time behaviors in an intuitive way.

In our method, the test purpose in MSC is converted to TIOA. Then, a synchronous product is constructed from test purpose TIOA and the specification TIOA. It represents the conformance requirements of both. The Synchronous Product is transformed to a Grid Automaton. Test cases are generated by traversing the Grid Automaton. Two types of fault models are identified: non-time-related faults describe the faults caused by states or transition errors of the implementation automaton, and time-related faults describe the faults that caused by time constraints errors of the implementation. The fault coverage power is decided by applied coverage traversal strategy. The criteria of our method's fault coverage power was given. We also provided the implementation of our method. Several examples are given to illustrate the whole process.

5.2 Future Work

Several problems still remain open in our method. Our method provides a way for partial procedure in the scope of test suite generation of the ISO Conformance Testing Framework. To achieve complete test processing in a more automatic level, more works should be done:

SIMPLE MSC Improvement. The MSC grammar we used in the method is only a small subset of the standard one. It lacks enough power to describe more complicated internal and external system behavior. For example, SIMPLE MSC only has the capability of expressing a deterministic actions sequence. The actions taken by the system is determined in SIMPLE MSC. They are in a fixed time sequence ordering. While in the most real-time distributed systems, the ordering of actions is not determined, and the action taken is decided by run-time condition. The general ordering scenarios are considered in MSC-2000, in which actions can be taken in various ordering; and the run-time conditions can be used to make the decision of which action is taken.

Moreover, to simplify the situation, only one process is considered in our solution (or the whole system under test is regarded as one process). We only consider the exchanging messages between the system under test and the outer environment. In the real world, multi-processes systems are very common. To test multi-processes system, one solution is to separate the different processes, and test each of them as a standalone one. But this solution cannot solve the problems of the coordinated processes, in which some actions are taken to collaborate with each other. An improved solution is to develop a tester that can test internal processes behaviors and the whole system behaviors simultaneously. Especially for the coordination scenarios, as it depicts in Figure 5-1, tester could stimuli the system by sending a message from environment to IUT, and observe the exchanged messages between the internal processes.

In the scope of our method, this requires the MSC describes the multi-processes messages exchanges and their coordination. Ideally, the entire MSC standard grammar should be considered.

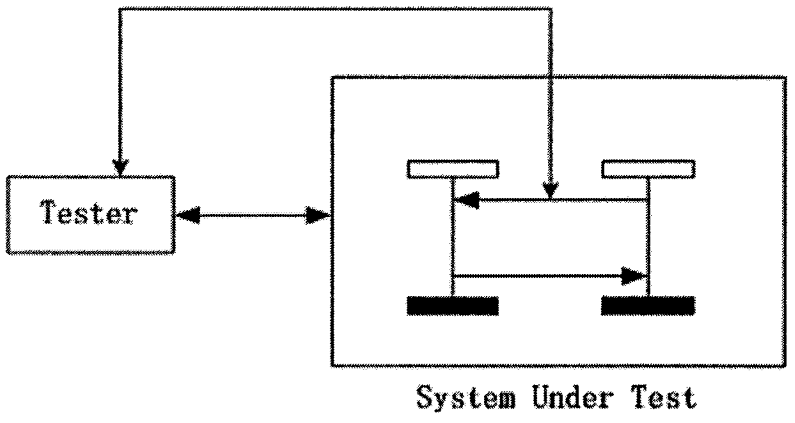

Figure 5-1. Multi-processes Testing

Test suite expression. Until now, the test suite generated by our implementation are expressed in a free format. As it is suggested in [18], Tree and Tabular Combined Notation (TTCN) is expected to be the representation of test suite. In TTCN, a test case describes a test sequence, which can be sent or received by the tester. A test case is expressed as a tree. Each path from the root to a leaf in a test cases tree is an observation. The observation leads to a testing verdict of either a PASS, or a FAIL, or an INCONCLUSIVE. Furthermore, TTCN provides the constructs to group test cases, and parameters can be declared to express the input and output values. Test cases can also be translated to abstract data structure in ASN.1, so they can be easily transformed to executable format.

Fault coverage and traversal algorithms. In the implementation of our method, two fault c overage strategies are realized in traversal a lgorithm: all path coverage and states coverage. The number of test cases generated by all path coverage algorithm are too large in our example; it is almost impossible to execute every test case to cover all the faults. While all states coverage algorithm can only cover a small part of fault model.

One of possible improvements of our work is to create another fault coverage purpose that describes how powerful the fault coverage is needed exactly. By considering the fault coverage purpose, the improved algorithm can adapt the correct traversal strategies. Therefore, a reasonable length of test case can be achieved.

Bibliography

[1] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646. ISO, 1991.

[2] ITU-T. Message Sequence Chart (MSC). International Telecommunications Union, Telecommunications Standards Sector (ITU-T). Recommendation Z.120, 2001.

[3] Abdeslam En-Nouaary and Rachida Dssouli. A Guided Method for Testing Timed Input Output Automata. TestCom'2003, France, 2003

[4] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In 1996 IEEE RTSS'96, Dec. 4-6, 1996, Washington, DC, USA.

[5] Paul Baker, Paul Bristow, et al. Automatic Generation of Conformance Tests From Message Sequence Charts. 3rd SAM Workshop, University of Wales, 2002.

[6] Hanene Ben-Abdallah and Stefan Leue. Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications. *Technical Report 97-04*, Dept. of Electrical and Computer Engineering, University of Waterloo, April 1997.

[7] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary and et al. Test Development for Communication Protocols: Towards Automation. Computer Networks 31 (1999) 1835-1827.

[8] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Wp-Method: Testing Real-time Systems. IEEE Transactions on Software Engineering, November 2002.

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

[10] Philipp Lucas. Timed Semantics of Message Sequence Charts Based on Timed Automata. Workshop on Theory and Practice of Timed Systems (TPTS'02), Grenoble, April 2002.

[11] Gang Luo, Gregor v. Bochmann and Alexandre Petrenko. Test Selection Based on Communicating Nodeterministic Finite-State Machines Using a Generalized Wp-Method. IEEE Transactions on Software Engineering, VOL.20, NO.2, Feb. 1994.

[12] Robert Nahm. Conformance Testing Based on Formal Description Techniques and Message Sequence Charts. March, 1995.

[13] A. En-Nouaary, F. Khendek, and R. Dssouli. Fault Coverage in Testing Real-time Systems. Proc. Sixth Int'l Conf. Real-time Systems Computing Systems and Applications(RTCSA '99), Dec.1999.

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

[15] K.G. Larsen and W. Yi. Time Abstracted Bisimulation: Implicit Specification and Decidability. Proc. Math. Foundations of Programming Semantics (MFPS 9), April 2001.

[16] A. En-Nouaary, R. Dssouli, and F. Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterisation Technique. In 19th IEEE Real-Time Systems Symposium(RTSS'98), Madrid, Spain, December,2-4 1998.

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

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

[19] Gerard J. Holzman. Design and Validation of Computer Protocols. Prentice Hall Internatinal, Inc. 1991.

[20] Jan Tretmans. A Formal Approach to Conformance Testing. PhD Thesis, University of Twente, August 1992.

[21] D. Clarke and I. Lee. Automatic Generation of Tests for Timing Constraints from Requirements. In Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems, California, February 1997.

[22] Sebastien Salva, Eric Petitjean, and Hacene Fouchal. A Simple Approach to Testing Timed Systems. In Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01), Aalborg, Denmark, August 2001.

[23] James Rumbaugh, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[24] Martin Fowler and Kendall Scott. UML Distilled Second Edition. Addison-Wesley, 2000.

[25] Ekkart Rudolph, Jens Grabowski, and Peter Graubmann. Tutorial on Message Sequence Charts (MSC'96).

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

[27] Z. Kohavi. Switching and Finite Automata Theory. McGraw-Hill Computer Science Series, New York, 1978.

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

[29] K. Sabnani and A. Dahbura. A Protocol Test Generation Procedure. Technical Report 11252-870922-10TM, AT&T Bell Laboratories, 1988.

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

[31] A. Petrenko. Testing Based on Finite State Machines. CSS/CSE'2002. August 2002.

[32] J. Springintveld, F. Vaadranger, and P. Dargenio. Testing Timed Automata. Technical Report CTIT97-17, University of Twente, Amesterdam, 1997.

[33] Jan Kroon and Antony Wiles. A Tutorial on TTCN. 11th International IFIP WG6.1 Symposium on Protocol, Specification, Testing and Verification, Vol. 11, pp40-92, 1991.

[34] R. Castanet, O. Kone, and P. Laurencot. On the Fly Test Case Generation for Real Time Protocols. IC3N'98, Bordeaux University, France, 1998.

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

[36] G. Gonenc. A Method for the design of fault detection experiments, IEEE Transactions on Computers C-19 (1970) 551-558.

[37] A. Khoumsi, M. Kalay, R. Dssouli, A. En-Nouaary, and L. Granger. An Approach for Testing Real-Time Protocols. TESTCOM, Aug./Sept. 2000.

[38] A. Khoumsi, A. En-Nouaary, R. Dssouli, M. Akalay. A New Method for Testing Real-Time Systems. RTCSA, Dec. 2000

[39] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In 3rd Workshop on Object-Oriented Real-Time Dependable Systems – (WORD '97). 1997. Newport Beach, CA.

[40] T. Higashino. Generating test cases for a timed I/O automaton model. In G. Csopaki, S. Dibuz, and K. Tarnay, eds, Proc. IFIP Int'l Work. Test. Communicat. Syst. (IWTCS). 1999. Budapest, Hungary: MA: Kluwer Academic.

[41] H. Fouchal, et al. Timed Automata Generation from Estelle Specifications. In Estelle 98. 1998. Paris.

[42] A. En-Nouaary and Gang Liu. Timed Test Suite Generation Based On Test Purpose. 1st International Conference On Information & Communication Technologies: from Theory to Applications – ICTTA'04. April, 2004. Damascus, Syria.

APPENDIX A. SIMPLE MSC Grammar

```
<message_sequence_char>:=

        <mschead> <mscbody> <endmsc>


mschead :=

        msc <msc name> ';'


<endmsc> :=

        endmsc ';'


<mscbody> :=

        <mscstmt>


<mscstmt> :=

        |

        <mscstmt> <stmts>


<stmts> :=

        <instheadstmt> <eventlist> <instendstmt>


<instheadstmt> :=

        instance <instance name> ';'

        | instance <instance name> ':' <instancekind> ';'
```

```
<instancekind> :=

        <kinddeno> <kind name>


<kinddeno> :=

        system

        | block

        | process

        | service


<instendstmt> :=

        endinstance ';'


<eventlist> :=

        |

        <eventlist> <instevent>


<instevent> :=

        <event name> <event> time <duration> <event name> ';'


        | <event> time duration <event name> ';'

        | <event name> <event> ';'

        | <event>
```

```
<event> :=

          <msgevent>

          | <timerevent>


<msgevent> :=

          out <msgid> to <address>

          | out msgid from address


<msgid> :=

          <message name>

          | <message name> '(' <paralist> ')'


<paralist> := /* empty */

          | <para>

          | <paralist> ',' <para>


<para> :=

          <parameter name>

          | <number>

          | <string>


<address> :=
            <address string>

            | env


<timerevent> :=

            <starttimer>

            | <stoptimer>

            | <timeout>


starttimer :=

            starttimer <timer name>

            | starttimer <timer name> <duration>

            | set <timer name>

            | set <timer name> <duration>


<duration> :=

            '[' <number> ',' <number> ']'

            | '(' <number> ',' <number> ')'

            | '[' <number> ',' <number> ']'

            | '(' <number> ',' <number> ')'


<stoptimer> :=

            stoptimer <timer name>
```

<timeout> :=

timeout <timer name>

APPENDIX B. TIOA Grammar

```
<tioafile> :=

        <systemhead> <systembody> <systemend>


<systemhead> :=

        system <system name> ';'


<systemend>:=

        end system ';'


<systembody>:=

        <clockstmt ><transitionstmt>


<clockstmt>:=

        <clockhead ><clockbody ><clockend>


<clockhead>:=

        clocks


<clockend>:=

        end clocks ';'
```

```
<clockbody>:=

          <clockitems>


<clockitems>:=

          |

          <clockitems ><clkitem>


<clkitem>:=

          <clock name> <number>


<transitionstmt>:=

          <tranhead ><tranbody ><tranend>


<tranhead>:=

          transitions


<tranend>:=

          end transitions ';'


<tranbody>:=

          <tranitems>


<tranitems>:=
          |
          <tranitems ><titem>

<titem>:=
          <location name> <label ><guard > <location name>

<label>:=
          <transition lable>

<guard>:=
          |
          <guard ><guarditem>

<guarditem>:=
          '(' <resetitem >')'
          | '(' <constritem >')'

<resetitem>:=
          <clock name> ':' '=' <number>

<constritem>:=
          <clock name> <op > <number>
```

```
<op>:=

        '<' '='

        | '>' '='

        | '<'

        | '>'

        | '='
```

APPENDIX C. Lex and Yacc Files for SIMPLE MSC

Simple MSC lex file:

```
%{
#include <stdlib.h>
#include "mscparser.cpp.h"
#define YYSTYPE char *
%}

%%
[Mm][Ss][Cc]    {return MSC;}
[Ee][Nn][Dd][Mm][Ss][Cc]  {return ENDMSC;}
[Ii][Nn][Ss][Tt][Aa][Nn][Cc][Ee]  {return INSTANCE;}
[Ee][Nn][Dd][Ii][Nn][Ss][Tt][Aa][Nn][Cc][Ee]          {return
ENDINSTANCE;}
[Oo][Uu][Tt]  {return OUT;}
[Tt][Oo]  {return TO;}
[Ii][Nn]  {return IN;}
[Ff][Rr][Oo][Mm]  {return FROM;}
```

```
[Ss][Tt][Aa][Rr][Tt][Tt][Ii][Mm][Ee][Rr]|[Ss][Ee][Tt]
{return STARTTIMER;}
[Ss][Tt][Oo][Pp][Tt][Ii][Mm][Ee][Rr] {return STOPTIMER;}
[Tt][Ii][Mm][Ee][Oo][Uu][Tt] {return TIMEOUT;}
[Tt][Ii][Mm][Ee]      {return TIME;}
[Ss][Yy][Ss][Tt][Ee][Mm] {return SYSTEM;}
[Bb][Ll][Oo][Cc][Kk] {return BLOCK;}
[Pp][Rr][Oo][Cc][Ee][Ss][Ss] {return PROCESS;}
[Ss][Ee][Rr][Vv][Ii][Cc][Ee] {return SERVICE;}
[Ee][Nn][Vv] {return ENV;}
[a-zA-Z`][a-zA-Z0-9_`]* {msclval=strdup(msctext);    return
WORD;}
[0-9]+    {msclval=strdup(msctext); return NUMBER;}
[ \t\n\r]+         ; /* ignore white space */
"/*".*"*/"  ; /* ignore comments */
"\'".*"\'"     {msclval=strdup(msctext); return STRING;}
. {return *msctext;}


%%
```

SIMPLE MSC Yacc file:

```
%{
#include <stdio.h>
```

```
#include <string.h>

#include "ptypes.h"

#include "parsemsc.h"


char msg[100];

char temp[100];

extern FILE *mscin;

Duration dura;


#define YYSTYPE char *


extern "C" {


int msclex(void);


int mscparse(void);


void mscerror(const char *str)

{

        fprintf(stderr,"error: %s\n",str);

}


int mscwrap()

{
```

```
          return 1;
    }

    }

%}

%token WORD NUMBER MSC ENDMSC INSTANCE ENDINSTANCE OUT TO
IN FROM STARTTIMER STOPTIMER TIMEOUT

%token SYSTEM BLOCK PROCESS SERVICE ENV STRING TIME

%%

message_sequence_char:
      mschead mscbody endmsc
      ;

mschead:
      MSC WORD ';'    {}
      ;

endmsc:
      ENDMSC ';' {}
    ;


mscbody:
    mscstmt
    ;


mscstmt:
    |
    mscstmt stmts
    ;


stmts:
    instheadstmt eventlist instendstmt
    ;


instheadstmt:
    INSTANCE WORD ';'
    {
        instancename($2);
    }
    | INSTANCE WORD ':' instancekind ';'
    {
        instancename($2);
    }
    ;


instancekind:

    kinddeno WORD

    ;


kinddeno:

    SYSTEM

    | BLOCK

    | PROCESS

    | SERVICE

    ;


instendstmt:

    ENDINSTANCE ';'      {}

    ;


eventlist:

    |

    eventlist instevent

    ;


instevent:

    WORD event TIME duration WORD ';'
```

```
        {
            eventlabel($1, dura, $5);
        }
        | event TIME duration WORD ';'
        {
            eventlabel(NULL, dura, $4);
        }
        | WORD event ';'
        {
            dura.min = 0;
            dura.max = 0;
            eventlabel($1, dura, NULL);
        }
        | event ';'
        ;

event:
        msgevent
        | timerevent
        ;

msgevent:
        OUT msgid TO address
        {
            msgevent($2, false, $4);

            msg[0] = '\0';

        }

        | IN msgid FROM address

        {

            msgevent($2, true, $4);

            msg[0] = '\0';

        }

        ;


msgid:

        WORD

        {

            $$ = $1;

        }

        | WORD '(' paralist ')'

        {

            sprintf(temp, "%s(%s)", $1, $3);

            strcpy(msg,temp);

            $$ = msg;

        }

        ;


paralist: /* empty */
```

```
        | para {sprintf(msg, "%s", $1); $$=msg;}
        | paralist ',' para
        {
                sprintf(temp, "%s,%s",$1, $3);
                strcpy(msg,temp);
                $$=msg;
        }
        ;


para:
        WORD {$$=$1;}
        | NUMBER  {$$=$1;}
        | STRING  {$$=$1;}
        ;


address:
        WORD { $$ = $1;}
        | ENV      { $$ = strdup("env");}
        ;


timerevent:
        starttimer      { }
        | stoptimer     { }
        | timeout { }
        ;


starttimer:
        STARTTIMER WORD
        {
                dura.min = 0;
                dura.max = 0;
                tsevent($2, dura);
        }
        | STARTTIMER WORD duration
        {
                tsevent($2, dura);
        }
        ;


duration:
        '[' NUMBER ',' NUMBER ']'
        {
                sprintf(msg, "[%s,%s]", $2, $4);
                $$=msg;
                dura.min = atoi($2);
                dura.max = atoi($4);
                dura.closemin = true;
                dura.closemax = true;
```

```
}
| '(' NUMBER ',' NUMBER ']'
{
     sprintf(msg, "(%s,%s]", $2, $4);
     $$=msg;
     dura.min = atoi($2);
     dura.max = atoi($4);
     dura.closemin = false;
     dura.closemax = true;
}
| '[' NUMBER ',' NUMBER ')'
{
     sprintf(msg, "[%s,%s)", $2, $4);
     $$=msg;
     dura.min = atoi($2);
     dura.max = atoi($4);
     dura.closemin = true;
     dura.closemax = false;
}
| '(' NUMBER ',' NUMBER ')'
{
     sprintf(msg, "(%s,%s)", $2, $4);
     $$=msg;
     dura.min = atoi($2);
```

```
                dura.max = atoi($4);
                dura.closemin = false;
                dura.closemax = false;
        }
;

stoptimer:
        STOPTIMER WORD
        ;

timeout:
        TIMEOUT WORD { toevent($2); }
        ;
%%

void parseMSCFile(char *filename)
{
        msg[0] = '\0';
        mscin = fopen(filename, "r");
        mscparse();
        fclose(mscin);
}
```

APPENDIX D. Lex and Yacc Files For TIOA

TIOA lex file:

```
%{

#include <stdlib.h>

#include "tioaparser.cpp.h"

#define YYSTYPE char *

%}


%%

[Ss][Yy][Ss][Tt][Ee][Mm]       {return SYSTEM;}

[Cs][Ll][Oo][Cc][Kk][Ss]  {return CLOCKS;}

[Tt][Rr][Aa][Nn][Ss][Ii][Tt][Ii][Oo][Nn][Ss]  {return

TRANSITIONS;}

[Ee][Nn][Dd]    {return END;}

[a-zA-Z`][a-zA-Z0-9_`]*   {tioalval=strdup(tioatext);  return

WORD;}

[0-9]+    {tioalval=strdup(tioatext); return NUMBER;}

[ \t\n\r]+          ; /* ignore white space */

"/*".*"*/"   ; /* ignore comments */

[?!][a-zA-Z`][a-zA-Z0-9_`]*    {tioalval=strdup(tioatext);

return LABEL;}
```

```
. {return *tioatext;}
%%
```

TIOA yacc file:

```
%{
#include <stdio.h>
#include <string.h>
#include "parsetioa.h"
char opsymbol[5];
extern FILE *tioain;
#define YYSTYPE char *

extern "C" {
int tioalex(void);
int tioaparse(void);

void tioaerror(const char *str)
{
        fprintf(stderr,"error: %s\n",str);
}

int tioawrap()
{
        return 1;
}


}
%}


%token WORD NUMBER END SYSTEM TRANSITIONS CLOCKS LABEL


%%


tioafile:
     systemhead systembody systemend
     ;


systemhead:
     SYSTEM WORD ';'      { setSysName($2); }
     ;


systemend:
     END SYSTEM ';'
     ;


systembody:
     clockstmt transitionstmt
     ;
```

```
clockstmt:

      clockhead clockbody clockend

      ;


clockhead:

      CLOCKS

      ;


clockend:

      END CLOCKS ';'

      ;


clockbody:

      clockitems

      ;


clockitems:

      |

      clockitems clkitem

      ;


clkitem:

      WORD NUMBER          { setClock($1, atoi($2));}
```

```
        ;


transitionstmt:

        tranhead tranbody tranend

        ;


tranhead:

        TRANSITIONS

        ;


tranend:

        END TRANSITIONS ';'


tranbody:

        tranitems

        ;


tranitems:

        |

        tranitems titem

        ;


titem:

        WORD label guard WORD    { setTran($1, $2, $4); }
```

```
    ;

label:
    LABEL     {$$=$1;}
    ;

guard:
    |
    guard guarditem
    ;

guarditem:
    '(' resetitem ')'
    | '(' constritem ')'
    ;

resetitem:
    WORD ':' '=' NUMBER { setClkReset($1);}
    ;

constritem:
    WORD op NUMBER      { setConstr($1, $2, atoi($3)); }
    ;
```

```
op:
        '<' '='
        {
                sprintf(opsymbol, "%s%s", "<", "=");
                $$ = opsymbol;
        }
        | '>' '='
        {
                sprintf(opsymbol, "%s%s", ">", "=");
                $$ = opsymbol;
        }
        | '<'       { $$ = "<"; }
        | '>'       { $$ = ">"; }
        | '='       { $$ = "="; }
        ;

%%
void parseTIOAFile(char *filename)
{
        tioain = fopen(filename, "r");
        opsymbol[0] = '\0';
        tioaparse();
        fclose(tioain);
}
```

APPENDIX E. Output Test Cases for Telephone System

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. ?HangOff. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
```

```
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
```

```
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
```

```
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. !Connect.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
```

```
1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff.
1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3.
1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
```

```
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. 1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. ?HangOff. 1/3. 1/3.
1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
```

```
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. 1/3. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. 1/3. !Error.
```

```
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect.
1/3. ?HangOn. ?HangOff. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn.
1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. ?HangOn. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. !Connect. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3. !Connect.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. 1/3. 1/3. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2.
1/3. !Connect. ?HangOn. ?HangOff. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect. ?HangOn.
1/3. ?HangOff.
```

```
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect. ?HangOn.
1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect. ?HangOn.
1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect.
1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. !Connect. 1/3.
1/3. ?HangOn.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. ! Connect. ? HangOn. ?HangOff. ? Digit1. ? Digit2. 1 /3. ! Connect. 1 /3. 1 /3.
1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. ?Digit2. 1/3. 1/3. 1/3. 1/3.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. ?Digit1. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. 1/3. 1/3. 1/3. !Error.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. ?HangOff. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. 1/3. ?HangOff.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. ?HangOn. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3.
1/3. ?Digit2. !Connect. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. !Connect.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. ?Digit2.
1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
```

```
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3.
1/3. 1/3. ?Digit2.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3. 1/3. 1/3.
1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
1/3. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. ?HangOff. ?Digit1.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. ?HangOff. 1/3.
**************
```

```
**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. ?Digit1. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. 1/3. ?Digit1. 1/3. 1/3. !Error.
**************

**************
?HangOff. 1/3. ?Digit1. 1/3. 1/3. 1/3.
**************

**************
?HangOff. 1/3. 1/3. ?Digit1. 1/3. !Error.
**************

**************
?HangOff. 1/3. 1/3. ?Digit1. 1/3. 1/3.
```

```
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. !Error.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. ?Digit1.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?Digit1.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. ?HangOff. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. ?HangOff. ?Digit1.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. ?HangOff. 1/3.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************
```

```
**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
?HangOff. 1/3. 1/3. 1/3. !Error. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3. 1/3.
**************

**************
?HangOff. 1/3. 1/3. 1/3. 1/3.
**************

**************
1/3. ?HangOff.
**************

**************
1/3. 1/3. ?HangOff.
**************

**************
1/3. 1/3. 1/3. ?HangOff.
**************

**************
1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************

**************
1/3. 1/3. 1/3. 1/3. 1/3. 1/3. ?HangOff.
**************
```