Testing Embedded Real-Time Systems Based On Test Purposes

Zhang Xiang

A Thesis
in
The Department
of
Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
in Electrical & Computer Engineering
at Concordia University
Montreal, Quebec, Canada

December 2004

Your file Votre référence
ISBN: 0-494-04407-1
Our file Notre référence
ISBN: 0-494-04407-1

Abstract

Testing Embedded Real-Time Systems Based on Test Purpose

Embedded real-time systems are those systems which have many components interacting with each other and with the environment. The behavior of these systems is time sensitive and governed by time constraints. One of the greatest difficulties of testing embedded real-time systems is the well-known state explosion problem, which is due to the clock variables used, the number of components that form the system, and the communication between these components.

Over the past decade, some researchers have investigated the issue of testing real-time systems and come up with new methods. Some of these methods successfully generate test cases with acceptable/good fault coverage. However, most of the proposed methods suffer from the well-known state space explosion problem and generate a great number of test cases. Therefore, it is very important to develop new timed testing methods that are practical enough and cover as much as possible the potential faults in the implementation being tested.

In this thesis, we introduce a methodology to generate test cases for embedded real-time systems based on test purposes expressed as Message Sequence Charts (MSCs), and timed input output automata as specification model. The approach consists of six main operations: (1) the parsing of the specification and test purposes; (2) the selection of the transition paths to be considered for test cases generation; (3) the construction of the synchronous product of the test purposes and the TIOA specifications; (4) the construction of a partial product for the system under test; (5) the sampling of TIOAs; and (6) the generation of test cases. Within these six operations, the sampling operation can be executed in different levels. Each level gives rise to a new method for test cases generation for embedded real-time systems. We implemented and studied three of these methods by comparing them in terms of the number of states generated and the number of test cases devised.

Keywords: *Embedded Real-Time Systems, Test Cases Generation, Test Purposes, Message Sequence Charts, Timed Input Output Automata. Grid Automata, Synchronous Product, Partial Product.*

Acknowledgements

I would like to take this chance to express my deep gratitude to my supervisor, Dr. Abdeslam En-Nouaary, for his guidance during each stage of my research. I am very grateful for his patience, support and concern. His valuable advices are extremely useful in extending and deepening my knowledge in the field of software testing.

I also appreciate the efforts of all the members of the Examining Committee for their time and patience to review my thesis.

Table of Contents

List of Abbreviations

RTS	Real-time Systems
MSC	Message Sequence Chart
BMSC	Basic Message Sequence Chart
TIOA	Timed Input Output Automata
CTIOA	Communicating Timed Input Output Automata
IUT	Implementation Under Test
SUT	System Under Test
RCS	Railroad Crossing System
TTCN	Test Tabular Combined Notation
PCO	Points of Control and Observation
OSI	Open Systems Interconnection
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
FDT	Formal Description Technology
TTCN	Test Tabular Combined Notation
OOP	Object Oriented Programming

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Overview

Real-time systems are used in many applications in our society, varying from telephone-switching systems to patient monitoring and plant control systems. The behavior of real-time systems is time sensitive and governed by time constraints. Real-time systems are usually embedded systems, and consist of many components running concurrently and communicating with each other and with the environment. It is well known to the real-time systems research community that the misbehavior of real-time systems is generally due to the non-respect of the timing aspect of their behavior and causes catastrophic consequences on both human lives and the environment. So, it is mandatory to make sure that the implementation of a real-time system is correct before its deployment. Testing is one of the formal techniques that can be used to ensure reliable real-time systems. It consists of generating test cases from the specification of the system, submitting these test cases to the implementation under test (IUT for short) and observing its reactions, and analyzing the results in order to conclude a verdict. If the reactions of the IUT match the expected outputs, the implementation is said to conform to its specification; otherwise the implementation is said to be faulty and the diagnosis process should be started in order to locate and fix the fault.

Testing real-time systems is different from testing classical applications because time is not under the direct control of the user/tester and the timing aspect generates too many states (i.e., the state explosion problem). Therefore, it is not advisable to apply untimed testing techniques to real-time systems. Hence, the development of new techniques for testing real-time systems is an urgent need.

Over the past decade, some researchers have investigated the issue of testing real-time systems with different backgrounds and come up with new methods (see for instance [1], [2], [3], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]). Some of these methods successfully generate test cases with acceptable/good fault coverage. However, most of the proposed methods suffer from the well-known state space explosion problem and generate a great number of test cases. Therefore, it is very important to develop new timed testing methods that are practical enough and cover as much as possible the potential faults in the implementation being tested.

In our thesis, we introduce a methodology to generate test cases for embedded real-time systems based on test purposes expressed as Message Sequence Charts (MSCs) and the specification given in Timed Input Output Automata (TIOA) model. A test purpose is a precise representation of the behaviors to be tested. In the case of embedded real-time systems, a test purpose is a sequence of interactions amongst the components of the systems and the environment, as well as some time constraints on these interactions. Test purposes are useful in testing because they help reduce the number of test cases to be generated and incrementally carry out the testing process by confining the behaviors to be tested. This is one of the reasons why ISO intensively uses them in its IS9646 testing framework [23]. In our work, test purposes are expressed as MSC because MSC is often used in the industry and provides graphical and textual grammar formats to specify the behavior to be tested. As for the specification of the system under test, we adopt the

TIOA model because it is general/rich enough to describe a large number of real-time systems.

1.2 The Objective of the thesis

This research focuses on test suite generation for embedded real-time systems based on test purposes. Both the specification of the system and the test purposes are given in text files and we need to create a parser for them. The test purposes are described in Message Sequence Chart (MSC) and need to be converted into TIOA models for later calculations. For convenience of the generation of test cases, the test purpose models and the specification models should be combined into one global model. The TIOA models have infinite states in the time domain and have to be discretized with enough states coverage. Finally, we need to generate the test cases with certain fault coverage.

In addition, not all components of the systems are involved in testing. We are only interested in the components of the IUT and its contexts. We are concerned with their transition paths that correspond to traces in the test purposes models. Therefore, a selection procedure is needed to construct a sub-specification model for testing. Meanwhile, we need to solve problems on path executability, path synchronization, and path testability during selection procedure.

We need to pay more attention to discretizing in the time domain. It is one of the key parts to incur states explosion. A sampling algorithm is designed carefully to decrease the number of states generated.

1.3 Organization

The remainder of this thesis is structured as follows. Chapter 2 will review the development on conformance testing. We will introduce conformance testing and how it works. Then, we will give an overview on formal conformance testing, how formal methods are used to generate a test suite automatically.

In Chapter 3 we will give a brief overview of real-time systems testing first. Then, we will introduce some basic concepts that are important in our methodology, such as test purpose and timed input output automaton. To explain those concepts clearly, we will use the well-known Railroad Crossing System (RCS) [17] as example.

Chapter 4 will present our approach for timed test suite generation. In this chapter, we will explain the ideas of our methodology and provide algorithms on MSC translation, transition paths selection, synchronous product construction, partial product construction, differential sampling, and test cases generation.

In Chapter 5, we will show the classes that are designed for implementing our methodology.

In Chapter 6, an example of applying our methodology is given and the results are analyzed.

Chapter 7 will draw conclusions based on the results of this research and offer recommendations for further work.

Chapter 2
Conformance Testing in software engineering

In this chapter, we concentrate on conformance testing. The whole chapter is divided into three sections. In the first section, we introduce the definition of conformance testing, its position in software testing family and why, when, where we use conformance testing. In the second section, we introduce the architecture of conformance testing under ISO conformance testing framework. In section three, we review formal conformance testing.

2.1 What is conformance testing?

Testing is critical in software engineering. It exists almost in every phase of software engineering life-cycle, from requirements to integration, to make sure that the quality of software products is good enough. In the requirement phase, testing is used to see whether the requirement document meets the client's real needs. In the specification phase, testing is used to check for contradictions, ambiguities, incompleteness, and whether it conforms to the requirements from clients. In the design phase, testing is mainly used to find logic faults, interface faults, and other faults in design structure and modules. It also checks the conformance to the specification through cross-reference. People usually differentiate testing before the implementation phase from testing applied in the implementation phase and integration phase by calling them V&V, verification and validation. In the implementation phase, test cases are applied to the modules with either testing against specifications or testing against code structures. In the integration phase, the modules which have been tested separately in the implementation phase are integrated as a whole system and varieties of testing methods are applied for conformance, performance, robustness, and reliability testing.

Conformance testing is a kind of testing where an implementation is tested with respect to its specification. "Conformance testing should be used by implementers early-on in the development process, to improve the quality of their implementations and by industry associations wishing to administer a testing and certification program. Conformance testing are meant to provide the users of conforming products some assurance or confidence that the product behaves as expected, performs functions in a known manner, or has an interface or format that is known. Conformance testing is NOT a way to judge if one product is better than another. It is a neutral mechanism to judge a product against the criteria of a standard or specification" [24].

One of the measurements to evaluate the quality of software products is how much it meets the requirements from customers. Every customer comes to a developer with sorts of requirements for functionalities and constraints. In most of the time, these requirements are vague, unreasonable, contradictory, or simply impossible to achieve. Under the cooperative work between the developer and the customer, these requirements are elicited and documented with informal language such as a natural language. The requirements document is the primal document that records all requirements from the customer. In software engineering, this procedure is called requirement phase. These requirements are supposed to be satisfied when the developer hands over the product to the customer. But the document in requirements phase cannot guide the developing phases afterwards because it is too informal to describe precisely the functionalities of the product. In the next phase, the document is completed and detailed by the developer with a semiformal or formal language. This is the specification phase and the document in this phase is called the specification of the product. It is a valid contract between the client and the developer and is also the essential document for later developing procedure. The developer is deemed to realize the functions and satisfy the constraints defined in the specification of the product. Conformance testing is the way to verify this conformance relation between the specification and the product.

In most cases, conformance testing is a kind of black-box testing. It ignores the codes and structures of the product and generates test cases only based on the specification. Exhaustive testing is impractical here because it needs countless test cases to reach high fault coverage. To generate a relatively small set of test cases with enough fault coverage, there are two common used techniques of black-box testing -equivalence testing with boundary values analysis and functional testing. In equivalence testing with boundary values technique, for each range [a,b] in either the input specification or the output specification, we generate five test cases: $x < a; x = a; a < x < b; x = b; x > b$. The key point in this technique is how to identify boundaries. In functional testing technique, we identify functions of the product and generate at least one test case for every function. In practice, these two techniques are often used in one testing job to generate test cases.

In contrast to black-box testing, there is another kind of testing called white-box testing, which generates test cases based on the knowledge of codes and the structure of the product. Usually this kind of testing happens in module testing at the implementation phase. There is also a testing method, called grey-box testing, which is a combination of black-box testing and white-box testing. In theory, all these testing methods are equally effective in finding faults. But usually, only black-box testing is a practical testing method during integration testing.

In addition to conformance testing, there are interoperability testing, performance testing, robustness testing, and reliability testing. Interoperability testing is to determine whether two implementations or more will actually inter-operate and if not, why[4]? Performance testing is to measure the performance characteristics of an implementation. Robustness testing is to examine the implementation's behavior in an erroneously behaving environment, and Reliability testing is to check whether the implementation continues to work correctly during a certain period of time [25] and under some circumstances.

2.2 How does conformance testing work?

Conformance testing plays a major role in software development. To make the conformance testing results repeatable, comparable, and auditable, there is a need to standardize the testing procedure. IS9646, "OSI Conformance Testing Methodology and Framework" [23], is one of the typical conformance testing standards. It is designed for the development of OSI protocols by ISO and IUT (formerly CCITT), but now it is widely used in software engineering field, especially in the development of communication protocols. In the standard, the procedure of conformance testing is divided into three steps: test cases generation, test cases implementation and test execution, as shown in Figure 2-1.

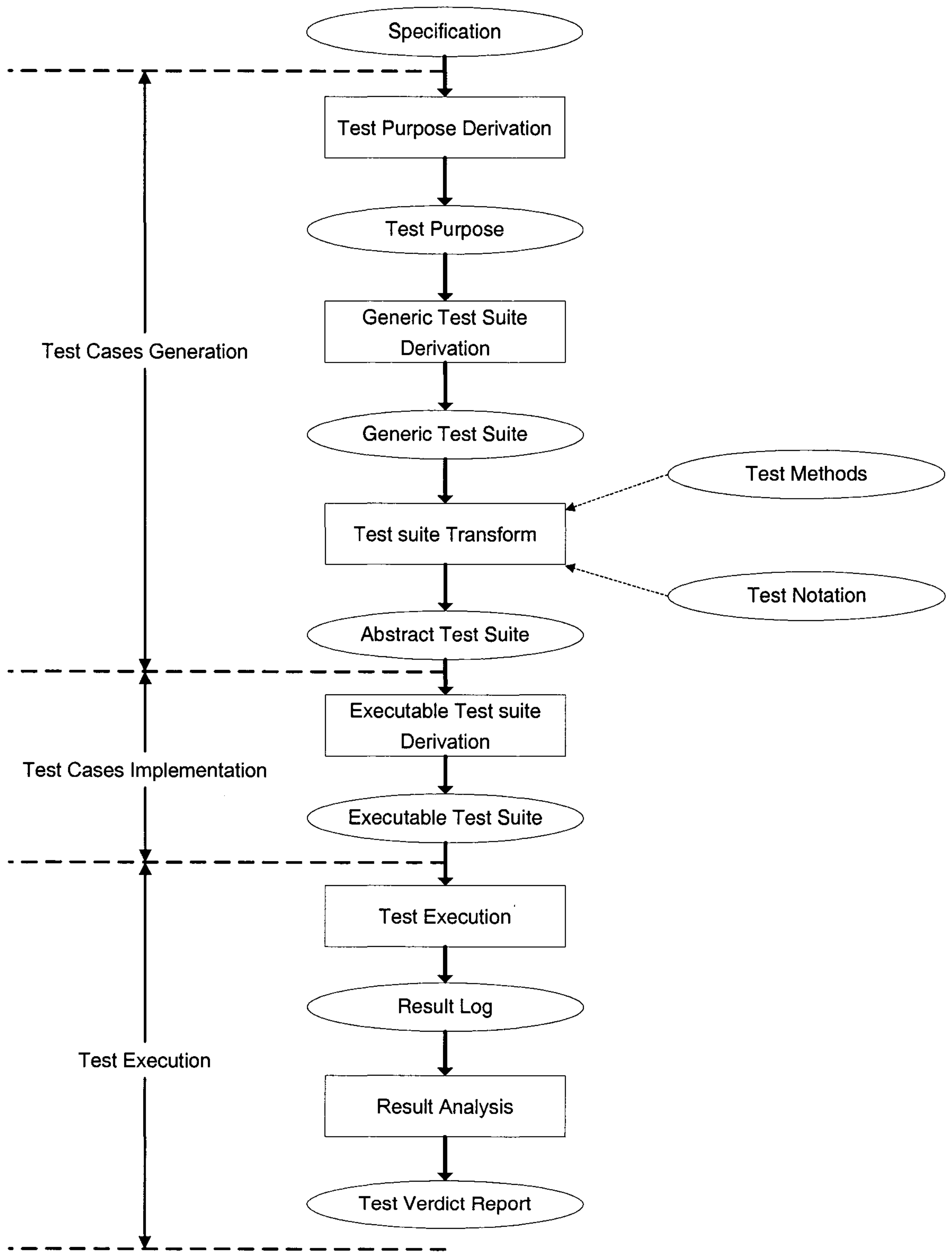


Figure 2-1 Conformance Testing Framework

2.2.1 Test cases generation

In the test cases generation step, test purposes are firstly derived deliberately based on the requirements of the specification to answer the question of "What are we testing". Then, generic test cases are devised carefully to realize these test purposes. A generic test case is an operationalization of a test purpose, in which the actions necessary to achieve the test purpose are described on a high level, without considering the test method or the environment in which the actual testing will be done [25]. It is structured by a sequential of ordered events that are interactive between the tester and the implementation under test. A generic test case can be divided into three parts: the preamble part, the test body, and the postamle part. The preamble is a sequence of events that leads the IUT to the state where the behavior corresponding to the test purpose starts. The test body controls the behavior of the IUT corresponding to the test purpose. The postamble leads to a sequence of external outputs for a verdict and bring the IUT to a neutral state after the test body has been executed. All these test cases together compose a complete test suite, which can be applied to the tested object to get a verdict on whether the IUT satisfies all requirements of the specification. Based on generic test cases, we derive an abstract test case for each generic test case under the consideration of particular test architecture and restrictions implied by the environment.

During the generation of an abstract test suite, test architecture need to be considered. For different architectures, test cases are different. There are four basic types of test architectures recommended in IS9646 standard, namely: Local Single-Layer Architecture, Distributed Single-Layer Architecture, Coordinated Single-Layer Architecture, and Remote Single-Layer Architecture (shown in Figure 2-2). These architectures are designed for the OSI protocol model testing, but they can be also used in testing other protocols and distributed systems. The test architectures are composed of a lower tester, an upper tester, the IUT, the SUT (System Under Test), test coordination protocols, and PCOs (Points of Control and Observation). A Lower tester behaves as the lower layer of the IUT and peer-layer of the IUT. It creates services (as lower layer) and sends PDUs (as peer-layer) to control the behaviors of the IUT through PCOs on lower boundary and observes the reactions of the IUT through PCOs. A Lower tester is also responsible for test verdict and test report generation. An Upper tester behaves as the upper layer of the IUT that asks for services from the IUT through PCO on upper boundary. PCOs are service access points which can be accessed by testers. They are points through which testers can control and observe behaviors of the IUT. In the OSI protocol model, Service Access Points, SAPs, are introduced as points through which service entities in the same or adjacent layers communicate with each other. The test coordination protocol is used to keep coordination and management between the upper tester and the lower tester. In Local Single-Layer Architecture, the upper tester, the lower tester and the IUT are in a same system. In Distributed Single-Layer Architecture, the lower tester is distant from the IUT and needs to communicate with the IUT indirectly through underlying communication services. Coordinated Single-Layer Architecture is the same as Distributed Single-Layer Architecture except that the lower tester has coordination with the upper tester through the test coordinating protocol. The difference between the Remote Single-Layer Architecture and the Distributed Single-Layer Architecture is that, in the former, there is no upper tester but multiple upper layers which have been tested instead.

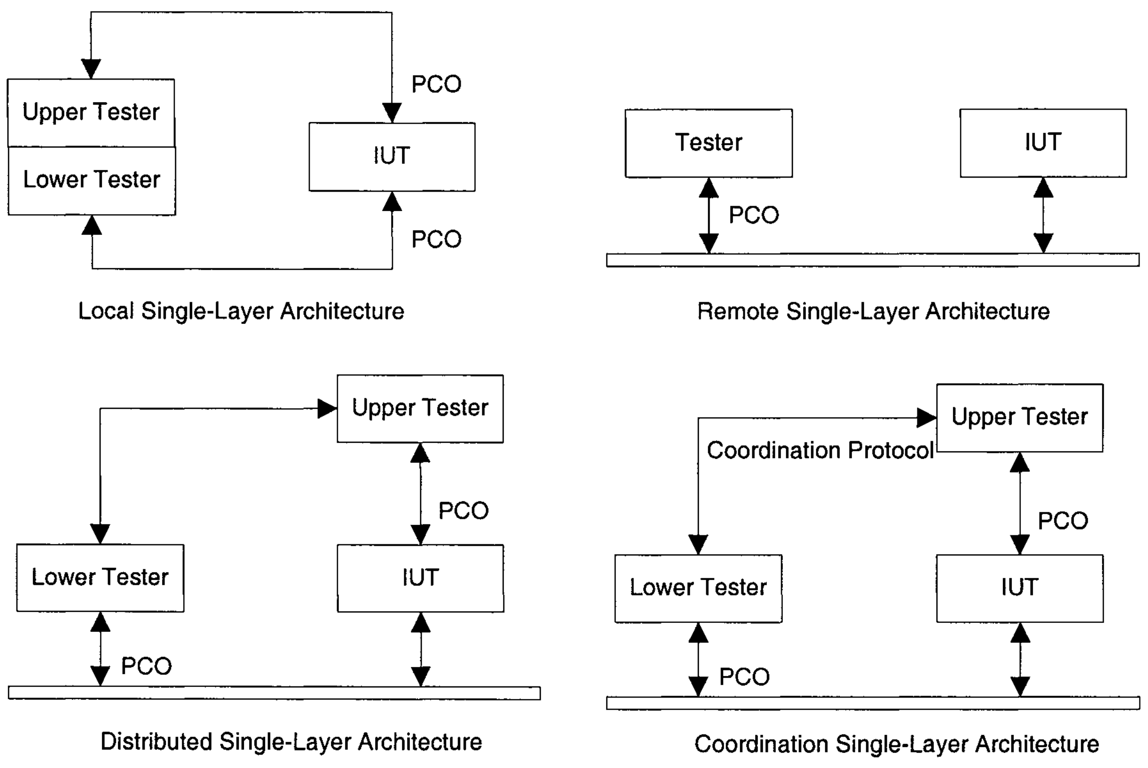

Upper Tester
Lower Tester
PCO
IUT
PCO
Local Single-Layer Architecture
Tester
IUT
PCO
Remote Single-Layer Architecture
Upper Tester
PCO
Lower Tester
IUT
PCO
Distributed Single-Layer Architecture
Upper Tester
Coordination Protocol
PCO
Lower Tester
IUT
PCO
Coordination Single-Layer Architecture

Figure 2-2 Four basic types of test architecture

2.2.2 Test cases implementation

In the test cases implementation step, abstract test cases from the last step are translated into executable cases with real factors for the specific IUT under the specific testing environment. Because abstract test cases are independent from any implementations, some factors in the test cases (eg. parameters and data for service calling) cannot be real value. These factors are different in different implementations. During test cases implementation step, these abstract factors need to be concretized. Abstract test cases are derived as the test specification for testing. For each implementation, we do not need to implement all abstract test cases but select ones which are concerned with the implementation. The abstract test cases selected need to be translated into the executable cases that the test device can recognize. Different test devices need different translators.

2.2.3 Test execution

In the test execution step, testing is executed through applying the concretized test cases from the implementation step. The reactions of the IUT are observed and a verdict is given for each test case. A "Pass" verdict indicates that the observed behavior conforms to the reference specification and the test purpose is covered during the execution of the test. A "Fail" verdict indicates that the observed behavior contradicts with the reference specification. An "Inconclusive" verdict indicates that the observed behavior conforms to the reference specification but the test purpose is not covered during the execution of the test. If all test cases applied on the IUT lead to the verdict "Pass", it means that this IUT conforms to its specification.

2.3 Formal conformance testing

In the last section we overviewed the standard of conformance testing procedure recommended by ISO and ITU-T (formerly CCITT). In that standard, the specification is assumed to be described in a natural language. In practice, especially in protocol design, more specifications are described in a formal description technology (FDT) such as SDL, LOTOS, Estelle and Z. With the formal specification, we can generate abstract test cases automatically. This is called formal methods.

In formal conformance testing, the definition of conformance is different from it in IS9646. In IS9646, the conformance is defined as satisfying the requirements of the specification. But in formal methods, the requirements of the specification are not described explicitly. Instead, the specification behaviors are described in terms of events which may be observed at PCOs. In this case, we use conformance relation instead of requirements to filter out the conformance implementations. Implementations are supposed to conform to the specification if they hold certain conformance relation with the specification. The conformance relation is established based on the mathematical models of the specification and implementations. In FSM model, for example, this conformance relation can be defined as equivalence relation, quasi-equivalence relation, and reduction relation.

There may be two development directions in generating test cases from a formal specification. One is deriving test cases directly from the formal specification. The test suite devised in this method is used to test the conformance relation between the specification and its implementations. But this method is hard to implement when the tested system is complicated and difficult to be understood. The other is transforming the formal specification into another formal model, called intermediate model, and then deriving tests from the intermediate model [27]. The test suite devised in this method is used to test conformance relation between the intermediate model and the implementations. This method is more practical because: first, comparing to the former model, the intermediate model is simpler in most cases because it only covers the specific aspects of the system to be tested. It is easier to devise test cases from simpler models with the fewer number of test cases. Second, usually the intermediate models are strict mathematic models on which many research works have been done both in theory and in implementations. For example, many works have been done on testing finite state machine [28] [29] [30] [31] [32] and extended finite state machine [33] [34] [35] [36] [37]. So the results obtained from these models are convincible and also formally provable. These models are usually Finite State Machines(FSMs), Input/Output Finite

State Machines (I/O FSMs), Extended Finite State machines (EFSMs), Labeled Transition Systems (LTSs), Asynchronous Communication Trees (ACTs) and Charts.

Testing coverage is another key point in formal testing area. To measure the quality and adequacy of a testing method, testing coverage is one of important measurement norms. The choice of testing coverage depends on the testing paradigm. There are two paradigms commonly used in testing [26]: one is exhibiting correct behavior concerning the criteria in question; the other is discovering any implementation faults in relation to the criteria in question. In the first paradigm, test cases are generated based on the structure of the specification model or its intermediate model, such as traces and data flows. Transition Tour [4] is an example of this kind of paradigm with trace equivalence relation. In this case, we often use structural coverage to measure the quality and adequacy of the test suite devised. A big problem in this paradigm is that it is hard to reach enough structural coverage because too many traces need to be covered, especially in the case of loops. Importing test purposes with reasonable hypotheses and assumptions can decrease covering traces by defining the traces that the tester is interested in and skipping others. But this kind of branches coverage will leave a few faults undetected.

In the second paradigm, test cases are generated based on the fault model of the specification or its intermediate model. Wp method [4] is an example of testing under this paradigm with trace equivalence. The fault model is the description of the effects of failures at some higher level of abstraction [4]. It limits the number of implementations by mutation approach with the regularity assumption. In this case, we use fault coverage to measure the quality and adequacy of the test suite devised. Obviously some kinds of faults uncovered by the fault model will be left undetected.

Above, we have overviewed formal techniques in automatically deriving an abstract test suite. The translation of an abstract test suite into an executable test suite can also be automatic with the help of TTCN (Test Tabular Combined Notation) [38]. TTCN is a formal language which is standardized by ISO for the test system specification. If abstract test cases are described in TTCN, they will be feasible to be executed automatically by any tester who supports TTCN inputs. Many works have been done on generating TTCN test cases from formal models [44] [45].

2.4 Conclusion

In this chapter we overviewed the conformance testing in software engineering. We explained how it works under the framework of IS9646. Moreover, we introduced the formal conformance testing based on formal models. These models gave rise to methods of automatic generation of test cases. In the following chapters we will focus on automatic generation of test suites for embedded real-time systems based on timed input output automata model. We will firstly introduce some basic concepts in our models in the chapter 3. Then, we will discuss our methodology and algorithms in detail in chapter 4.

Chapter 3
Related backgrounds and Basic Concepts

In the last chapter, we gave readers an overview of conformance testing. As our thesis is about conformance testing of embedded real-time systems, in this chapter, we firstly give a brief introduction to research works on testing real-time systems. Then, we provide some main concepts and definitions, which are related to the formal models of our approach.

3.1 Testing Real-Time Systems: Backgrounds

The behaviors of real-time systems are time sensitive. Most malfunctions of this kind of systems happen because of the unsatisfaction of the time constraints in the specification. It is hard to find out these faults with testing techniques, which do not include time factors in their formal models. To cope with the timing properties in real-time systems, many formal languages are created or extended to offer capabilities of analyzing, designing, implementing and testing real-time systems, such as Timed Input Output Automata (TIOA), SDL, MSC, Constraint Graph, Extended Temporal Logic, etc. Research works have been done for test methods based on models described in these languages [1], [2], [3], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14] [40].

There are two types of real-time properties: hard real-time properties and performance properties [4]. Hard real-time properties state the time constraints (time bounds) to actions of systems. Performance properties state the statistical nature of systems. In conformance testing, we only care about hard real-time properties and leave performance properties to performance testing. As we mentioned earlier, test cases are composed of sequences of actions. Considering hard real-time properties and continuous time domain, it means infinite set of test cases need to be generated. But a test suite needs to be finite for practical reasons. To solve this contradiction, one way is discretizing time over time domain. For example, some researchers [3] discretizes time by introducing clock region concept in the time domain and then, sampling in the time domain. All states in each clock region are supposed equivalent and can be represented by one state. There are possibly other ways that can be used to solve this contradiction in testing. For example, for each test case, testers can be designed to start actions randomly following a certain statistical model in time.

In integration testing, the IUT is not tested in isolation. It is tested with other components of the system. For the IUT, these components are its contexts, which interact directly or indirectly with the IUT. These interactions are carried on through internal interface and they are unobservable to testers. Testers can only control and observe the IUT behaviors partially through PCOs. This kind of testing is known as embedded testing or testing in context [38][41] [42].

Our methodology is based on TIOA models. It follows the theories of testing embedded real-time systems [11] and testing with test purposes [1]. The basic concepts we used in our methodology are introduced in the next section.

3.2 Basic Concepts

3.2.1 Timed Input Output Automaton

A Timed Input Output Automaton (TIOA) [17, 2, 13] A is a tuple (I_A, O_A, L_A, l^0_A, C_A, T_A), where:

- I_A is a finite set of input actions. Each input action is denoted by "?" followed by a label.
- O_A is a finite set of output actions. Each output action is denoted by "!" followed by a label.
- L_A is a finite set of locations. The term "location" is chosen instead of term "state" because the latter is used to define the operational semantics (i.e., all the executions) of TIOA.
- $l^0_A \in L_A$ is the initial location.
- C_A is a finite set of synchronous clocks set to zero in l^0_A. We assume that time is dense, which means that all clocks have values are real numbers.
- T_A is the set of transitions. Each transition consists of a source location, an input or an output action, a clock guard that should hold in order to execute the transition, a set of clocks to be reset when the transition is executed, and a destination location. We assume that all transitions are instantaneous.

Figure 3-1 is an example of TIOA that describes the behavior of the train component in RCS. First, the train sends the signal *APPROACH* to the environment to indicate that it's approaching. Then, the train sends the signal *ENTRY* to the controller to let it know that it is at the entry sensor. Then within [60s, 120s], the train should send *INCROSS* signal to the environment to indicate that it is in the crossing region. At this time, the gate should be closed to make sure that no other train could enter the crossing region. When the train leaves the crossing region, it sends the signal *EXIT* to the controller within [0,180s] after it sends the *ENTRY* signal. Next, the train sends the signal *LEAVE* to the environment to show that it is out of the crossing region. The TIOA specification of the train has five locations *A0* (the initial location), *A1*, *A2*, *A3* and *A4*, five transitions and one clock *x*. The transition from *A2* to *A3*, denoted by $A2 \xrightarrow{!INCROSS,60<=x<=120} A3$, is executed when the train sends the signal *INCROSS* and the value of clock *x* is between *60* and *120* inclusively. When the transition is fired, the clock *x* is set to *0*.

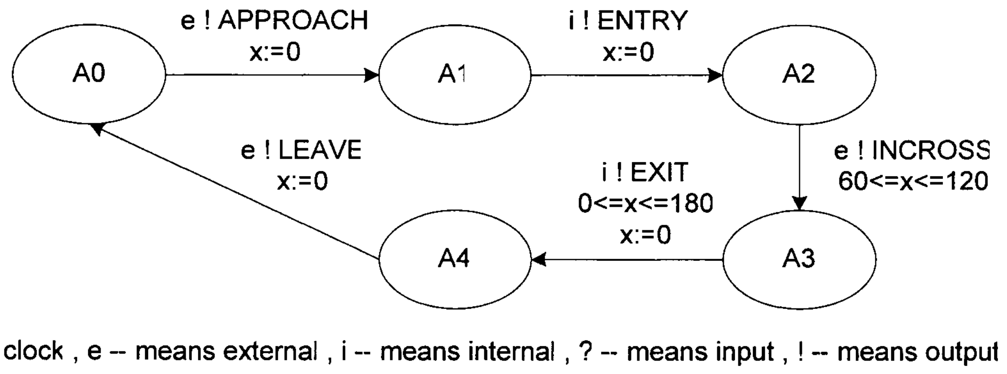


Figure 3-1 The specification of Train described by TIOA

3.2.2 Grid Automaton

Let $A = (I_A, O_A, L_A, l^0_A, C_A, T_A)$ be a TIOA. The Grid Automaton (*GA*) of *A* [2, 3, 19, 13] is a finite input output automaton $GA = (I_{GA}, O_{GA}, S_{GA}, s^0_{GA}, T_{GA})$, where:

- $I_{GA} = I_A \cup \{g\}$, where *g* is a special time delay (*g* is a rational number).
- $O_{GA} = O_A$.

- S_{GA} is a finite set of system states. Each state is a pair (l, v), where $l \in L_A$ and v is a clock valuation in which the value of each clock is a multiple of g.
- s^0_{GA} is the initial state that consists of the initial location , l^0_A , and the clock valuation that set all clocks' value to zero.
- T_{GA} is a finite set of Transitions.

Each transition consists of a source state, an action (input, output, or time delay g), and a destination state. There are two types of transitions in GA: the delay transitions on time delay g and the explicit transitions on input and output actions. Each state has an outgoing delay transition on time delay g. However, a state (l, v) has an outgoing explicit transition on input or output action a if and only if there is a transition $l \xrightarrow{\{?,!\}a,G,\lambda} l'$ in A and v satisfies the clock guard G. After the execution of a delay transition on g, the value of each clock is incremented with g time-units. However, after the execution of an explicit transition the value of each clock in λ in the corresponding transition in A is set to zero.

The grid automaton represents a subset of the executions of a TIOA because it only contains the delay transitions on the same time delay g. However, the executions of TIOA contain delay transitions on every time delay between 0 and 1 and all the delays are obtained by the addition of those delays since we are using dense time model [3].

3.2.3 Synchronous Product of Two TIOAs

Let $S = (I_S, O_S, L_S, l_S^0, C_S, T_S)$ and $T = (I_T, O_T, L_T, l_T^0, C_T, T_T)$ be two TIOAs. The synchronous product of S and T is a special composition $SP = (I_{SP}, O_{SP}, L_{SP}, l_{SP}^0, C_{SP}, T_{SP})$ of S and T such that [1, 14]:

- $I_{SP} = I_S \cup I_T$ and $O_{SP} = O_S \cup O_T$.
- $L_{SP} \subseteq L_S \times L_T$.
- $l^0{}_S = (l^0{}_S, l^0{}_T)$.
- $C_{SP} = C_S \cup C_T$.
- L_{SP} and T_{SP} are the smallest relations defined by the following two rules:

  a. Suppose $l_{SP} = (l_S, l_T) \in L_{SP}$, $l_S \in L_S, l_T \in L_T, l_S' \in L_S$, then

$$\left.\begin{array}{l} l_S \xrightarrow{\{?,!\}a,G1,\lambda1} l_S' \in T_S \\ l_T \xrightarrow{\{?,!\}a,G2,\lambda2} l_T' \notin T_T \end{array}\right\} \Rightarrow l_{SP}' = (l_S', l_T) \in L_{SP} \text{ and } l_{SP} \xrightarrow{\{?,!\}a,G1,\lambda1} l_{SP}' \in T_{SP}$$

  b. Suppose $l_{SP} = (l_S, l_T) \in L_{SP}$, $l_S \in L_S, l_T \in L_T, l_S' \in L_S, l_T' \in L_T$, then

$$\left.\begin{array}{l} l_S \xrightarrow{\{?,!\}a,G1,\lambda1} l_S' \in T_S \\ l_T \xrightarrow{\{?,!\}a,G2,\lambda2} l_T' \in T_T \end{array}\right\} \Rightarrow l_{SP}' = (l_S', l_T') \in L_{SP} \text{ and } l_{SP} \xrightarrow{\{?,!\}a,G1\&G2,\lambda1\cup\lambda2} l_{SP}' \in T_{SP}$$

3.2.4 Message Sequence Chart

A Message Sequence Chart (MSC)[16] is a structure $M = <P, S, R, A, O, T, T_X, Y>$, where:

- P is a finite set of processes.
- 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, and timer stop.
- O is the ordering of S, R and A. We assume a total ordering relation among S, R, and A, which means we know, by O, which event happens first and which one happens next.
- 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.

We also define an instance (process) P_i in a MSC $T < P, S, R, A, O, T, T_X, Y >$, $P_i \in P$, as a tuple of $< S_i, R_i, A_i, O_i, T_i, T_{Xi}, Y_i, >$, where,

- $S_i \subseteq S$, a finite set of message events sent by P_i.
- $R_i \subseteq R$, a finite set of message events received by P_i.
- $A_i \subseteq A$, a finite set of local events in P_i.
- $O_i \subseteq O$, the ordering S_i, R_i and A_i on instance P_i.
- $T_i \subseteq T$, a finite set of timers which are associated with P_i.
- $T_{Xi} \subseteq T$, associates each timer related event with its timer.

- $Y_i \subseteq Y$, associates each pair of dependent events in $S_i \cup R_i$ with its timing restriction, and associates each action in A_i with its duration.

There are two kinds of MSC diagrams: Basic Message Sequence Chart (BMSC) and High-Level Message Sequence Chart (HMSC). BMSC is used to describe simple scenarios of messages flow between instances. HMSC is used to describe complicated structures of BMSCs composition such as coregion, inline expression (alternative, parallel composition, iteration, exception and optional regions) and MSC reference.

3.2.5 Test Purpose

A Test purpose is a partial behavior of the system under test. It represents a sequence of interactions among the components of the system and with the environment as well as the time constraints on these interactions.

In this thesis, test purposes are expressed using MSC 2000 [15]. MSC 2000 provides the user with constructs to specify the timing behavior of a real-time system not only by timers but also with time constraints between any pair of events. Moreover, a MSC specification can be converted into a TIOA under some conditions.

An example of test purposes in MSC 2000 is given in Figure 3-2. In this example, The user is interested in checking if the implementation of RCS allows the train to send the signal *INCROSS* followed by the signal *EXIT*, have the controller send the signal *RAISE* to the gate at the latest 30 time-units after getting the signal *EXIT*, and have the gate open (signal *UP*) within 60 to 120 time-units after the reception of the signal *RAISE*.

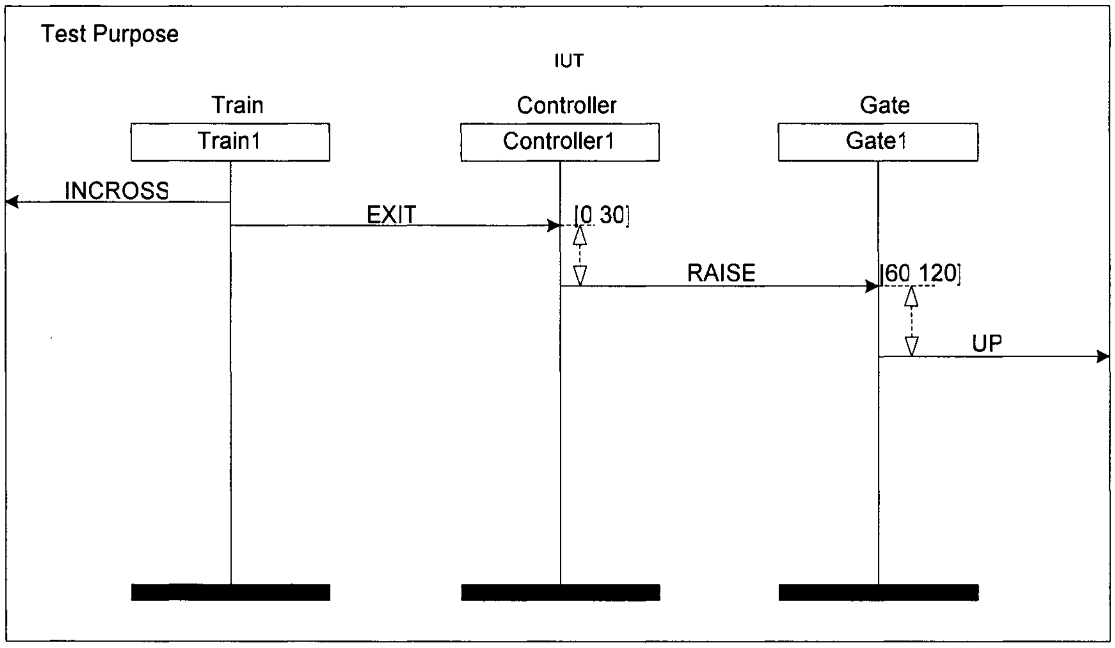


Figure 3-2 Test Purpose for Train going out of the cross region

3.2.6 Communicating TIOA Model (CTIOA)

A Communicating Timed Input output Automaton (CTIOA) [11] is formally defined as a tuple $CT = (I_{ct}, O_{ct}, L_{ct}, l^0_{ct}, C_{ct}, T_{ct}, F_{ct})$, where

- I_{ct} is a finite set of input actions. Each input begins with "?".
- O_{ct} is a finite set of output actions. Each output begins with "!".
- L_{ct} is a finite set of locations.
- $l^0_{ct} \in L_{ct}$ is the initial location.
- C_{ct} is a finite set of clocks all initialized to zero in l^0_{ct}.
- T_{ct} is a finite set of transitions.
- F_{ct} is a FIFO channel.

3.2.7 Partial Product of CTIOAs

Let (A_1, A_2, A_3... A_m) be m CTIOAs describing an embedded real-time system. The product of these CTIOAs is a global TIOA $G = (I_G, O_G, L_G, l^0_G, C_G, T_G)$ [10], where:

- I_G is the union of the input sets of A_1, A_2, A_3... A_m.
- O_G is the union of the output sets of A_1, A_2, A_3... A_m.
- L_G is a finite set of global locations. A global location is a tuple $(l_1, l_2, \ldots, l_m, F_1, F_2, \ldots, F_m)$, where l_i, $1 \le i \le m$, is a location in A_i, and F_i is the FIFO associated with A_i, for all $1 \le i \le m$.
- $l^0_G = (l^0_1, l^0_2, \ldots, l^0_m, F^0_1, F^0_2, \ldots, F^0_m)$ is the initial global location, where l^0_i is the initial location of A_i and $F^0_i = \Phi$, for $1 \le i \le .m$.
- C_G is the union of the clock sets of A_1, A_2, A_3... A_m.
- T_G is a finite set of global transitions. A global transition is a transition between two global locations caused by a transition in a component of the system.

3.3 Conclusion

In this chapter we gave a brief introduction on testing real-time systems. We provided the basic concepts on timed input output automaton model, grid automaton, synchronous product, message sequence chart, test purpose, communicating TIOA model, and partial product of communication automata. These concepts are very important to our methodology.

In chapter 4, we will discuss our methodology step by step and introduce our algorithms on MSC translation, transition paths selection, synchronous product construction, partial product construction, differential sampling, and test cases generation.

Chapter 4
A New Approach for Testing Embedded RTS

In this chapter we present a new methodology that is developed to generate test cases for embedded real-time systems with test purposes and the specification. We will explain the ideas of the methodology and provide algorithms on MSC (Message Sequence Charts) translation, transition path selection, synchronous product construction, partial product construction, differential sampling, and test cases generation. Test Purposes are described by MSC and the specification is described by TIOA (Timed Input Output Automata).

4.1 Overview of our Approach

Our approach [43] consists of six main operations: (1) the parsing of the specification and test purposes, (2) the selection of the transitions to be considered for test cases generation, (3) the construction of the synchronous product of test purposes and TIOA specifications, (4) the construction of a partial product for the system under test, (5) the sampling of TIOAs, and (6) the generation of test cases.

The sampling operations can be executed in different levels in an order of these six operations. Each level of sampling operation gives rise to a new method for test cases generation for embedded real-time systems. We implemented and studied three of these methods. The difference between these methods is the position of sampling in the whole process. These methods are illustrated in Figure 4-1 and we will refer to them as M1, M2, and M3 respectively.

In what follows, we present in more detail each of the aforementioned operations and point out the specificities of methods M1, M2, and M3.

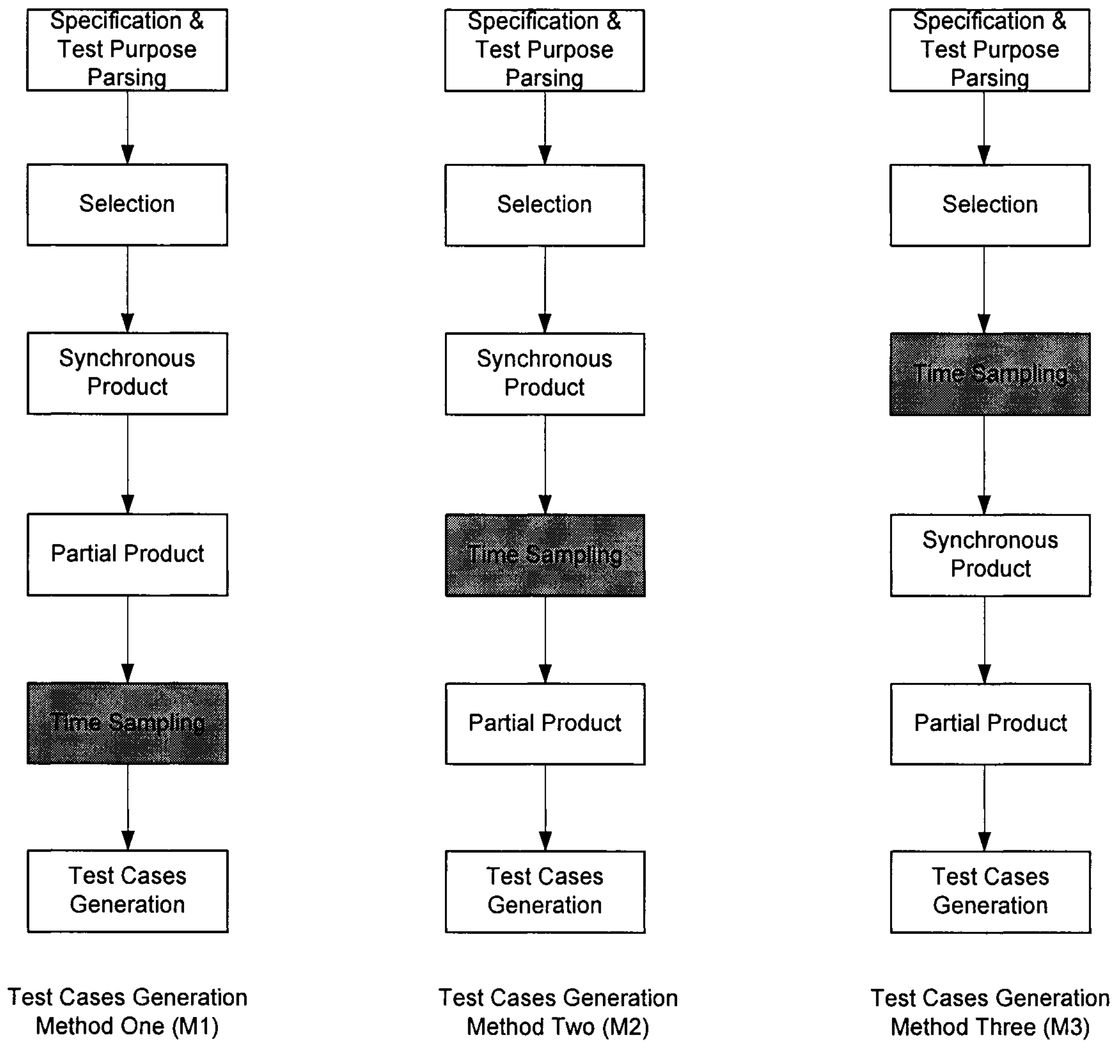


Figure 4-1 Three Methods of Test Cases Generation

4.2 Translate Basic MSC into TIOA

In our methodology, we suppose that test purposes are offered by the tester. As MSCs are widely used in engineering practices for modeling distributed interactive systems, we use them to describe our test purposes. We use Timed Input Output Automata (TIOA) model to describe the specification since it is a formal language that is widespread in computing applications with its depth and mathematic precision of description. Because our methodology is based on the theory of Timed Input Output Automata, we need to translate MSC into TIOA. Each instance of MSC (i.e., test purpose) gives rise to a TIOA. Obviously, such a TIOA is acyclic with no self-loop transitions.

4.2.1 Mapping MSC Into TIOA

To translate MSC into TIOA, we need to establish a mapping between a MSC and a TIOA.

First, we map each instance in a MSC into a TIOA. From automata theory, we know that transitions are driven by inputs/outputs. These inputs/outputs are interactions between an automaton and the outsiders. In our thesis, we assume that internal actions do not cause any state transition and do not need to be considered. We describe the translation algorithm in mathematics as follows: let $T < P, S, R, A, O, T, T_X, Y >$ be a test purpose in BMSC. According to the definition of a test purpose, P is a finite set of instances (processes). For each $p_i \in P$, by using translation function f, we find a TIOA M_i such that $M_i = f(P_i)$.

Second, our translation algorithm is based on events. We use events as the connection to map an instance of a MSC into a TIOA. In MSCs, sequences of events are used to describe interactions between instances and the environment. One instance has interactions with both other instances and with the environment through events. An event is a behavior of sending or receiving a message. Obviously after each event happened, the state of the concerning instance should be changed. If we define a state s_1 of an instance before an event e happens and another state s_2 after, we can say that the state of the instance is transferred from s_1 to s_2 after event e happenes. This is just like state transitions that are driven by inputs/outputs in automata. Thus, if we use inputs/outputs to describe events and create states before and after events happen, then we can get an automaton with which we can describe the behavior of an instance in MSC.

Now we discuss the translation function in detail. Suppose $P_i< S_i,R_i,A_i,O_i,T_i,T_{Xi},Y_i>$ is a process in a test purpose of MSC $T < P,S,R,A,O,T,T_X,Y >$ and $M_i<I,O,L,l^0,C,T>$ is a specification of TIOA, then we define a translation function as $M_i = f(P_i)$. The mapping function $f = f_1 \bullet f_2 \bullet f_3 \bullet f_4$ ("$\bullet$" is a composition operator) can be defined as follows:

- Map events into inputs/outputs. $I \cup O = f_1(S_i \cup R_i \cup A_i, T_i, T_{Xi})$

  In MSC, events can be divided into three categories from IUT point of view: message sending events (outgoing events), message receiving events (incoming events), and local events. Both message sending events and message receiving events belong to message exchanging actions and can be directly mapped into output actions and input actions of a TIOA M_i. For local events (actions and timers), currently we only consider timer events (start timer, stop timer and timeout) and map them into input actions in TIOA M_i. We do not include the local actions because we ignore the situation with data variables. In this case, action events do not influence the system states.

- Create locations with events in order. Let $l \in L, e \in S_i \cup R_i \cup A_i$, then $l = f_2(e, O_i)$

  There are no elements corresponding to states in MSC. In order to translate instances in MSC into automata, we need to generate states for each instance.

The states generation function f_2 should follow three rules: first, there must be an initial location l_0 for each process/instance. Second, for each event belonging to an instance/process, there should be two states, one is before the event happening, the other is after the event happening. That is, for each $e \in S_i \cup R_i \cup A_i$, generate states s_1 and s_2 with $s_1 \xrightarrow{e} s_2$. Finally, two adjacent states between which there are no events happening should be combined into one state.

- Map MSC timing constraints into TIOA clocks. That is, $C = f_3(Y_i)$.

Timing constraints are introduced in MSC2000 to describe time-sensitive systems. It supports the notion of quantified time for the description of real-time systems with precise meaning of the sequence of events in time [16]. With timing constraints, we can define the time point of an event occurring, or the time period between two events happening. It is similar to what clocks do in TIOA.

The mapping rules are simple: for each timing constraint without interval label, we generate a unique clock as the correspondent; for the time constraints with interval labels, we generate one clock corresponding to each interval label.

- Generate transitions $T = f_4(X, Y_i)$, where X is the result of functions f_1, f_2, and f_3.

In TIOA, a transition t can be expressed as $t = l \xrightarrow{e,G,\lambda} l'$. It consists of a source location l, an input or an output action e, a clock guard G that should be satisfied when the transition executes, a set of clocks λ that are reset after the transition is executed, and the destination location is l'. The source location, destination location and their associated input/output action are from the results of function f_1 and f_2. G and λ are obtained from the timing constraints, which are associated with event e. Clocks in G and λ are from the results of function f_3. The logic relations in G can be abstracted from Y_i, so do the clocks in λ. We will discuss it in detail in test purpose parsing section.

From the description above, we can see that our translation algorithm is based on one-instance-one-TIOA mapping. Because of the limitation of TIOA model, there are shortages on the mapping of ordering and timing constraints.

In MSC definition, ordering is divided into two categories, normal ordering and general ordering. Normal ordering is given by the MSC semantics. It assumes the time order along the instance axis. Events on this instance should follow this order. Ordering among the events of different instances is decided via messages. That is, a message must be sent before it is received. But, sometimes we need to introduce more complicated ordering among different instances, for example, to specify that an event on one instance just happen before an unrelated event on another instance[16]. In this case we need to declare the ordering explicitly with keywords. This kind of ordering is defined as general ordering in MSC.

In our definition of an instance, $O_i \subseteq O$ only includes the ordering along one instance axis. We ignore any ordering among instances. It is because we need to translate one instance into one TIOA and in a TIOA there is no element that is used to describe the ordering among TIOAs. But with the help of the unique signature of each message, we can recover a part of the ordering information. It is clearly shown when we construct partial product with TIOAs. We can recover general ordering information by establishing a global ordering table for each MSC. But, this is outside the Instance-TIAO translation algorithm and currently we do not include it in our methodology. We assume that overlooking the general ordering will not influence the correctness of our methodology because our results include all the possibilities that satisfy the general ordering requirements.

The mapping of timing constraints also has the same problem as ordering in our Instance-TIOA translation. Timing constraints can be used to measure not only the time distance between the two events happening on the same instance but also the time distance between the two events happening on different instances. But, the clock set *C* in a TIOA model can only include the clocks that are used to measure the time distance between any two events happening on the same TIOA. In other words, there is no way to measure the time distance between the two events happening on different TIOAs. For example, in a MSC, instance *A* sends a message *m* to instance *B*. It requires that the time period from *A* sending *m* till *B* receiving *m* must be within *s* ms. We cannot describe this kind of time constraint in a TIOA model when we use the translation function introduced above. To overcome this shortage, one of possible solutions is to improve TIOA model by adding communication channels with time delay and global clocks which will be used to measure the time distance among events on different TIOAs. This solution can be considered in our future works and will not be included in this thesis.

4.2.2 Translation algorithm

Based on the previous discussion, we design our translation algorithm as follows:

```
Input:
    a MSC chart $Ch<\mathrm{P,S,R,A,O,T,T_X,Y}>$
Output :
    an automata set S. $S=\{A<I,O,L,l^0,C,T>\}$
Step0: /* Initialize the set to use: */
    S = $\Phi$
Step1: /* Translate each instance in MSC chart C into automata
object and save the result automata object in the set S.*/

  For each instance $P_i$ in Ch, do
    Create automata A<I, O, L, $l^0$, C, T>, where
      $A:I\cup O=P_i:f_1(S_i\cup R_i\cup A_i,T_i,T_{Xi})$
      $A:C=P_i:f_3(Y_i)$
     Create initial location $A:l_0$
     Set current location $A:l=l_0$
     For each event $e\in P_i:S_i\cup R_i\cup A_i$, do
       Create new location $A:l'=P_i:f_2(e,O_i),l'\in L_i$ after event e
happened
       Create transition $A:t=l\xrightarrow{e,G,\lambda}l'$ by function $f_4$
       Set current location $l=l'$
     End For
     Save automaton A into set S, that is $S=S\cup\{A\}$
  End For
```

4.3 The Selection of Transitions

As we mentioned earlier, a test purpose describes a partial behavior of the system under test. It defines which system is under test and which part of behaviors of the system needs to be tested. Under the guidance of a test purpose, we can confine our test cases generation by concentrating on these specific behaviors of the system and ignoring the others that are not concerned. This means it will be helpful to avoid the well-known states explosion problem.

We would like to mention that we could not generate test cases only from a test purpose. In our assumption, test purposes are offered by the customers who, in most cases, do not know the whole systems very well. The description of the behaviors in test purposes could be incomplete, or there could be mistakes in it. So the test cases could be invalid if they have been generated only based on test purposes and thus the testing results based on these test cases would be unreliable. To overcome the shortages of test purposes, we combine test purposes with the system specification, which is supposed to be a complete and accurate description of the system. The result of combination should cover both the test purpose and the corresponding part of the specification. The whole combination procedure is divided into two steps: first, selecting traces from the system specification according to the test purpose and composing new TIOA models; second, construct a synchronous product with traces in test purpose and traces selected from the system specification. We introduce selection part in this section and leave the synchronous product construction part to the next section.

4.3.1 IUT (Implementation Under Test) and its context

During the selection process, we distinguish between the TIOA specification of the IUT and the rest of the TIOA specifications, called the context of the IUT. The IUT is the testing object for which we generate test cases. In an embedded real-time system, we describe the IUT as one of the relatively independent components which have interactions between each other and the environment. Because the IUT has communications with other components, the behaviors of these components have influences on the behavior of the IUT and should be considered as the context during test cases generation. In our methodology, we extract the specification of these components as the contexts of the IUT and combine them with the IUT specification. The combination is called the partial product of the system and will be introduced in section 4.5.

In our embedded real-time system model, the components of the IUT and its contexts are communicating with each other through FIFO (First-In-First-Out) channels. The length of a FIFO channel is set to one for simplicity, which means that only one message in the channel at a time. Figure 4-2 gives us a simple example of this kind of system models. In this example, the IUT interacts with two components of context *system1* and *system2*. *system1* sends message *msg2* to FIFO channel *C1*. The IUT receives input message *msg2* from channel *C1* and changes its location from location *B0* to location *B1*. After that, the IUT changes its location from *B1* to *B0* and sends message *msg3* to FIFO channel *C2*. *system2* catches message *msg3* from channel *C2* and changes its location from *C0* to *C1*. From this example, we can see how our model describes interactions among the components of embedded real-time systems. We also notice that, in the example, the transition from location *A1* to *A0* in *system1* causes the transition from location *B0* to *B1* in the IUT. Both of the transitions share the same message *msg2*. We call these two transitions a relative pair. In a relative pair, one transition is caused by a message output; its correspondent part is caused by the same message. These two transitions should be in two different components. Two transitions are considered as a relative pair if they share the same message and one transition is caused by an input, the other is caused by an output.

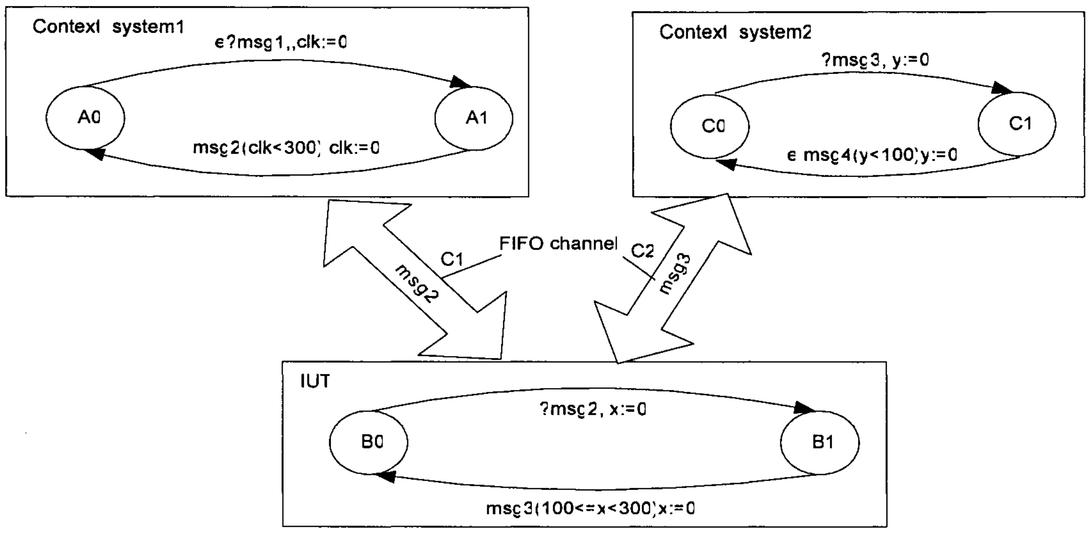


Figure 4-2 An example of a embedded real-time system model

4.3.2 Transitions' selection

The objective of the selection process is the extraction of the transitions necessary for test cases generation. Only the transitions of the specifications that correspond to the test purpose and the transitions of the context influencing (or influenced by) the IUT are necessary for test cases generation. As we mentioned before, we do not need to generate test cases to test every aspect of the IUT. We generate the test cases only for verifying the test purpose offered by customers. In order to reach our goal, for each scenario in the test purpose, we extract traces accordingly from the specification $S=\{A_0, A_1,..., A_n\}$ and compose a new TIOAs subset $S'=\{B_0, B_1,..., B_m\}$. Then, we combine *S'* with TIOAs of the test purpose through synchronous product and partial product, and generate test cases based on the combination result at the end. Note that *S'* is composed of TIOAs of the IUT and its context. It is a subset of *S*, $S' \subseteq S$. For each TIOA $B<I,O,L,l^0,C,T>$ in *S'*, there is a TIOA $A<I,O,L,l^0,C,T>$ in *S* from which *B* is extracted, where:

- $B.I \subseteq A.I$; The input set of *B* is a subset of the inputs set of *A*.

- $B.O \subseteq A.O$; The output set of B is a subset of the output set of A.
- $B.L \subseteq A.L$; The location set of B is a subset of the location set of A.
- $B.l^0 = A.l^0$; The initial location of B is equal to the initial location of A.
- $B.C \subseteq A.C$; The clock set of B is a subset of the clock set of A.
- $B.T \subseteq A.T$; The transition set of B is a subset of the transition set of A.

To implement the selection operation, we assume the followings. First, we suppose that each instance in the test purpose has its corresponding specification and we use Instance Kind in test purpose and TIOA Name in the specification to establish this correspondence. Second, we assume that for each internal input/output event in each TIOA of the specification, there is one and only one corresponding internal output/input event in another TIOA in the specification. The second assumption makes it easy to search relative pair when we do selection.

The selection operation is carried out in two steps:

In the first step, we have two objectives to achieve: one is extracting transitions from the specification according to the test purpose and composing new TIOAs; the other is extracting all context TIOA components, which influence (or are influenced by) the transitions of the IUT. For each trace of each instance in a test purpose, we select the transition path in the corresponding specification. With these transitions, we compose a new TIOA, whose location set, clock set, input/output set and transition set are subsets of the ones in the original TIOA. The path should start at the initial location. It must include the transitions that are on the same events as the corresponding trace of an instance in the test purpose. It means that the events on the path should cover all the events on the trace in the same order. After each new TIOA is generated, we find its context through relative pair relation among transitions. These contexts are also extracted from the specification. The transitions in any of these contexts compose a transition path, which is corresponding to the transition paths in other contexts and in the new TIOA.

In the second step, we search for a postamble to reach an external output, which is observed and conclude a verdict. After the first step, we get a TIOA set, which is a subset of the specification. The TIOAs in this set compose a sub-system model whose behavior covers the scenario described in the test purpose. But the system described with the model may not be testable under black-box test because the model can not surely give us an observable output to conclude a verdict. We need to supplement the model by adding an observable output to it. The output should satisfy two requirements: first, the output must be an external output, which can be detected by the tester. The interactions among the components of the system are unobservable from outside. Second, the transition incurred by the output should be behind, in order, the last transition of the path in the TIOA model of the IUT. The only exception is when the event on the last transition is an external output. The required output is not necessarily from the IUT. It could be from the contexts of the IUT.

4.3.3 Algorithm designed for step one

During the first step of the selection process, we use depth-first traversal, starting from the initial location, to walk through the specification tree till finding the path we need. There are three main requirements that must be satisfied: first, the selected transition paths must cover the scenario of the test purpose. The sub-specification generated after the selection process is scenario oriented. Second, the selected paths must keep correlations among them. For any transition driven by an internal event in a selected path, there must be an internal event, as its counterpart, happening in another selected path. Third, the transitions in the selected paths must be consistent. Since transitions in a TIOA are time guarded, when picking up a transition from the specification model and adding it to the selected path, we must check if this transition is consistent with other transitions of the selected path in the time domain. If not, we will not add it into the selected path.

To satisfy the first two requirements above, we define a *Constraint* set in our algorithm to constrain a path selection. Each *Constraint* in the *Constraint* set reflects a requirement from the test purpose, or from one of other selected paths. It comprises a sequence of events and requires the path to be selected to cover the transitions driven by these events in the same order. These events are counterparts of internal events from one of selected paths, or from the corresponding instance in the test purpose.

To start the paths selection process, we firstly choose the components models that have corresponding instances in the test purpose from the specification. We initialize a *Constraint* set for each model by generating a *Constraint* with the requirement from the corresponding instance in the test purpose. The following algorithm describes the initialization process:

```
Step0:
Initialize set Q = $\Phi$.
/* Q is a collection of TIOAs to be proceeded */

For each $TIOA_i$ of the specification
/*(i=1,2,…,n, n is the number of TIOAs in the specification)*/

   If $TIOA_i$ has instance in the test purpose, then

         Create a Constraint set $CS_i$

         Initialize $CS_i$ = {$C_i^0$},
/* $C_i^0$ is Constraint from corresponding instance in test
purpose.*/

        Q = Q ∪ { $TIOA_i$ }

   End If

End For
```

Then, for each *$TIOA_i$* model in the collection *Q*, we walk through the transitions tree with depth-first traversal in order to find the transition path *P* under *Constraints* in *CS_i*. If *P* has correlated requirements to other TIOA models, say *$TIOA_k$*, we create *Constraints* to represent these requirements and update corresponding *Constraint* set, say *CS_k*, with these new *Constraints*. If there is any requirement for the TIOA models outside of *Q*, we add that TIOA model to *Q*.

```
Step1:
For each $TIOA_i$ in Q

    Depth-first search transition tree till finding path P
    satisfying Constraints in $CS_i$

    Update Constraint sets $CS_k$, k =1,2, .., n, $k \neq i$

    If $TIOA_k \notin Q$, update by Q =  Q ∪ { $TIOA_k$ }

End For
```

We repeat step1 till no *CS_i* is updated, $i = 0,1,\ldots,n$.

The flow chart in Figure 4-3 describes graphically the algorithm.

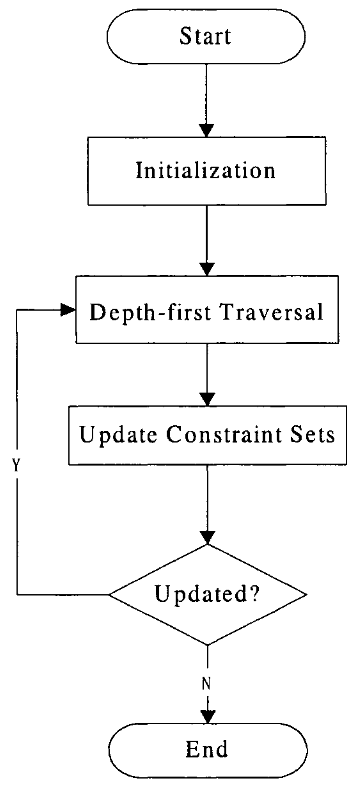


Figure 4-3 Flow chart of selection algorithm (first step)

4.3.4 Algorithm designed for step two

In the second step, we try to find postambles where an external output event can be used to decide whether the tests pass or fail. If the tester observes the output during a reasonable period of time, then we can say that this test case passes; otherwise, it fails. Not any external output event can take this role. The event must have orderable relation with the last event happening in the IUT. That is, it must surely happen after the last event of the IUT. It does not need to be an external event in the IUT. We do not need to find postamble anymore if this kind of event exists in the result of our first step selection.

Considering the orderable relations among the TIOAs of the specification, in our algorithm, we compose a global location tree, and use breadth-first traversal to search for the external output transition we need. The root location of the global location tree is defined as (l_0', l_1', ..., l_m', l_{m+1}^0, ..., l_n^0), where (0, 1, ... m, m+1,... n) are the indices of the TIOAs in the specification. TIOAs (0, 1,..., m) are the TIOAs that compose the result in first step and the locations (l_0', l_1', ..., l_m') are the last locations of them. TIOAs(m+1,...,n) are the rest of TIOAs in the specification and the locations (l_{m+1}^0, ..., l_n^0) are their initial locations. The algorithm for postamble searching is given below:

```
Input:
    A TIOA set S = {$TIOA_0$,$TIOA_1$, …, $TIOA_m$, $TIOA_{m+1}$, …, $TIOA_n$}
Output:
    A TIOA set S' = {$TIOA_0'$,$TIOA_1'$, …, $TIOA_m'$, $TIOA_{m+1}'$, …,
    $TIOA_k'$}, m+1≤k≤n

Step0: Compose initial global location.
         $gl_0 = (l_0', l_1', ..., l_m', l_{m+1}^0, ..., l_n^0)$ ;
Step1: Initialize the sets to use:
         $SS = \{gl_0\}$ ; /* source location set */
         $DS = \Phi$ ;  /* destination location set */
         $FS = \{f_0 = 1, f_1 = 0, ..., f_n = 0\}$ ;  /*flags set. $f_i = 1$ means
external events in $TIOA_i$ can be used as a judgement.*/

Step2: Search for external output event.
         While $SS \neq \Phi$ do
             For each global location $gl_i \in SS$ do
                 For each $TIOA_j$ location $l_j$ in $gl_i$ do
                     Get transition $t = l_j \xrightarrow{(?,!)e,G,\lambda} l_j'$ ;
                     Test consistency of t, if pass then
                   If e is external output event and $f_j = 1$ then
                       Exit While;
                   End If
                   If e is internal output event and $f_j = 1$ then
                       $f_m = 1$ ;  /* if $\bar{e} \in TIOA_m(I,O,L,l_0C,T).I$ ) */
                   End If
        Compose a new global location $gl_i'$ with $l_j'$ instead of $l_j$ ;
                   $DS = DS \cup \{gl_i'\}$ ;
                   End If
                 End For
             End For
             $SS = DS$ ;
             $DS = \Phi$ ;
         End While
```

4.4 Synchronous Product

The objective of this step is to construct the synchronous product of the specification and the test purpose according to the definition in 3.2.3. At this stage, the test purpose is described by a TIOA because its corresponding MSC has been parsed and converted into TIOA in step 1 (i.e., the parsing operation). The computation of a synchronous product has at least two advantages. On the one hand, the combination of a test purpose with its specification helps in avoiding the state space explosion problem since the test purpose restricts the behavior to be tested. On the other hand, the synchronous product serves to validate the test purpose against its specification since the user could include some non-executable and/or inconsistent behaviors in his/her test purpose.

In addition to the general rules stated in section 3.2.3, three new rules are added here to conclude a verdict during the execution of test cases:

1. Pass rule: A Pass verdict is concluded if a test case satisfies both the specification and test purpose.
2. Inconclusive rule: An inconclusive verdict is concluded if a test case satisfies the specification but fails the test purpose.
3. Fail rule: A fail verdict is concluded if a test case fails the specification.

These three verdict rules are assigned to the states/locations of synchronous product by adding two dummy states/locations in the synchronous product. One dummy state/location corresponds to the inconclusive case and is called "*Inconclusive*" state/location; the other dummy state/location corresponds to the fail case and is called "*Fail*" state/location. With these two dummy states/locations, we complete our synchronous product in the time domain. This will be helpful for test cases generation later. In fact, test cases will be generated based on some test selection criteria such as the coverage of all paths leading to Pass verdicts, the coverage of one path leading to each Pass verdict, etc.

The synchronous product construction is not the same for methods M1, M2, and M3. For methods M1 and M2, the construction of synchronous product is finished before the sampling step (see section 4.6). However, for method M3 the synchronous product is constructed after the sampling step is terminated. This is the reason why M3 generates less states and test cases than M1 and M2. In fact, the earlier the time sampling is, the fewer states and fewer test cases are generated. It should be noted here that the algorithm of constructing a synchronous product in M3 is not exactly the same as the algorithm in M1 and M2 because the algorithm in M3 takes as inputs two grid automata instead of two TIOAs.

4.4.1 Synchronous product construction – Algorithm one

In this section, we introduce the algorithm of synchronous product construction with two TIOA as inputs. One TIOA is from the specification, the other is from the test purpose. In our description, we use *S* to denote the TIOA from the specification, *T* to denote the TIOA from the test purpose, and *SP* to denote the TIOA of the synchronous product. We also use *l* to denote a location and *t* to denote a transition. The algorithm is executed in breadth-first traversal. It generates transitions of synchronous product layer by layer till reaching leaves. The algorithm is described as follows:

```
Step0: Initialize the sets to use.
           $H$ = $\Phi$ ; /* a set of locations to be handled */
           $R$ = $\Phi$ ;  /* Reachable locations set */

Step1: Construct the initial location of $SP$, $SP.l_0$, which is the
       simple combination of initial location of TIOA $S$, $S.l_0$,
       and initial location of TIOA $T$, $T.l_0$. That is,
           $SP.l_0$ = ( $S.l_0$, $T.l_0$ );

Step2: Construct synchronous product.
         Add initial location $SP.l_0$ into set $H$. That is, $H = \{ SP.l_0 \}$ ;
         Assign $SP.I$ = $S.I$ ; $SP.O$ = $S.O$ ; $SP.I$ = $S.C \cup T.C$ ;
         While $H \neq \Phi$ do
            For each $SP.l_k$ = ( $S.l_i$, $T.l_j$ ) in $H$, do
              Get a transition from $S.T$, whose starting
              location is $S.l_i$, that is,
                 $S.t = S.l_i \xrightarrow{\{?,!\}e1,G1,\lambda 1} S.l_{i+1}$ ;
              Get a transition from $T.T$, whose starting
              location is $T.l_j$, that is,
                 $T.t = T.l_j \xrightarrow{\{?,!\}e2,G2,\lambda 2} T.l_{j+1}$ ;
              Compare the two events $e1$ and $e2$, If $e1$ = $e2$, then
                 Create a new synchronous product location
                     $SP.l_{k+1}$ = ($S.l_{i+1}$, $T.l_{j+1}$) ;
                 Create a new transition
                     $SP.t = SP.l_k \xrightarrow{\{?,!\}e1,G1\&G2,\lambda 1\cup\lambda 2} SP.l_{k+1}$ ;
                 Create a "Inconclusive" transition
                     $SP.t_{Inconclusive} = SP.l_k \xrightarrow{\{?,!\}e1,G1\&\overline{G2},\lambda 1\cup\lambda 2} SP.Inconclusive$ ;
                 Create a "Fail" transition
                     $SP.t_{Fail} = SP.l_k \xrightarrow{\{?,!\}e1,\overline{G1},\lambda 1\cup\lambda 2} SP.Fail$ ;
              Else
                 Create a new synchronous product location
                     $SP.l_{k+1}$ = ($S.l_{i+1}$, $T.l_j$) ;
                 Create a new transition
                     $SP.t = SP.l_k \xrightarrow{\{?,!\}e1,G1,\lambda 1} SP.l_{k+1}$ ;
                 Create a "Fail" transition
                     $SP.t_{Fail} = SP.l_k \xrightarrow{\{?,!\}e1,\overline{G1},\lambda 1} SP.Fail$ ;
              End If
       If both location components of $SP.l_{k+1}$ are not leaves, then
                 Add the new location $SP.l_{k+1}$ into set $R$. That is
                    $R$ = R $\cup$ { $SP.l_{k+1}$ };
       End If
            End For
            Assign $H$ = $R$;
            Clear the set $R$. $R$ = $\Phi$
         End While
```

4.4.2 Synchronous product construction – Algorithm two

In this section, we introduce the algorithm of the synchronous product construction with two grid automata as inputs. One grid automaton is from the specification, the other is from the test purpose. The output is the grid automaton of the synchronous product. A grid automaton is the result of sampling TIOA behaviors in the time domain. Its definition was introduced in section 3.2.2.

A state of synchronous product, denoted as G_{sp}, is a combination of two states: one is a state from the grid automaton of the specification, denoted as $G_s = (l_s, [v])$; the other is a state from the grid automaton of the test purpose, denoted as $G_t = (l_t, [v])$. That is, $G_{sp} = (G_s, G_t)$.

There are two types of transitions in grid automata, one is the transitions driven by interactions among systems and the environment without time consuming. The other is delay transitions used to describe time increments. We treat these two kinds of transitions differently in our algorithm. For transitions under inputs/outputs actions, we first check the source state G_{sp} to see whether the clocks' values of G_{sp} satisfy the guard condition. If they satisfy the guard condition, we generate the destination state G_{sp}' and reset the clocks' values if needed. If only G_s satisfies the guard condition, we lead the transition to "*Inconclusive*"state. If G_s does not satisfy the guard condition, we lead the transition to "*Fail*" state.

In the case of delay transitions, we do not need to check whether the source state satisfies the transition guard condition. We only generate the new destination state G_{sp}' for the synchronous product by adding the delay time value d to the clocks' values of G_{sp}. That is, $G_{sp}' = (G_s', G_t')$, where $G_s' = (l_s, [v+d])$, $G_t' = (l_t, [v+d])$. But, there is a problem here if the grid automata from the specification and the test purpose are generated with different sampling units. We can not make sure that there is a state as G_s' in the grid automaton from the specification or a state as G_t' in the grid automaton from the test purpose because they are on different time sampling units. In this case, the transitions tree in a grid automaton is a dynamic one and has to grow accordingly. That is, we generate the missing states and the missing branches for the grid automata from the specification and the test purpose.

In our description, we use S to denote the grid automaton from the specification, T to denote the grid automaton from the test purpose, and SP to denote the grid automaton of the synchronous product. We also use G to denote a grid location (state) and t to denote a transition.

The algorithm is executed in breadth-first traversal. It generates transitions of the synchronous product layer by layer until reaching leaves. The algorithm is described as follows:

```
Step0: Initialize the sets to use.
            $H$ = $\Phi$ ; /* a set of grid locations to be
                           Handled */
            $R$ = $\Phi$  ; /* Reachable grid locations set */
            $D$ = $\Phi$  ; /* a set of delay time value */

Step1: Construct the initial grid location of $SP$, $SP.G_0$, which
       is the simple combination of the initial location of
       the grid TIOA $S$, $S.G_0$, and the initial location of the
       grid TIOA $T$, $T.G_0$. That is,
            $SP.G_0 = (S.G_0, T.G_0)$ ;

Step2: Construct synchronous product.
          Add the initial location $SP.G_0$ into the set $H$.
          That is,
             $H = \{ SP.G_0 \}$ ;
          Assign $SP.I = S.I$ ; $SP.O = S.O$ ;
          While $H \neq \Phi$ do
          For each $SP.l_k$ = ( $S.l_i$, $T.l_j$ ) in $H$, do
            Get a transition from $S.T$, whose starting
            location is $S.l_i$, that is,
              $S.t = S.G_i \xrightarrow{\{?,!\}e1,G1,\lambda 1} S.G_{i+1}$ ;
            Get a transition from $T.T$, whose starting
            location is $T.l_j$, that is,
              $T.t = T.G_j \xrightarrow{\{?,!\}e2,G2,\lambda 2} T.G_{j+1}$ ;
            If $S.t$ is a delay transition, then
              Save delay time into set $D$ if no same
              value in $D$ ;
            Else if T.t is a delay transition, then
              Save delay time into set $D$ if no same
              value in $D$ ;
            Else
/*******************************************************/
/*   this is for transition under input output action   */
/*******************************************************/

              Compare the two events $e1$ and $e2$,
              If $e1$ = $e2$, then,
                If $S.G_{i+1}$ is not "Fail" state and
                $T.G_{j+1}$ is not "Fail" state, then,
                  Create a new synchronous product
                  Location
                    $SP.G_{k+1}$ = ($S.G_{i+1}$, $T.G_{j+1}$) ;
                  Create a new transition
                    $SP.t = SP.G_k \xrightarrow{\{?,!\}e1,G1\&G2,\lambda 1\cup\lambda 2} SP.G_{k+1}$ ;
                End if

                If $S.G_{i+1}$ is not "Fail" state and $T.G_{j+1}$ is
                "Fail" state, then,
                  Create a "Inconclusive" transition
                    $SP.t_{Inconclusive} = SP.G_k \xrightarrow{\{?,!\}e1,G1\&G2,\lambda 1\cup\lambda 2} SP.Inconclusive$ ;
                End if

                If $S.G_{i+1}$ is "Fail" state, then
                  Create a "Fail" transition
                    $SP.t_{Fail} = SP.G_k \xrightarrow{\{?,!\}e1,\overline{G1},\lambda 1\cup\lambda 2} SP.Fail$ ;
                End if
              Else
                If $S.G_{i+1}$ is not "Fail" state, then
                  Create a new synchronous product Location
                    $SP.G_{k+1}$ = ($S.G_{i+1}$, $T.G_j$) ;
                  Create a new transition
                    $SP.t = SP.G_k \xrightarrow{\{?,!\}e1,G1,\lambda 1} SP.G_{k+1}$
                Else
                Create a "Fail" transition
                  $SP.t_{Fail} = SP.G_k \xrightarrow{\{?,!\}e1,\overline{G1},\lambda1} SP.Fail$
              End if
            End If

            If both location components of $SP.G_{k+1}$ are not
            leaves, then
              Add the new location $SP.G_{k+1}$ into set $R$.
              That is, $R = R \cup \{SP.G_{k+1}\}$ ;
            End If

/**********************   End   ***************************/

/*********************************************************/
/*            This part is for delay transition          */
/*********************************************************/
            For each time delay $d$ in set $D$, do
              Create a new synchronous product location
                $SP.G_{k+1} = (S.G_{i+1}+d,\ T.G_j+d)$ ;
              Create a new delay transition
                $SP.t = SP.G_k \xrightarrow{d} SP.G_{k+1}$ ;
              Complete transition tree if no state $S.G_{i+1}+d$
              in $S$ ;
              Complete transition tree if no state $T.G_{j+1}+d$
              in $T$ ;
              Add the new location $SP.G_{k+1}$ into set $R$,
                $R = R \cup \{SP.G_{k+1}\}$ ;
            End For

************************   End   ***************************
          End For
          Assign $H = R$ ;
          Clear the set $R$. $R = \Phi$ ;
        End While
```

4.5 Partial Product

The ultimate goal of this operation is to construct the partial product of the IUT and its context using only the transitions chosen by the selection operation as explained previously. The construction of the partial product helps us avoid the state explosion problem that arises when a complete product is constructed. This is true because not all the transitions of the context of the IUT have direct or indirect relationship with the transitions in the IUT. The partial product algorithm is a simple translation of its definition in 3.2.7. The critical point in the algorithm is the correspondence between internal inputs and internal outputs. In other words, any internal input should be consumed only after its corresponding internal output is produced. We realize this relation by the FIFO channel. When an internal output is produced by a component, it is placed in the corresponding FIFO channel, and when an internal input is consumed by a component it is removed from the corresponding FIFO channel. We assume that the IUT and its context start their execution at the same time with all clocks set to zero.

Note that the partial product construction is the step where M1 and M2 start to be different in order. For M1, the sampling step is done after the construction of the partial product. However in M2, the sampling step is done before the construction of the partial product. This means that M2 should generate fewer states than M1.

Similar to the synchronous product, there are also two different partial product algorithms: one is for TIOAs and the other is for grid automata.

4.5.1 Partial product construction – Algorithm one

In this section, we introduce the algorithm of the partial product construction with a set of TIOA as inputs. The output is a global TIOA, which is designed to describe the behaviors of the IUT and its context. A location in a global TIOA is called global location and it is the combination of the locations from the IUT and its context. The input/output set in a global TIOA is the union of the input/output set of the IUT and its context. So is the clock set in a global TIOA. Transitions in a global TIOA are partially ordered to keep the correspondence between internal inputs and internal outputs. FIFO channels are introduced into a global TIOA as the media of interactions among the IUT and its context.

The algorithm is executed in the breadth-first traversal. It generates transitions of the partial product layer by layer until reaching leaves. The algorithm is described as follows:

```
Input:
        A set of TIOA $\{A_1, A_2, ..., A_n\}$ i = 1, 2, ..., n
Output:
        Global TIOA $G$

Step0: Initialize the sets to use:

         $S = \Phi$ ; /* a set of global locations to be
                    Handled */
         $D = \Phi$ ; /*reachable global locations set */

Step1: Construct the initial global location of $G$, $G.l_0$, which
       is the simple combination of initial locations of
       $\{A_1, A_2, ..., A_n\}$, That is,

           $G.l_0 = ( A_1.l_0, A_2.l_0, ..., A_n.l_0 )$ ;

Step2: Construct partial product

     $S = \{ G.l_0 \}$ ;   /* Add initial location $G.l_0$ into set $S$ */
     Assign $G.InputOutSet = A_1.InputOutputSet \cup ... \cup A_n.InputOutputSet$
            $G.ClockSet = A_1.ClockSet \cup ... \cup A_n.ClockSet$ ;
     While $S \neq \Phi$, do
         For each global location $G.l_i$ in $S$, do
               For each component location $A_x.l \in G.l_i$, do
                   Get a transition from TIOA $A_x$
                     $A_x.t = A_x.l \xrightarrow{\{?,!\}a,G,\lambda} A_x.l'$ ;

                   If event $a$ is an internal incoming event and
                   $a$ is in a FIFO channel, then
                       Create a next global location $G.l_{i+1}$,
                         $G.l_{i+1} = ( A_1.l, ..., A_x.l', ..., A_n.l )$ ;
                       Create a global transition
                         $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
                       Add the global location $G.l_{i+1}$ to set $D$
                         $D = D \cup \{G.l_{i+1}\}$ ;
```

```
                Empty the FIFO channel ;
            End If
            If event $a$ is an internal outgoing event and
            the FIFO channel is empty, then
                Send $a$ to the FIFO channel ;
                Create a next global location $G.l_{i+1}$,
                  $G.l_{i+1}$ = ( $A_1.l, \ldots, A_x.l', \ldots, A_n.l$ ) ;
                Create a global transition
                  $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
                Add a global location $G.l_{i+1}$ to set $D$
                  $D = D \cup \{G.l_{i+1}\}$ ;
            End If

            If event $a$ is external event, then
                Create the next global location $G.l_{i+1}$,
                  $G.l_{i+1}$ = ( $A_1.l, \ldots, A_x.l', \ldots, A_n.l$ ) ;
                Create a global transition
                  $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
                Add a global location $G.l_{i+1}$ to set $D$
                  $D = D \cup \{G.l_{i+1}\}$ ;
            End If

        End For
    End For
    Assign $S$ = $D$ ;
    Empty set $D$, that is $D$ = $\Phi$ ;
End While
```

4.5.2 Partial product construction – Algorithm two

In this section, we introduce the algorithm of partial product construction with a set of grid automata as inputs. A grid automaton is the result of sampling the behaviors of a TIOA in the time domain. Its definition is introduced in section 3.2.2. The definition is similar to the definition of TIOA except that, in a grid automaton, we use states (location plus time information) instead of locations. The output is a grid global TIOA.

The algorithm is executed in the breadth-first traversal. It generates transitions for the partial product layer by layer until reaching leaves. For each layer, delay transitions with the same delay time period must be combined as one. The algorithm is described as follows:

```
Input:
    A set of grid TIOA objects {$A_1$, $A_2$ ,..., $A_n$} i = 1, 2, ..., n

Output:
    Grid global TIOA object $G$

Step0: Initialize the sets to use:
        $S = \Phi$ ; /* a set of states to be handled */
        $D = \Phi$ ; /* reachable states set */

Step1: Construct the initial global state of $G$, $G.l_0$, which is
       the simple combination of initial state of {$A_1$, $A_2$ ,...,
       $A_n$}, That is,

        $G.l_0 = ( A_1.l_0, A_2.l_0 ,..., A_n.l_0 )$

Step2: Construct the partial product

        Add the initial global state $G.l_0$ into set $S$.
          $S = \{ G.l_0 \}$ ;
        Assign
          $G.InputOutSet = A_1.InputOutputSet \cup ... \cup A_n.InputOutputSet$
          $G.ClockSet = A_1.ClockSet \cup ... \cup A_n.ClockSet$ ;
        While $S \neq \Phi$, do
          For each global state $G.l_i$ in $S$, do
            For each component state $A_x.l \in G.l_i$, do
              Get a transition from TIOA $A_x$,
                $A_x.t = A_x.l \xrightarrow{\{?,!\}a,G,\lambda} A_x.l'$ ;
              Else If $A_x.l'$ is "Inconclusive" state, then
                Create a global transition
                  $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.Inconclusive$ ;

              Else If event $a$ is internal incoming event
              and $a$ is in a FIFO channel, then
                Create a next global state $G.l_{i+1}$,
                  $G.l_{i+1} = ( A_1.l, ..., A_x.l', ..., A_n.l )$ ;
                Create a global transition
                  $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
                Add a global state $G.l_{i+1}$ to set $D$
                  $D = D \cup \{G.l_{i+1}\}$ ;
                Empty FIFO channel ;
          Else If event $a$ is internal outgoing event
          and FIFO channel is empty, then
            Send $a$ to a FIFO channel ;
            Create a next global state $G.l_{i+1}$,
              $G.l_{i+1}$ = ( $A_1.l, \ldots, A_x.l', \ldots, A_n.l$ ) ;
            Create a global transition
              $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
            Add a global state $G.l_{i+1}$ to set $D$
              $D = D \cup \{G.l_{i+1}\}$ ;

          Else If event $a$ is external event, then
            Create a next global state $G.l_{i+1}$,
              $G.l_{i+1}$ = ( $A_1.l, \ldots, A_x.l', \ldots, A_n.l$ ) ;
            Create a global transition
              $G.t = G.l_i \xrightarrow{\{?,!\}a,G,\lambda} G.l_{i+1}$ ;
            Add a global state $G.l_{i+1}$ to set $D$
              $D = D \cup \{G.l_{i+1}\}$ ;

          Else If event $a$ is delay event with delay
          time $d$, then
            Create a next global state $G.l_{i+1}$,
              $G.l_{i+1}$ = ( $A_1.l+d, \ldots, A_x.l+d, \ldots, A_n.l+d$ ) ;
            Complete the transition tree if no $A_m.l+d$ state
            in $A_m$, $m=1,2\ldots n$ ;
            Create a delay transition
              $G.t = G.l_i \xrightarrow{d} G.l_{i+1}$ ;
            Add a global state $G.l_{i+1}$ to set $D$
              $D = D \cup \{G.l_{i+1}\}$ ;
          End If
        End For
      End For
      Assign $S$ = $D$ ;
      Empty set $D$, $D$ = $\Phi$ ;
    End While
```

4.6 Time Sampling

The objective of time sampling is to construct the GA (Grid Automata) of the test purpose TIOA, the specification TIOA, the synchronous product TIOA, and the partial product TIOA depending on which of the methods M1, M2, or M3 is used. Indeed, when method M1 is used, the GA to be constructed is the one corresponding to the partial product of the system under test. But, if method M2 is used then we construct the GA of the synchronous product. However, if method M3 is used then we should construct the GA of each specification TIOA and the GA of each test purpose TIOA.

The way the time domain is sampled is very important because it greatly affects the number of states to be generated as well as the length of the test suite. Therefore, the key question is how to choose the sampling unit. The sampling unit, which we refer to as the granularity of sampling (the symbol g in section 3.2.2), represents the amount of time the system is allowed to pass from one time zone to another in order to execute explicit transitions on input and output actions.

According to [1] [3] [20], the granularity of sampling should be less than or equal to $\frac{1}{n+1}$ in order to be able to cover all the clock regions of a TIOA with n clocks. The problem of this method is numerous states generated when the number of clocks and the bounds of constraints are bigger. In this thesis, we alleviate this problem by proposing different formulas for sampling the time space of the system under test.

For M1:

$$(1)\ SampleUnit = \frac{TheLeastTimePeriodInTransitionsOfGlobalTIOA}{TheNumberOfClocks + 1}, \textit{ where,}$$

$$TheLeastTimePeriodInTransitionsOfGlobalTIOA = Min\{TheLeastTimePeriodInATransition_x\}$$

$x = 1,2,3,\ldots,n$, *n is the number of transitions in a global TIOA.*

$$TheLeastTimePeriodInATransition_x = Min\{LowBound - 0, UpBound - LowBound\}$$

For M2 and M3:

(2) $SampleUnit = Min\{SampleUnit_x\}, x = 1,2,...,n$, *n is the number of TIOAs.*

Where, $SampleUnit_x = \dfrac{TheLeastTimePeriodInTransitionsOfTIOA_x}{TheNumberOfClocks_x + 1}$

$TheLeastTimePeriodInTransitionsOfTIOA_x = Min\{TheLeastTimePeriodInATransition_i\}$
$i = 1,2,3,...,n$, *n is the number of transitions in a TIOA.*

$TheLeastTimePeriodInATransition_i = Min\{Min\{LowBound - 0, UpBound - LowBound\}_k\}$
k=1,2,3,..,N, the number of clocks in constraints.

These two formulas are used when we sample the time space with the same granularity for each transition in a TIOA. We can also sample the time space with different granularities based on the difference between the lower and upper bounds of each transition, as follows:

(3) $SampleUnit_x = \dfrac{Min\{\, Min\{\, LowBound - 0, UpBound - LowBound\,\}_k\,\}}{TheNumberOfClocks + 1}$

x=1, 2, 3...N, is the index of transitions in TIOA, and k =1, 2, 3...N, is the index of clocks in time constraints of a transition.

The idea of sampling with different granularities is based on the fact that the periods of clock regions in different transitions can be different according to their time constraints. For example, there are two transitions in the transition set of a TIOA with one clock t: one has a time constraint of 0<t<1, the other has a time constraint of 0<t<2. Therefore, there are four clock regions in each transition, t=0, 0<t<1, t=1, t>1 for the first one and t=0, 0<t<2, t=2, t>2 for the second. With our former sampling algorithm we need to sample this TIOA with the period of 0.5. It is an efficient sampling for the first transition. But it is not efficient for the second one because points t=0.5, t=1,t=1.5 are equal and can be represented with one point t=1. We can see that, as the number of clocks and transitions in a TIOA is increasing, more useless points are generated with the former algorithm. Sampling with different sample period is a way to decrease the number of these points.

The algorithm is executed in the breadth-first traversal. It generates transitions of grid TIOA layer by layer till reaching leaves. The algorithm is described as follows:

```
Input:  TIOA $A$

Output: Grid TIOA $W$

Step0: Compute the granularity g with formula (1) for the
       method one and formula (2) for the method two and
       three.   /* this step is only for sampling with the
                   same granularity. */

Step1: Initialize the sets to use:
         $S = \Phi$  ; /* a set of states to be handled */
         $D = \Phi$  ; /* reachable states set */

Step2: Construct the initial state of $W$, $W.s_0$,
         $W.s_0$ = ($A.l_0$, 0);
Step3: Construct a grid automaton $W$
         $W.InputOutSet = A.InputOutputSet$ ;
         $W.ClockSet = A.ClockSet$ ;
         Save initial state $W.s_0$ into set $S$ ;

    While $S \neq \Phi$, do
      For each state $W.s_i$ = ($A.l_i$, v) $\in S$, do
        Get transition $A.t = A.l_i \xrightarrow{\{?,!\}a,G,\lambda} A.l_i'$ ;
        Calculate granularity $g = f(t)$ ;
          /*formula (3), only for sampling with different
            Granularities */
        While time point value v< $\infty$, do
          If state $W.s_i$ satitify the guard condition of
          transition $t$, then
             Create state $s_i'$ such that $s_i' = (l_i', [\lambda \mapsto 0]v)$ ;
             Create transition $W.t$
               $W.t = s_i \xrightarrow{\{?,!\}a} s_i'$ ;
             Add $s_i'$ to set $D$. ;
          End If
          $W.s_i = (l_i, v = v + g)$ ;   /* Add sample unit to state $W.s_i$
                                     for next Iteration */
        End While
      End For
```

```
        $S$ = $D$ ;
        $D$ = $\Phi$ ;
      End While
```

Notice that the main difference of sampling in M1, M2, and M3 is the number of clocks. In M1, the number of clocks is the sum of clocks in the IUT and its context. In M2, it is the sum of clocks in the test purpose and the specification. In M3, it is the number of clocks in the test purpose or the specification. Therefore, M3 obviously uses the least number of clocks and therefore generates fewer states and fewer test cases.

4.7 Test Cases Generation

This operation is the last step and is common to all the methods M1, M2, and M3. The objective is to derive test cases from the global GA resulting from the previous steps. To this end, we use all pass-verdict paths selection criterion to reach every leaf in GA, which satisfies both the test purposes and the specification. In other words, the all-pass-verdict paths selection criterion consists of generating every path that leads to a verdict Pass. To do so, we start at the initial state of GA and apply a depth-first traversal to reach each leaf with a pass verdict. During the traversal, we pick up, in an order, each input output (including delay) to compose test cases.

The algorithm walks through the transitions tree with depth-first traversal. The algorithm is described as follows:

```
Input: Grid TIOA $W$

Output: Test cases string

Step0: Initialization
      $tc$ = $\varepsilon$ ; /* string for test cases */
      $TCS$ = $\Phi$ ; /*a set of test cases string */
      $s$ = $s_0$ ; /* the reached state, $s_0$ is initial state of
```

```
                 grid TIOA W */

Step1: Test cases generation
    While state s is not a leaf, do
       $tc = tc \bullet e$ ;   /* e is an event which cause transition
                                 $s \xrightarrow{\{?,!\}e} s'$  */
       $s = s'$ ;
    End While
    If the path of tc is verdicted "Pass"
       $TCS = TCS \cup \{tc\}$ ;
    End If

Step2: Repeat step0 and step1 till all transitions are covered.
```

4.8 Conclusion

In this chapter, we introduced a new methodology that is developed to generate test cases for embedded real-time systems with test purposes and the specification. We discussed each operation of the methodology in detail and provided the algorithms for each of them. Moreover, we noticed that the number of clocks has a great influence on the number of states generated during sampling operation. The earlier sampling is done the less is the number of states generated.

In the next chapter, we will implement the new methodology introduced in this chapter with object oriented programming technology. We will discuss the design of the implementation through the package diagrams, class diagrams and activity diagrams.

Chapter 5
Implementation of Algorithms

In this chapter we will introduce the implementation of our methodology. The implementation is developed with OOP methodology. It is coded in C++ under SUN Solaris system (Generic_117000-01) and compiled with g++ 3.3.2. Lex and Yacc are used to design our lexical and syntax parsers for MSC and the specification files. We will discuss the design of the implementation through the package diagrams, class diagrams and activity diagrams.

5.1 Parsing the Test Purpose and the Specification files

In our approach, the specification and test purposes are described in two separate text files. Those files are parsed using the well-known UNIX tools Lex and Yacc. Lex is used to analyze the grammars of MSC and TIOA and return the different tokens in the files. Examples of such tokens are clock names, transitions, input and output actions, and clock guards. However, Yacc is used to verify the syntax of the grammars of MSC and TIOA and perform the actions associated with each grammar line. In our case, the actions consist of converting the contents of MSC and TIOA files into memory data structures corresponding to the TIOA objects of our design. In fact, the results of this operation are two collections of TIOA objects, one for the specification, and the other for the test purpose.

5.1.1 Lex

Lex is a well-known UNIX tool, which is used to generate a lexical analyzer for text file parsing. The lexical analyzer reads a source file and returns tokens to the syntactical analyzer, which is generated by Yacc. Tokens are terminal symbols of concrete textual syntax and a sequence of tokens composes the textual source, like the test purpose file and the specification file. Tokens can be classified into different entities which are usually described with regular expressions. In Lex, tokens are defined in a regular expression table and each item in the table corresponds to a fragment of C codes. These codes are executed when the corresponding tokens are recognized from inputs, and then, the recognized tokens are returned to the syntactical analyzer. An example of a regular expression table in Lex is shown in Figure 5-1.

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "MyLib.h"
    #include "y.tab.h"

%}

LETTER          [A-Za-z]

DECIMAL_DIGIT       [0-9]

OTHER_CHARACTER        ["?"  "%"  "+"  "\-"  "!"  "/"  ">"  "*"  "\\"  "<"  "="]

SPECIAL       "("  ")"  ","  ":"

%%

";"    {printf(";\n");}
[aA][cC][tT][iI][oO][nN]       {printf("ACTION\n");}
[aA][fF][tT][eE][rR]     {printf("AFTER\n");}
[wW][iI][lL][dD][cC][aA][rR][dD][sS]        {printf("WILDCARDS\n");}

[0-9]+           {printf("NUMBER\n");}

[A-Za-z_]+[A-Za-z0-9"_""."]*   {printf("NAME\n");}

[A-Za-z0-9]+([A-Za-z0-9]|{OTHER_CHARACTER})*     {printf("WORDS\n");}

"/*".*"*/"    {;}
```

Figure 5-1 An example of regular expression table in Lex

Texts in our test purpose file will follow the concrete textual syntax defined in[16], where terminal symbols are defined by Lexical rules. In our implementation, we define our Lex regular expression table following these rules, especially the definition of keywords. The regular expression table of our implementation is provided in Appendix A.

5.1.2 Yacc

Yacc is a UNIX tool which is used to generate syntactic analyzer for a certain language. The syntax is the structure of strings in a language and is described by a grammar. Users specify the syntactical structures of a language in a yacc file together with fragments of C codes, which are invoked when the corresponding structures are matched by a sequence of tokens returned from the lexical analyzer. Our yacc file for the test purpose is designed following the concrete textual grammar defined in [16]. In our thesis, we only concentrated on message input/output events with time constraints in our yacc file and left the rest for further research.

For the specification file, there is no standard concrete textual grammar for TIOA description. So we designed our own grammar following BNF to meet our need. Obviously, we can change it to meet other grammars if necessary without any influence on our results. Our textual grammar designed for TIOA specifications is given in Appendix B.

5.2 Class Diagram

To show our implementation structure clearly, we organized the classes of our implementation into nine packages: *Specification Compiler*, *MSC Compiler*, *Msc*, *Selection*, *Sampling*, *Synchronous Product*, *Partial Product*, *Test Suite* and *TIOA*. Their dependency relations are shown in Figure 5-2. Each package includes one or several classes. In the following sub-sections, we pick up some important packages and introduce them in detail.

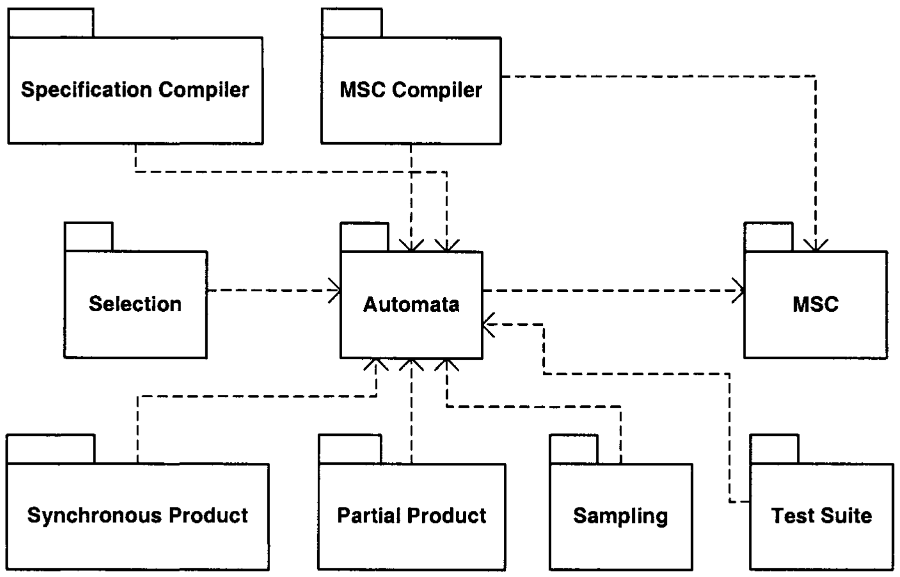


Figure 5-2 Package Diagram of our implementation

5.2.1 The MSC Package

During MSC syntax parsing, we converted the content of MSC file into objects. The conversion is realized in fragments of codes which are related to the syntax structures in yacc files.

We designed objects in MSC package according to the structure of MSC in [16]. They are: MSC document object, MSC chart object, instance object, event object, message object, clock object, and constraint object. Clock object and constraint object were designed for the time feature of MSC2000. To make things simple, we only considered, in our design, the simplest message sequence charts in which there are only message input events and message output events with time constraints information. We left the complicated structure, data, and HMSC parts to further research. The relationship among these objects is shown in Figure 5-3.

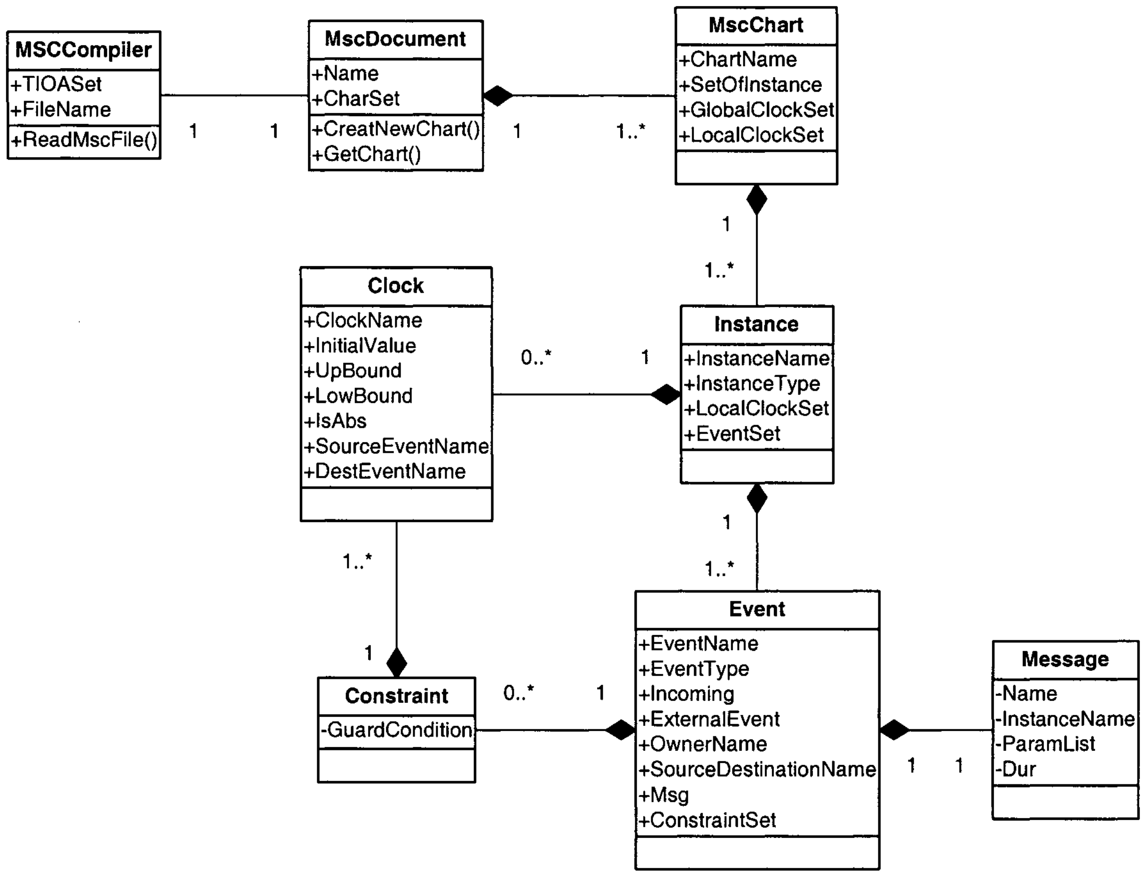


Figure 5-3 Class diagram in MSC

- MSC document class -- *MscDocument.*

In MSC, the Message Sequence Chart document defines an instance kind and the associated collection of message sequence charts, each of which again defines a set of traces [16]. It is the entity that gathers the related MSC charts. We designed class *MscDocument* to simulate the function of Message Sequence Chart document.

A *MscDocument* object is created when the parser recognizes msc document head pattern "**mscdocument**<instance kind>[**related to**<sdl reference>] [<inheritance>] <end>". We used this object as a collection to save MSC chart objects.

The main attributes are described below:

Attributes:

- *ChartSet.*
  A set used to save MSC chart objects.

- *Name.*
  Takes the value of <instance kind> in the pattern.

• MSC chart class – *MscChart*

A message sequence chart is the entity in which several instances are interactive by sending and receiving messages. It is the part that describes systems' behaviors. Therefore, we designed class *MscChart* to keep it's content.

A *MscChart* object is created when the parser recognizes <msc head> pattern "**msc** <msc name>[<msc parameter decl>][time offset]<end>".

Attributes:

- *ChartName.*
  Takes the value of <msc name> in the pattern

- *SetOfInstance*
  A set used to collect Instance objects.

- *GlobalClockSet*

It is the set where we collect global clocks. We define the global clock for absolute timing, thus there should be only one global clock for the whole system.

- *LocalClockSet*

  It is the set where we collect a kind of clock which measures relative timing between two events on different instances. We will not include these clocks when we conduct translation because there is no definition for this kind of clock in TIOA. This clock set is designed for future use.

• Instance class – *Instance*

We will create instance objects when the parser recognizes the pattern "<instance name>:: **instance**<instance kind>[<decomposition>]<end>".

Attributes:

- *InstanceName*

  Takes the value of <instance name> in the pattern.

- *InstanceType*

  Takes the value of <instance kind> in the pattern.

- *LocalClockSet*

  Collects clocks which are used to measure relative timing between two events in the instance.

- *EventSet*

  Collects events which are on the instance.

• Event class – *Event*

We will create event objects when the parser recognizes the pattern of events. In MSC, events can be classified into two types: orderable events and non-orderable events. Orderable events include message events, incomplete message events, method call events, incomplete method call events, timers and local actions. Non-orderable events include start method, end method, start suspension, end suspension, start coregion, end coregion, shared condition, shared msc reference, share inline expression, instance head statement, instance end statement and stop. At present, we only considered message events, timers, instance head statement and instance end statement in our implementation and left other events for future improvement.

The pattern for the events mentioned above with timing is "[label<event name><end>]{**out**<msg identification>**to**<input address> | **in**<msg identification>**from**<output address> | **starttimer** <timer name>[,<timer instance name>][<duration>][(<parameter list>)] | **stoptimer** <timer name>[,<timer instance name>] | **timeout** <timer name>[,<timer instance name>][(<parameter list>)]}[**before**<order dest list>][**after**<order dest list>]<end>[**time**<time dest list><end>]".

Attributes

- *EventName*

  Takes the value of <event name> in the pattern. It is used to differentiate events.

- *EventType*

It defines the type of events, including MESSAGE, STARTTIMER, STOPTIMER, and TIMEOUT.

- *Incoming*

  If 1, it means that the event is an incoming event. If 0, it means the event is an outgoing event.

- *ExternalEvent*

  If 1, it means the event is sending/receiving a message to/from the environment. If 0, it means the event is sending/receiving a message to/from another instance.

- *OwnerName*

  The name of the instance that owns this event.

- *Msg*

  The message that is corresponding to the event.

- *SourceDestinationName*

  The name of the instance to which the message is sent or from which the message is received. It is used when the event is an internal event.

- *ConstraintSet*

  The collection of *Constraint* objects that are used as guard conditions for this event.

• Message class – *Message*

A message is a relation between an output event and an input event [16]. It is created for each <msg identification> in message event pattern. <msg

identification> is defined as <msg identification>::<message name>[,<message instance name>][(<parameter list>)].

Attributes

- *Name*

  The message name. It takes the value of <message name> in the pattern.

- *InstanceName*

  It takes the value of <message instance name> in the pattern. It is employed for unique mapping in message exchanging.

- *ParamList*

  It takes the value of <parameter list> in the pattern.

- *Dur*

  It is used for a timer. It takes value of <duration> in the starttimer event pattern.

• Clock class – *Clock*

Clocks are used for timing between events. Time concepts are introduced into MSC to support the notion of quantified time for the description of real-time systems with a precise meaning of the sequence of events in time. MSC events are instantaneous. Time constraints can be specified in order to define the time at which events may occur [16].

A clock object is created when the parser recognizes the pattern **time**<time dest list><end>, where:

<time dest list>::=<time interval>[<time dest>][,<time dest list>]

<time interval>::=<interval label> <singular time> | <interval label> <bounded time>

Attributes

- *ClockName*

  ClockName is used to differentiate clocks. There are two types of timing in MSC. One is absolute timing; the other is relative timing. Absolute timing is used to define the time point at which an event happens. Relative timing uses pairs of events – preceding and subsequent events, where the preceding event enables the subsequent event [16]. Absolute timing is described by the global clock. There should be only one global clock in the system and we named it GLOBALCLOCK. For relative timing, we created as many clocks as we needed and used the value of <interval label> in the pattern as the clock name.

- *InitialValue*

  The initial time value for a clock. "0" if it cannot be decided.

- *Upbound*

  Upbound is the upper time point in the pattern <bounded time>.

- *Lowbound*

  Lowbound is the lower time point in the pattern <bounded time>.

With the help of *Lowbound* and *Upbound*, we can define a time interval to constrain the occurrence of a pair of events. We can also define the time point at which an event must occur by setting the values of *Lowbound* and *Upbound* to the same value.

- *IsAbs*

  True if there is an absolute time mark "@" in <time interval>; otherwise, false.

- *SourceEventName*

  It is the name of the preceding event in relative timing.

- *DestEventName*

  It is the name of the subsequent event in relative timing.

• *Constraint*

Class *Constraint* was designed to describe the time constraints of an event. We attached constraints to an event by saving these constraints in *ConstraintSet* of the event. It means that the event can occur legally only when the constraints are satisfied. For relative timing, there are two events involved: one is the preceding event; the other is the subsequent event. In this case, *Constraint* objects are attached to the subsequent event and the corresponding clocks are initialized at the preceding event. For absolute timing, there is only one event involved and *Constraint* is attached to that event.

Attribute

- *GuardCondition*

  GuardCondition is the collection of clock objects that define the minimal or maximal bounds in the time domain. These clocks have "AND" logical relations among them. With "OR" logical relation among the constraints in the constraint set of an event, we can describe any logical expressions for an event.

5.2.2 The Automata Package

There are four sub-packages in the *Automata* package. Each of them includes a series of entity classes, which are designed for the implementation of our algorithms. They are *TIOA* sub-package, *Grid TIOA* sub-package, *Global TIOA* sub-package, and *Grid Global TIOA* sub-package. The package diagram is shown in Figure 5-4.

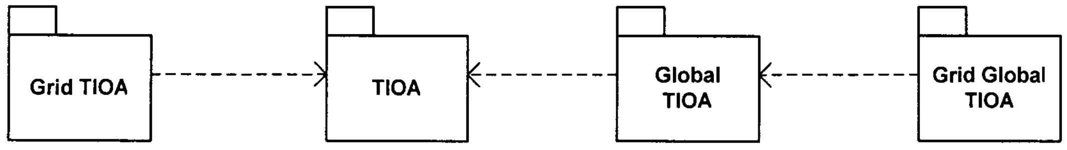


Figure 5-4 Package diagram in Automata

5.2.2.1 Sub-package TIOA

In this part, we introduce classes in the sub-package TIOA. We designed these classes according to the definition of TIOA in section 3.2.1. They are *Automachine*, *InputOutput*, *Location*, *Transition* and *Clock.* We first generated the objects of these classes when we parsed the specification file. The relationship among these objects is shown in Figure 5-5.

- *Automachine*

  The class *Automachine* was designed based on the definition of TIOA.

  Attributes

  - *Name*

    It is assigned the value of <tioa ID> when the parser recognizes the pattern "System<tioa ID><end>". We discriminate TIOAs by assigning a name to each TIOA. For the TIOAs that have correspondent instances in the test purpose, their names should be the same as the instance kinds of the correspondent instances.

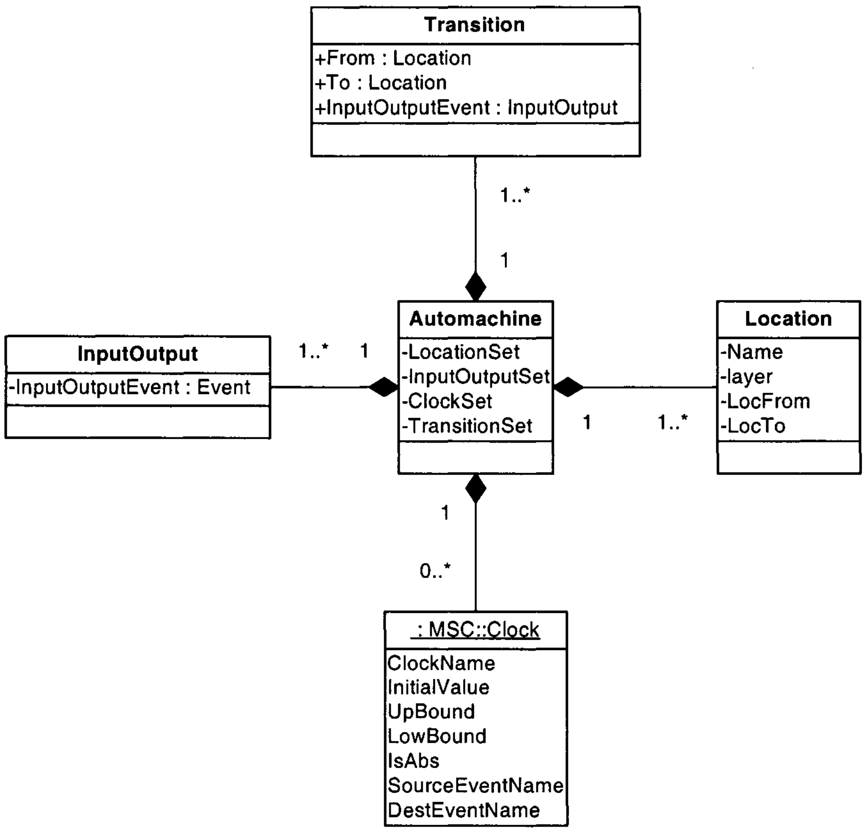


Figure 5-5 Class diagram in TIOA

- *LocationSet*

  It is the collection of the TIOA locations. The first element in *LocationSet* is the initial location of the TIOA.

- *InputOutputSet*

  It is the collection of inputs/outputs of the TIOA.

- *ClockSet*

  It is the collection of clocks. These clocks are used as guard conditions in transitions.

  - *TransitionSet*

    It is the collection of transitions.

- *InputOutput*

InputOutput was designed to describe the input and output actions of TIOA.

Attribute

  - *InputOutputEvent*

    It is an *Event* object created when the parser recognizes the pattern of "<input item>" or "<output item>". Class *Event* is introduced in the section 5.2.1. We assign the *Name* of *Msg* with the value of <message ID> in the pattern. For the pattern "<input item>::=<message ID> from <tioa ID> | ENV", we set the attribute *Incoming* to true. We set the attribute *ExternalEvent* to true if ENV is recognized; otherwise, we set it to false. For the pattern "<output item>::=<message ID>to<tioa ID>|ENV", we set attribute *Incoming* to false. We set the attribute *ExternalEvent* to true if ENV is recognized; otherwise, we set it to false.

- *Location*

Location was designed to describe the location of TIOA. It will be created when the parser recognizes the pattern "Locations::<location ID> [,<location ID>]<end>" and is saved in the *LocationSet* collection.

Attributes

  - *Name*

It takes the value of symbol "<location ID>". We assigned each location a unique name to discriminate locations of a TIOA.

- *Layer*

  It is used for the location tree of a TIOA. It is a measurement of the distance from a location to the initial location.

- *LocFrom*

  It is the collection of the pointers to the locations that can be transferred to this location under some actions happening.

- *LocTo*

  It is the collection of the pointers to the locations which can be reached from this location under some actions happening.

- *Transition*

Transition was designed according to the transitions in a TIOA. It is created when the parser recognizes the pattern of "Transitions:<transition item>+" and saved in the *TranstionSet* collection of a *Automachine* object.

- *From*

  It is the pointer to the source location of the transition. It points to the location whose name is the same as the value of symbol "<source location ID>" in "<transition item>".

- *To*

  It is the pointer to the destination location of the transition. It points to the location whose name is the same as the value of symbol "<destination location ID>" in "<transition item>".

- *InputOutputEvent*

  It is one of the input/output events in *InputOutputSet* of the TIOA according to the value of the symbol "<message ID>". Constraints will be created and added to the collection *ConstraintSet* of the event when the parser recognizes the symbol "<logical expression>", which is defined as "<relational expression>[<logical operators><relational expression>]*". One constraint is correspondent to one "<relational expression>". At last, reset clocks will be added to *ClockValueSet* of the event when the parser recognizes the symbol "<clock reset>". It means that these clocks need to be reset to zero after the transition is success. The Class *Clock* is defined in section 5.2.1.

- *Clock*

  We create a *Clock* object when the parser recognizes the pattern "Clock:<clock item>*" and save it in *ClockSet* of *Automachine*. The class *Clock* is defined in section 5.2.1.

5.2.2.2 Sub-package Global TIOA

In this sub-package, we include entity classes designed for the procedure of partial product. They are *Global_TIOA*, *Global_Transition*, *Global_Clock*, *Global_Locaion*, *Global_Loc*, and *FIFO*. Similar to class *Automachine*, the attributes of the class *Global_TIOA* are also composed of *LocationSet*, *TransitionSet*, *ClockSet* and *InputOutputSet*. However, here we use objects of *Global_Location*, *Global_Transition*, and *Global_Clock* as the elements of these sets.

A *Globle_Location* object is a location collection, which represents the combination of the locations from the embedded components. An element in *Globle_Location* is an object of class *Global_Loc*, which represents a location of a component TIOA. *FIFO* objects associate with *Global_Loc* objects to express the status of communication channels.

Class *Global_Transition* was developed from class *Transition* in package *TIOA*, the only different part is that, in *Global_Transition*, the type of *From* and *To* is *Global_Location.*

Global_Clock is the same as *Clock* in package *TIOA* except that we added an attribute *AutomachineName* to *Global_Clock* to show the owner of this clock.

Figure 5-6 shows the class diagram of this package.

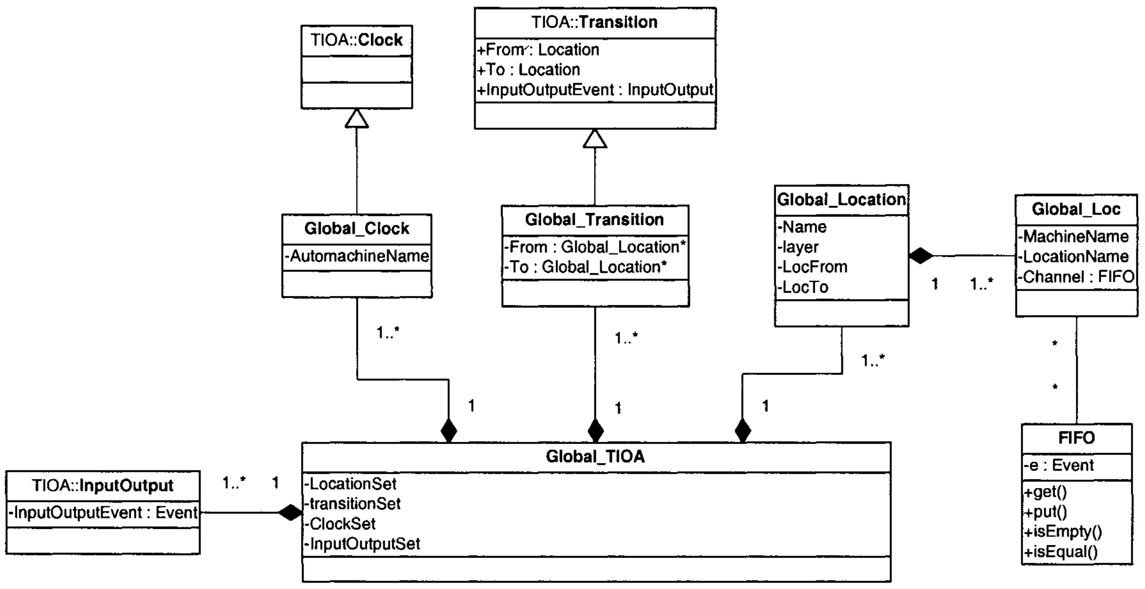


Figure 5-6 Class diagram in Global TIOA

5.2.2.3 Sub-package Grid TIOA & sub-package Grid Global TIOA

The entity classes in sub-package *Grid TIOA* and sub-package *Grid Global TIOA* were designed for the procedure of sampling. They are similar to their parents except that we added time information in the location classes. Figure 5-7 and Figure 5-8 shows the class diagram of this package.

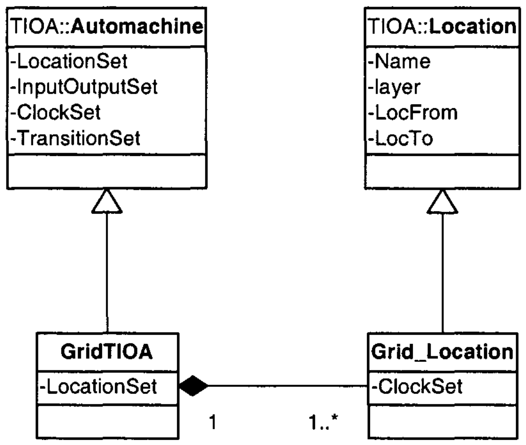


Figure 5-7 Class diagram in *Grid TIOA*

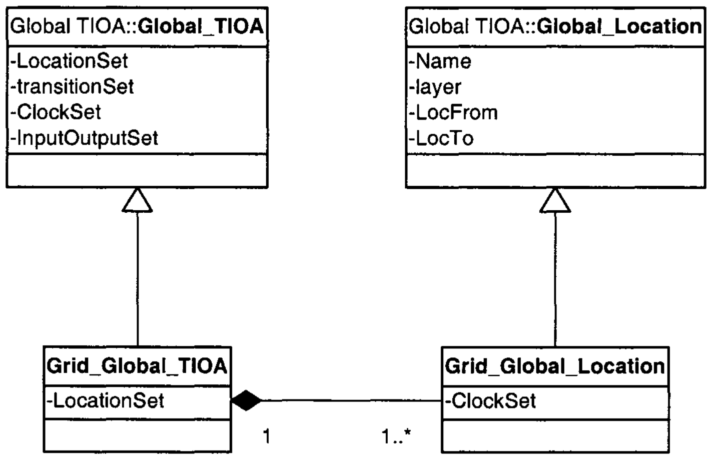


Figure 5-8 Class diagram in *Grid Global TIOA*

5.2.3 The Selection Package

We designed two sub-packages of classes in this package to implement the two steps of the selection procedure. The Sub-package "*Paths Selection with TP*" includes classes designed for transition path selection based on the test purpose. It is used in the step one of the selection process. The sub-package "*Postamble Selection*" includes classes designed to find the postambles with an external output for a verdict. Figure 5-9 shows the package diagram.

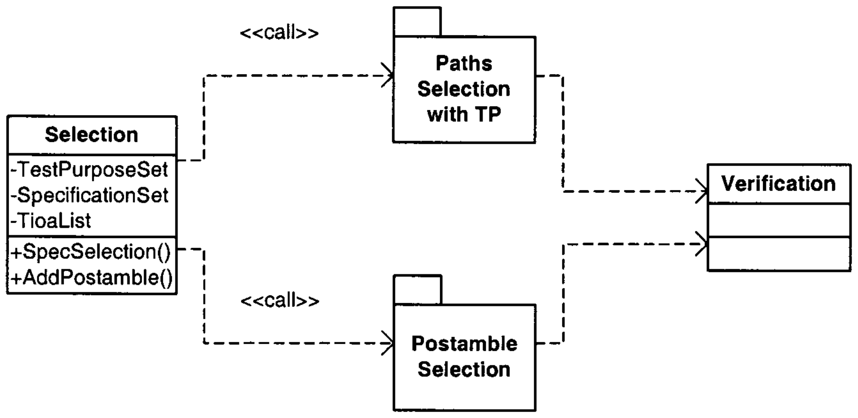


Figure 5-9 Package diagram for Selection

In this diagram, there are two classes, which have dependency relationship with the two packages. Class "*Selection*" was designed as the interface and the controller class. It decides the execution of other classes in *Selection* package and returns the result. Class "*Verification*" was designed to check the consistency of the selected path. It is used during paths selection and postamble selection operations.

5.2.3.1 Sub-package Paths Selection with TP

The basic idea of the algorithm in our first step of the selection process is that, for each TIOA component in the specification, we walk through the transitions tree to find the path which covers the test purpose of the component and also a context of the IUT at the same time. The search starts from the initial location of each TIOA in the specification, with depth-first traversal, until we find the path that satisfies our requirements. We designed a series of objects to control our searching procedure. These objects assure that the selected paths cover the test purpose while keeping synchronization among them. The relationship among these objects is shown in Figure 5-10 and the explanation of these objects is given below:

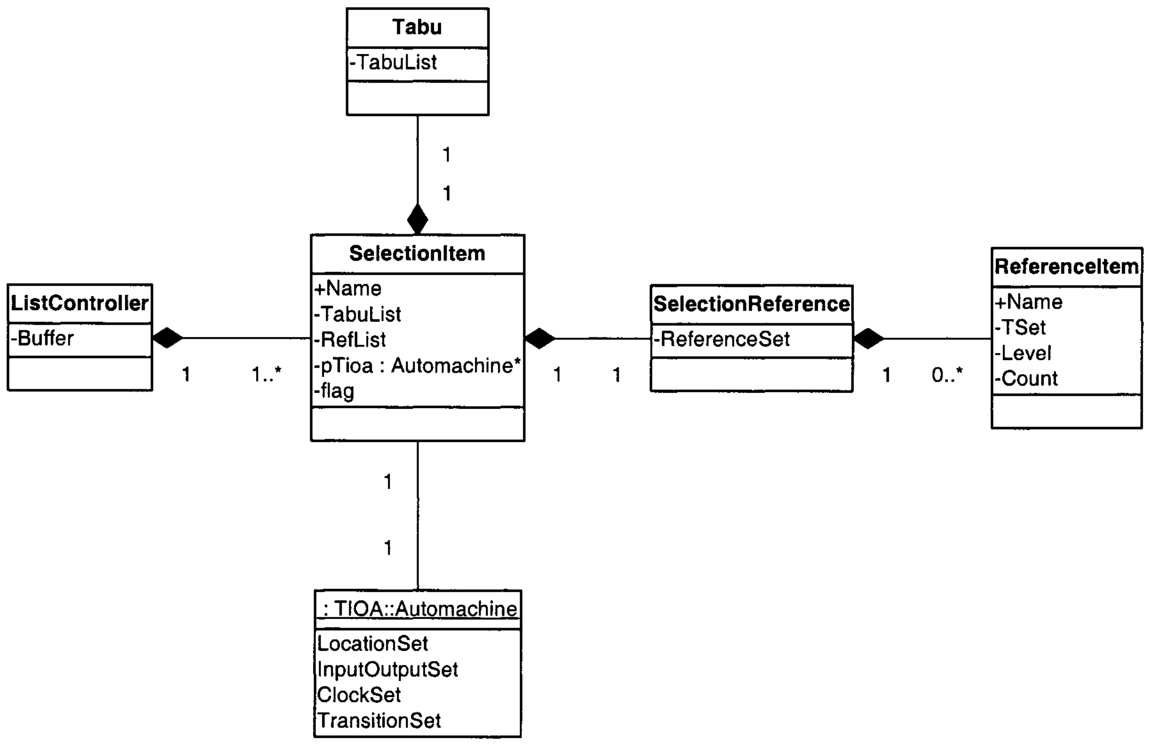


Figure 5-10 Objects relationship in package "Paths Selection With TP"

- *ReferenceItem*

This class was designed to constrain the path selection. It preserves a set of ordered transitions that the selected path must cover in the same order. In fact, these transitions represent a requirement from other TIOA components or from the test purpose. The selected path must satisfy this requirement to ensure that it is in accordance with other selected paths in other components and with the test purpose if any. We want to mention that these transitions may be not consecutive in the specification, thus, they may be not consecutive in the selected path.

Main attributes

- *Name*

  It is the name of a TIOA component in the specification. It means that the set of ordered transitions is a requirement from the TIOA components with the name.

- *Tset*

  A collection object where we save the transitions as a requirement from other components or from the test purpose.

- *Level*

  It is used to dynamically record the progress of covering. If the value of *Level* is equal to, or greater than, the value of *Count*, it means that all transitions in *TSet* are covered in the selected path.

- *Count*

  It is the number of transitions in *TSet.*

- *SelectionReference*

It was designed to save all *ReferenceItem*s for the path selection for a TIOA. These *ReferenceItems* record the requirements from other TIOA components or from the test purpose. The selected path of the TIOA must satisfy all of these requirements.

Main attribute:

- *ReferenceSet*

  It is a collection that is used to save *ReferenceItem.*

- *Tabu*

It was designed to save the paths, which are forbidden during the selection process. When we did searching in the transitions tree of the TIOA, we avoided selecting these paths. We borrowed this name from Tabu search.

Main attribute:

- *TabuList*

  It is a collection which is used to save the paths that are forbidden during the selection process.

- *SelectionItem*

This class was designed to associate a *SelectionReference* object and a *Tabu* object with a TIOA component in the specification. When we select a path from this TIOA, the selected path must satisfy both *SelectionReference* and *Tabu.*

Main attributes:

- *Name*

  It is the name of the TIOA that is associated with this *SelectionItem.*

- *TabuList*

  It is a Tabu object in *SelectionItem.* It defines the paths which are forbidden in path selection.

- *RefList*

It is a *SelectionReference* object in *SelectionItem.* It includes the *ReferenceItems* from other TIOAs and the test purpose.

- *pTioa*

  It is a pointer to the new TIOA that is extracted from the associated TIOA. The transitions of the new TIOA are from the selected path.

- *Flag*

  It is an indicator to show whether this *SelectionItem* object needs to be updated. The update is needed when the content of *TabuList* or *RefList* is changed.

- *ListController*

  This class was designed to collect *SelectionItems* during selection operation. Each *SelectionItem* is associated with a TIOA component in the specification. The path selection in each TIOA must consider the requirement from other TIOAs and from the test purpose. As a result, the TIOAs in *ListController* compose a relatively independent system model whose behavior follows a scenario described in the test purpose.

5.2.3.2 Sub-package Postamble Selection

To find an external output which can be observed by the tester for a verdict, we configured a global searching tree which was generated from the combination of the transitions trees in the related TIOAs. The root of this global tree is composed of the final states in the result TIOAs of the first step and the initial states in the TIOAs other than the result TIOAs in the first step. The combination procedure is similar to the procedure of partial product construction. We walk through the global tree with breadth-first traversal until we meet the first external output, which satisfies the two requirements mentioned before, or until the searching depth reaches a certain threshold. Three main objects were designed for this procedure. They are *Node*, *NodeSet*, and *Postamble.* The relationship among them is shown in Figure 5-11 and the explanation of these objects is given below:

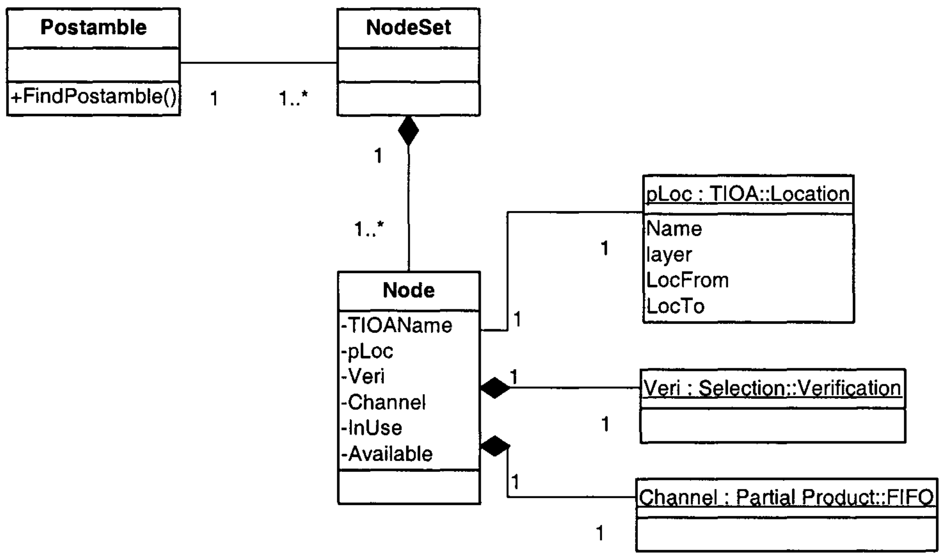


Figure 5-11 Objects diagram of Postamble Selection package

- *Node*

One *Node* object is associated with one location of the related TIOA. It contains an identification of the location, a *Verification* object, a *Channel* object and a Boolean type variable *Available.* The *Verification* object has two functions, one is to check the executability of the transition starting from this location; the other is to keep the information of the trace that starts from the initial location and ends at this location. The *Channel* object is used to show the current communication status. The Boolean variable *Available is* very important in our searching algorithm: if it is true, it means that the transition starting from the location of this node will happen after the last transition of the IUT in order.

- *NodeSet*

  One *NodeSet* object is associated with a global location, which is composed of the current locations from all related TIOAs at a certain step. It is a collection of *Nodes* corresponding to those locations.

- *Postamble*

  It is the controller object which realizes the postamble finding algorithm. It includes four methods, *FindPostamble*, *IsExternalOutputExist*, *CreateNewNodeSet*, and *CanStop*. *FindPostamble* implements the main function of the algorithm. *IsExternalOutputExist* checks the global searching tree to see whether it has an available external output. *CreateNewNodeSet* creates the next *NodeSet* during the global searching tree composition. *CanStop* decides whether the searching should be stopped.

5.2.4 The Synchronous Product & Partial Product & Sampling Packages

There are two classes in the *Synchronous Product* package, one is *SynchronizeProduct*, and the other is *SynchronizeProductWithGrid*. *SynchronizeProduct* is used when we construct a synchronous product between two TIOAs, while *SynchronizeProductWithGrid* is used when we construct a synchronous product between two grid automata. These classes were designed to implement the two algorithms, which are introduced in section 4.4.1 and section 4.4.2. Figure 5-12 shows the class diagram of these two classes.

As for the *synchronous product* package, two classes were designed for the partial product construction between two TIOA or between two grid automata. Figure 5-13 shows the class diagram of these two classes.

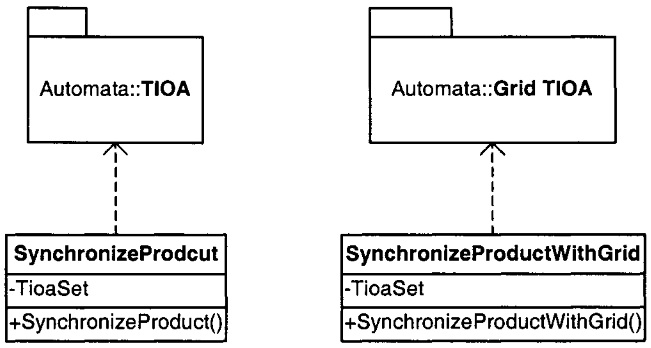


Figure 5-12 Class diagram of *Synchronous Product* package

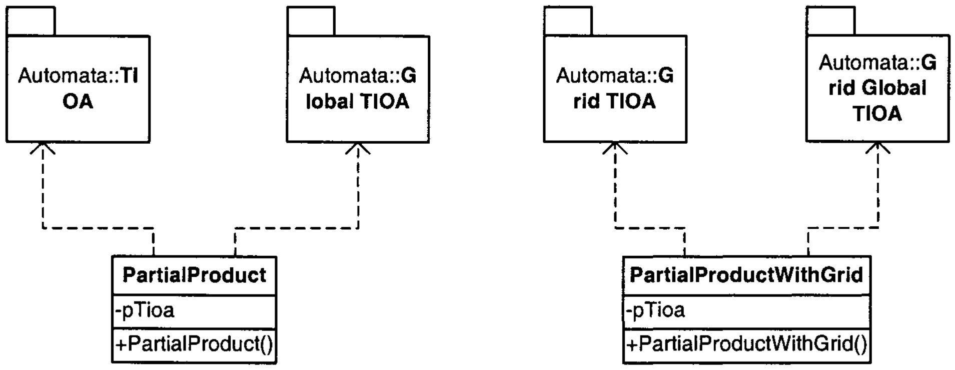


Figure 5-13 Class diagram in *Partial Product* package

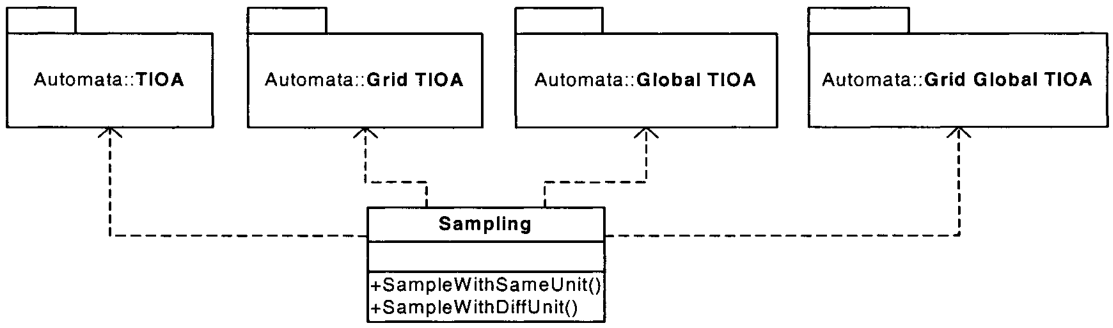


Figure 5-14 Class diagram in *Sampling* package

We implemented two sampling algorithms in our implementation. One is sampling a TIOA with the same granularity; the other is sampling a TIOA with different granularities. Both sampling algorithms are introduced in section 4.6.We found that sampling with different granularities can generate fewer states for a grid automaton model (see later chapter 6). Figure 5-14 shows the class diagram of these two classes

5.3 Activities diagrams

Our implementation can be divided into six main operations: The parsing of the specification and test purposes; the selection of the transitions to be considered for test cases generation; the construction of the synchronous product of the test purpose and the specification; the construction of a partial product for the system under test; the sampling of TIOAs; and the generation of test cases.

These operations can be executed in different orders. Each order gives rise to a new method for test cases generation for embedded real-time systems. We implemented three of these methods. The difference between these methods is the position of sampling in the whole process. Figure 5-15 shows the activity diagram of each of these three methods.

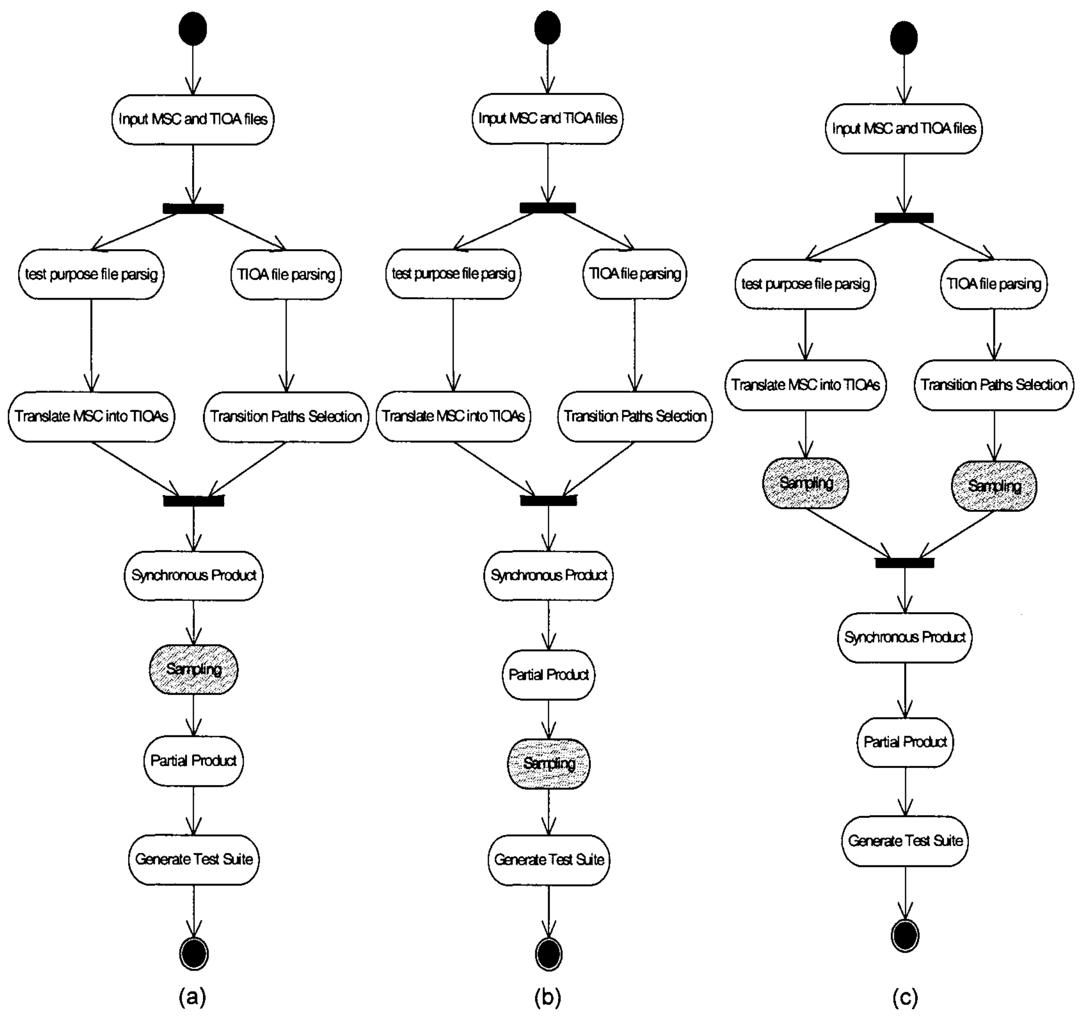


Figure 5-15 Activity diagram for (a) method 1, (b) method 2, and (c) method 3.

5.4 Conclusion

In this chapter, we introduced classes designed for the implementation of our methodology. We showed the relationship among these classes with package and class diagrams. Finaly, we used activity diagrams to describe the processing of our implementation.

In the next chapter, we will give an example to show how our implementation works. We will analyze the results and draw a conclusion.

Chapter 6
Case Study

In this chapter, we will give an example of test cases generation with our three methodologies Ml, M2, M3. We will compare them in terms of the number of states generated and the number of test cases derived. We will also apply time sampling with the same granularity and different granularities and compare the results obtained.

6.1 Railroad Crossing System

In this section, we use the well-known Railroad Crossing System (RCS) [17] as our example. There are three components in RCS, *Train*, *Controller*, and *Gate*. These three components are interacting with each other as follows:

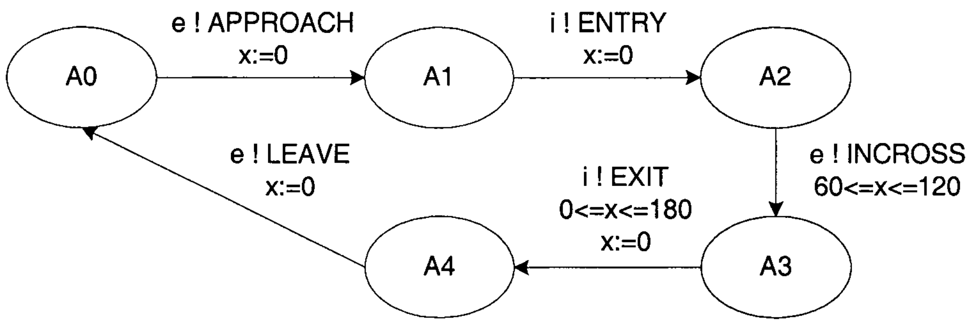
Train: As the trigger of the whole system, the *Train* has three external signals which can be observed and two internal signals which make the *Controller* control the *Gate*'s movement (Up or Down). The *Train* sends an external signal *APPROACH* to the environment to trigger the whole system working. After sending the *APPROACH* signal, the *Train* sends an internal signal *ENTRY* to the *Controller* through the entry sensor *S1*, informing that it comes. Then within [60s,120s], the *Train* should send a *INCROSS* signal to the environment to indicate that it is in the cross region. At the time, the *Gate* should be closed to ensure that no other trains could be in the same region. The *Train* triggers the exit sensor *S2* to send an internal signal *EXIT* to the *Controller* within [0,180s] after it sends the *ENTRY* signal when it leaves the cross. At the same time, it sends an external signal *LEAVE* to the environment to inform that it is out of the cross region.

Controller: When receiving the internal signal *ENTRY* from the *Train*, the *Controller* sends an internal signal *LOWER* to the *Gate* in [0,60s] to close the *Gate*. Then, after receiving the internal signal *EXIT*, it sends an internal signal *RAISE* to the *Gate* in [0,40] to open it.

Gate: The initial state of the *Gate* is open. When it receives *LOWER* signal from the *Controller*, it starts to move down. This action should be finished within [0,60s] after the *Gate* receives the internal signal *LOWER*. After that, it sends an external signal *DOWN* to the environment to show that the *Gate* is closed and no other trains can be in the cross region. The *Gate* starts to open after receiving the internal signal *RAISE* from the *Controller* and it will take the *Gate* [60s, 180s] to be in open state. The *Gate* sends external signal *UP* after it opens.

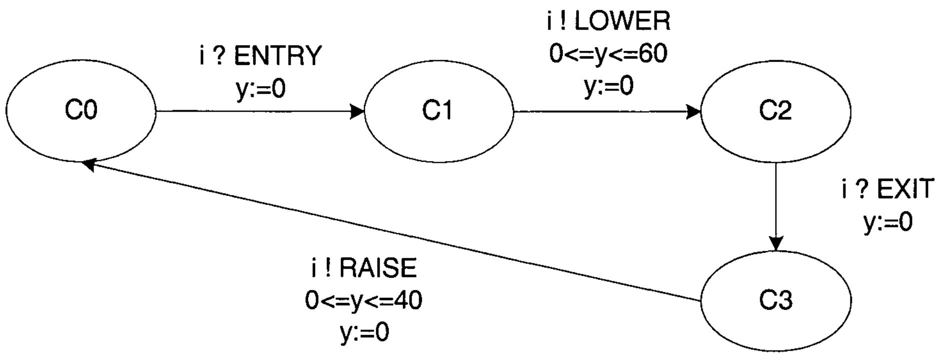
Our time constraint assumption follows the safety property and utility property. That is, the *Gate* keeps closed during the cross is occupied and the *Gate* is open when there is no *Train* in the cross region. Figure 6-1, 6-2, 6-3 shows the specification of Railroad Crossing System.

In this example, we suppose that we want to test the conformance of the component *Controller*. We want to know whether the *Controller* can send *RAISE* signal to the *Gate* within 30s after it receives *EXIT* signal from the *Train*. This requirement conforms to the specification, which allows the *Controller* to give a reaction to the signal *EXIT* within 40s. The test purpose that describes the property to be checked is shown in Figure 6-4. Here, the *Controller* is the implementation under test. The *Train* and the *Gate* are its context.



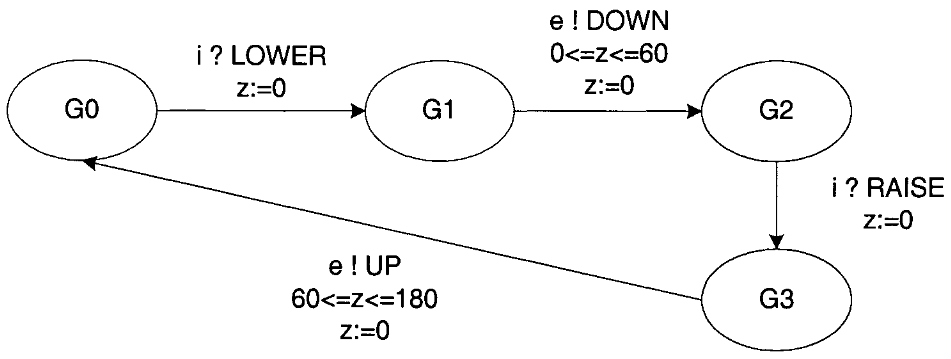
x -- clock , e -- means external , i -- means internal , ? -- means input , ! -- means output

Figure 6-1 The specification of the *Train* described by TIOA



y -- clock , e -- means external , i -- means internal , ? -- means input , ! -- means output

Figure 6-2 The specification of the *Controller* described by TIOA



z -- clock , e -- means external , i -- means internal , ? -- means input , ! -- means output

Figure 6-3 The specification of the *Gate* described by TIOA

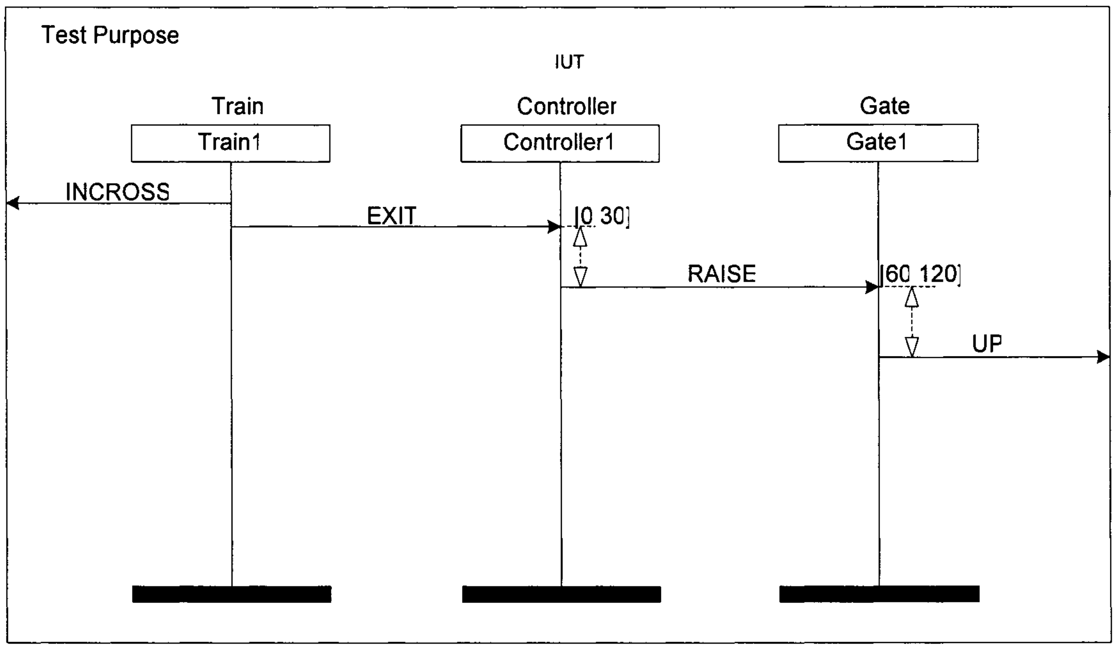


Figure 6-4 Test Purpose for the *Train* going out of the cross region

Following the steps of our methodologies, first we need to translate the test purpose into automata. We generate three TIOA objects: One for the instance *Train*, one for the instance *Controller*, and another one for the instance *Gate*. For each of these instances, we first use Lex to extract the events names, the events types, the events sources, and the events destinations from the MSC text file by looking for MSC keywords *IN*, *OUT*, *TO*, and *FROM*. Then, we generate the locations and the transitions under each event in Yacc. We create the initial location at the beginning of each instance, and add a new location after the happening of each event. Moreover, clocks names and clock guards are found according to the keyword *TIME*. The results of the translation are shown in Figure 6-5.

Then, we start our selection procedure. As we discussed in 4.3, we need to generate a sub-specification model, a relatively independent system model whose behavior covers the scenario described in the test purpose with postambles. We show the result procedure in Figure 6-6. In this result, the transition path of the instance Controller in Figure 6-5 is $tc0 \xrightarrow{?EXIT} tc1 \xrightarrow{!RAISE} tc2$, and the corresponding path of the specification is $C0 \xrightarrow{?ENTRY} C1 \xrightarrow{!LOWER} C2 \xrightarrow{?EXIT} C3 \xrightarrow{!RAISE} C0$. Because *ENTRY*, *LOWER*, *EXIT* and *RAISE* are internal events in RCS, the context selection step should find their counterparts in the context of the IUT. Thus, the path $A0 \xrightarrow{?APPROACH} A1 \xrightarrow{!ENTRY} A2 \xrightarrow{!INCROSS} A3 \xrightarrow{!EXIT} A4 \xrightarrow{!LEAVE} A0$ should be selected from the specification of the *Train*, and the path $G0 \xrightarrow{?LOWER} G1 \xrightarrow{!DOWN} G2 \xrightarrow{?RAISE} G3 \xrightarrow{!UP} G4$ should be selected from the specification of the *Gate*.

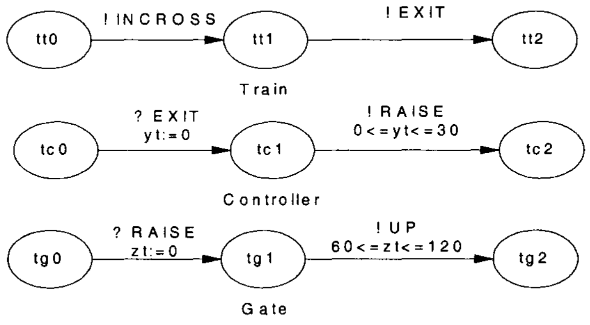


Figure 6-5 Test purpose of the *Train* going out of cross-region, described in TIOA

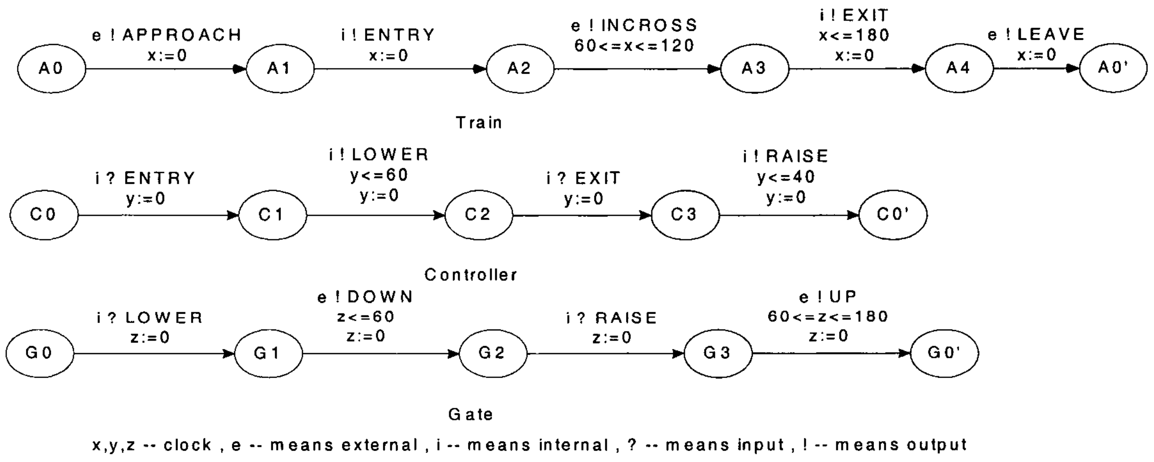


Figure 6-6 The specification after selection

The two steps above are the same in M1, M2, M3. For later steps, M1, M2 and M3 become different by sampling at different positions. M3 starts sampling procedure before synchronous product, while the other two start it after. Here we give the result of synchronous product first and leave sampling behind. Figure 6-7, Figure 6-8, and Figure 6-9 show the results of synchronous product in locations (not states). In Figure 6-7, the transition $(C0, tc0) \xrightarrow{ENTRY y:=0} (C1, tc0)$ is the result of the application of rule 1 (see 3.2.3) because the initial location of the specification has a transition on *ENTRY* but there is no transition on *ENTRY* from the initial location of the test purpose. However, the transition $(C3, tc1) \xrightarrow{!RAISE, 0 \leq y \leq 40 \& 0 \leq yt \leq 30, y:=0, yt:=0} (C0', tc2)$ is the result of the application of rule 2 in section 3.2.3 because both locations *C3* and *tc1* of the specification and the test purpose respectively have a transition on *RAISE*. Note that the time constraint of this transition is the conjunction of the constraints of the transitions of the test purpose and the specification. Moreover, the clocks to be reset by this transition are the union of the clocks to be reset by the transitions of the test purpose and the specification.

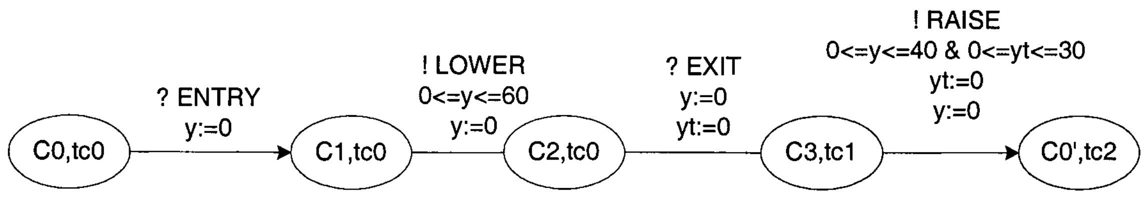


Figure 6-7 Synchronous product of Controller

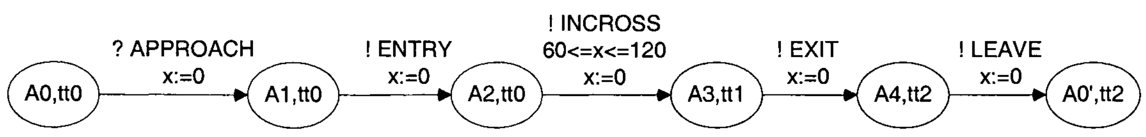


Figure 6-8 Synchronous product of Train

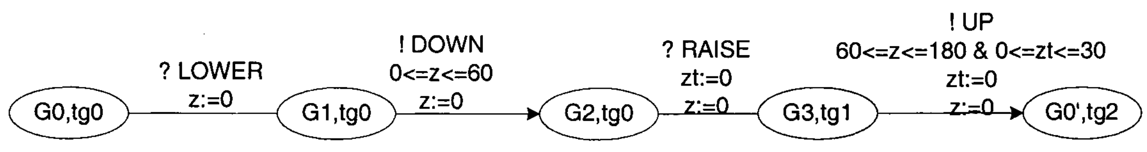


Figure 6-9 Synchronous product of Gate

An example of partial product construction is shown in Figure 6-10. The TIOA of the figure is obtained from the Figure 6-7, Figure 6-8 and Figure 6-9 that are the results of the previous steps. The partial product contains a global transition on each external event. Moreover, a transition on an internal output is included in the partial product whenever a system component produces that output. Finally, a transition on an internal input is included in the partial product only when that input is in the FIFO channel of the system. For example in Figure 6-10, the transition from location (A4, tt2, C2, tc0, G0, tg0) to location (A4, tt2, C3, tc1, G0, tg0) on internal input *EXIT* is possible because the internal output *EXIT* is written to the FIFO channel by the transition from location (A3, tt1, C2, tc0, G0, tg0) to location (A4, tt2, C2, tc0, G0, tg0) or the transition from the location (A3, tt1, C1, tc0, G0, tg0) to the location (A4, tt2, C1, tc0, G0, tg0).

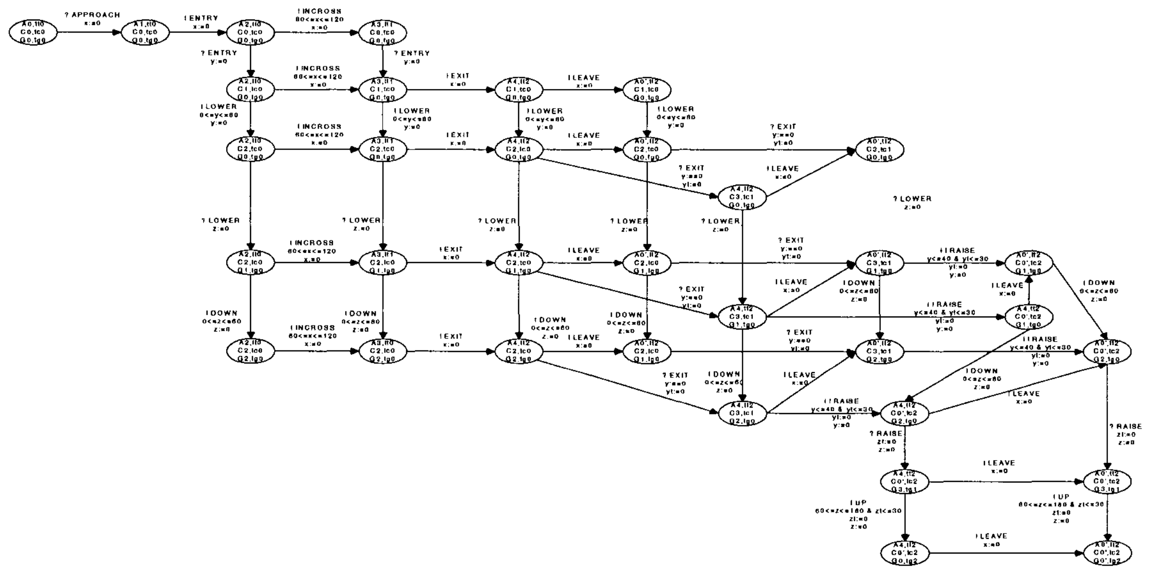


Figure 6-10 the result of Partial product

After the partial product, M1 enters its sampling procedure. In this example, we apply two sampling methods on our TIOA models. One is sampling with different granularities, the other is sampling with the same granularity. The sampling results show us that we generate much less states with the former method.

6.2 Results comparison between M1, M2, and M3

Figure 6-11 shows some test cases generated for a RCS example using the test purpose in Figure 6-4. These test cases do not contain the internal events exchanged between the components of RCS since the tester does not have a direct control over those events and cannot observe them. For example, the test case *"!APPROACH!DOWN?120!INCROSS?60!UP"* does not contain any internal events and means that after the observation of the outputs *APPROACH* and *DOWN*, the tester should wait *120* time-units and observe the output *!INCROSS*, and then waits another *60* time-units and observes the output *!UP*. The summary of the results is shown in Table 6-1.

```
Testcase 0 :
!APPROACH!DOWN?120!INCROSS?60!UP

Testcase 1 :
!APPROACH!DOWN?120!INCROSS?90!UP

Testcase 2 :
!APPROACH!DOWN?120!INCROSS?120!UP

Testcase 3 :
!APPROACH?90!DOWN!INCROSS?60!UP

Testcase 4 :
APPROACH!DOWN?90!INCROSS?90!UP

Testcase 5 :
!APPROACH!DOWN?90!INCROSS?120!UP

Testcase 6 :
!APPROACH?90!INCROSS!DOWN?80!UP

Testcase 7 :
!APPROACH?90!INCROSS!DOWN?110!UP

Testcase 8 :
!APPROACH?90!INCROSS!DOWN?140!UP

Testcase 9 :
!APPROACH!ENTRY?90!INCROSS!DOWN?75!UP
```

Figure 6-11 Example of test cases from Figure 6-10 when M1 is applied

Method	M1	M2	M3
Number of grid states	8089	573	199
Number of Test Cases	61227	25027	5375

Table 6-1 Result for sampling with the same granularity

From the table above, we can see that M3 generates the least number of states and test cases and M1 generates the most ones. This can be explained by the number of clocks used in M1, M2, and M3 to sample the time space of the system under test.

More specifically, method M3 samples each TIOA before the calculation of the synchronous and partial products. Therefore, the number of clocks used in sampling is the number of clocks of each TIOA of the specification and the test purpose. However, method M2 conducts sampling after it calculates the synchronous product for each pair of the test purpose and the specification and before it calculates the partial product of the system. Therefore, the number of clocks used in this method is the sum of the numbers of clocks in a test purpose and its corresponding TIOA specification. Obviously, this number is greater than the one used in method M3. Finally, method M1 conducts sampling on the global TIOA after the calculation of the partial product of the system. In this case, the number of clocks used is the total number of clocks in all TIOAs involved (see section 3.2.7). Hence among our three methods, M1 uses the greatest number of clocks in sampling and therefore generates the largest number of test cases.

6.3 Results comparison between the two sampling methods

As we introduced in section 4.6, we designed two sampling methods to generate the GA model. One is sampling with the same granularity, the other is sampling with different granularities. We will show, in this section, that the latter generates less states than the former by reducing states redundancy.

To illustrate the sampling operation, let us consider the specification of Controller in Figure 6-6. The resulting GA is shown in Figure 6-12. Here, we use different granularities to sample the TIOA of Controller. The transition from location C1 to C2 is sampled with a granularity equal to 30, and the transition from location C3 to location C0' is sampled with a granularity equal to 20. One can easily see, from Figure 6-13, that the number of states in the resulting GA is smaller than that if the same granularity was used to sample the TIOA.

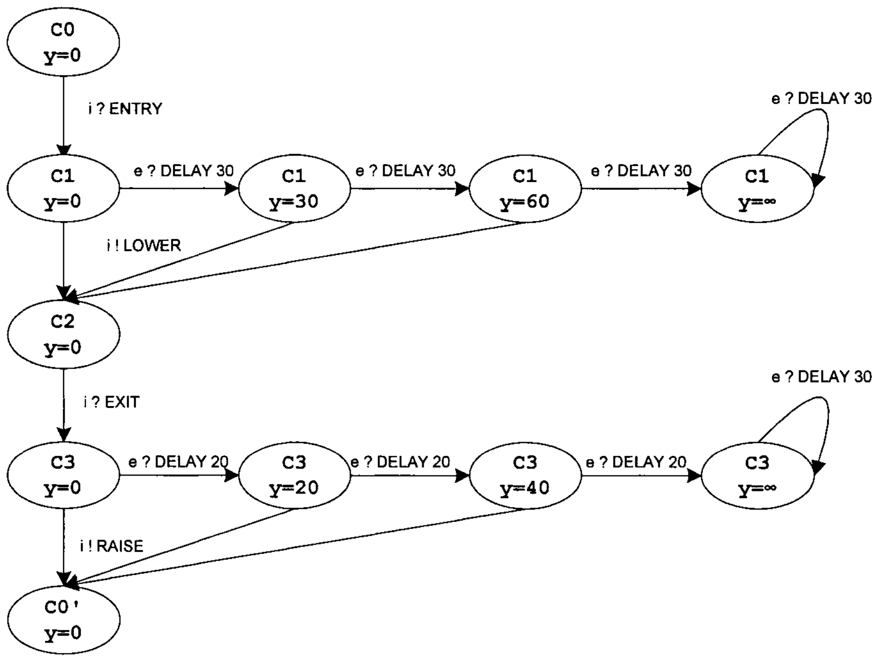


Figure 6-12 Grid TIOA of Controller specification with different sampling

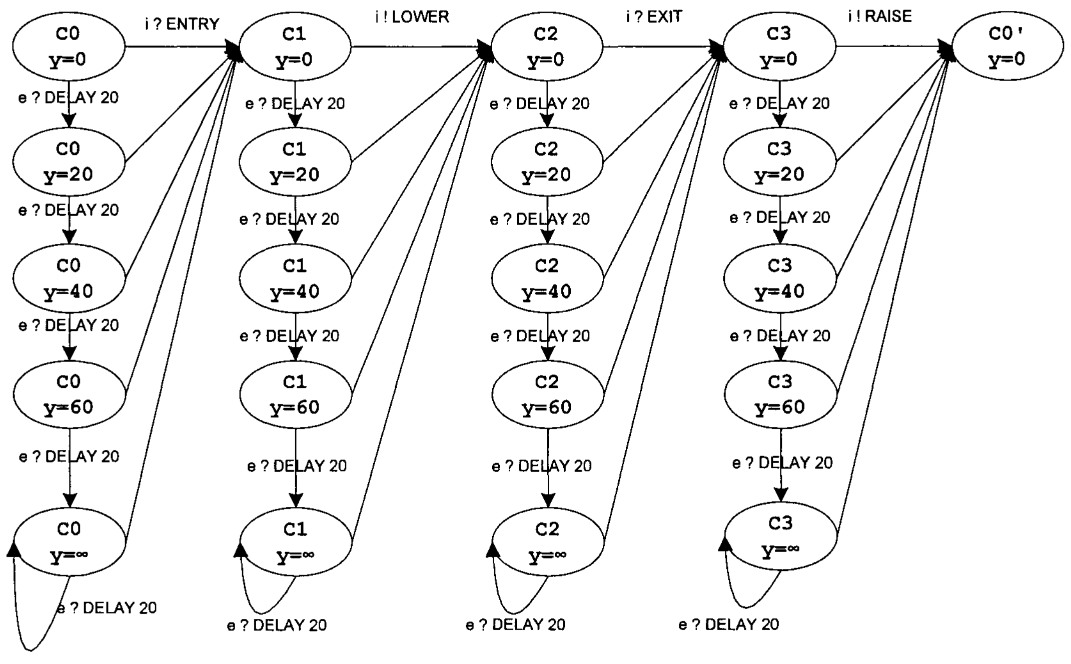


Figure 6-13 Grid TIOA of Controller specification with one sampling unit 20

Table 6-2 shows the different results of the number of states and test cases with applying the different sampling methods. Indeed, as can be seen from the sampling formulas given in section 4.6, the granularity of sampling decreases as the number of clocks increases. Hence, the less the sampling unit is, the more the generated grid states and test cases are.

		M1	M2	M3
With the same granularity	Number of grid states	?	?	?
	Number of Test Cases	?	?	?
With different granularities	Number of grid states	8089	573	199
	Number of Test Cases	61227	25027	5375

Note: ? means no results because the number of states is too large.

Table 6-2 Result for sampling with the same granularity and different granularities

6.4 Conclusion

In this chapter, we applied our implementation introduced in chapter 5 to Railroad Crossing System. We implemented our three methods separately to our example and analyzed the results by comparing the number of states and test cases obtained. The analysis showed that the third method generated fewer states than the other two. We also applied our two sampling algorithms and compared the number of states they generated.

In the next chapter, we will summarize our methodology and discuss possible improvements and extensions of it.

Chapter 7
Conclusion

7.1 Summary of Contributions

Real-Time Systems (RTS) are those systems whose behavior is time sensitive and is governed by time constraints. RTS are usually embedded systems, and consist of many components running concurrently and communicating with each other as well as with the environment. It is well known to the real-time systems research community that the misbehavior of real-time systems is generally due to the non-respect of the timing aspect of their behavior and causes catastrophic consequences on both human lives and the environment.

Testing embedded real-time systems is difficult because: first, there are internal actions that the tester cannot control and observe directly. It means that not all aspects of the Implementation Under Test (IUT) can be tested because they cannot be executed at all. Moreover, the context of the implementation could tolerate certain faults. Thus, the method of the test cases generation for the embedded IUT is quite different from that for the isolated IUT. Second, the state explosion problem is critical in embedded real-time systems testing because of the timing aspect. The situation is getting worse when the number of the embedded components increases.

In this thesis, we presented a methodology based on test purposes expressed as MSCs to generate test cases for embedded real-time systems specified by communicating TIOAs. We implemented three variants of our approach (M1, M2, and M3) and we compared them in terms of the number of states generated and the number of test cases derived. The main difference between M1, M2, and M3 is the position of sampling in the whole process. In M1, sampling is conducted after the calculation of the partial product of the system and hence the granularity of sampling is smaller. However in M2, sampling is conducted after the construction of synchronous product and before the calculation of the partial product and therefore the granularity of sampling is medium. Finally in M3, sampling is conducted at the very beginning for each TIOA of the specification and each test purpose and hence the granularity of sampling is coarser. We noticed based on the fundamentals of M1, M2, and M3 and from the results obtained for each method that the earlier sampling is done the better is the method. We also pointed out the importance of sampling with different granularities in our methods. It is helpful to decrease the number of redundant states in the grid automaton model by sampling with different granularities. Therefore, fewer test cases are generated for testing the IUT.

7.2 Future work

Our work could be extended to cover more aspects of embedded real-time systems testing. Possible extensions are the following:

- Extensions during the translation of Message Sequence Charts (MSC) into Timed Input Output Automata (TIOA).

  MSC is a practical formal description language with complicated syntaxes and semantics. It can be used to describe sophisticated scenarios in reality. According to ITU-T Recommendation Z.120, MSC support control flow, structure design and data entities. In our MSC – TIOA translation operation, we only considered the simplest message sequence charts in which there are only message input events and message output events with time constraints. It is only a small subset of MSC standards. More complicated contents, such as control flow, general ordering, data variables, coregion, inline expression, instance decomposition, MSC reference, and High-level MSC, are left to our future work.

- Extensions in the test notation

The test suite we generated with our methodology is the abstract test suite, which is specified independently of any real testing device and can be standardized as the specification of the executable test suite. The translation of an abstract test suite into an executable test suite can be automatic with the help of TTCN. TTCN is a formal language which is standardized by ISO [38] for the test system specification. If abstract test cases are described in TTCN, they will be feasible to be executed automatically by any tester who supports TTCN inputs.

At present, we described the test suite in text format. We left the test notation in TTCN to our future work.

Bibliography

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

[2] A. En-Nouaary, R. Dssouli, F. Khendek. Timed Wp-Method : Testing Real-Time Systems. *IEEE Transactions on Software Engineering*, November 2002.

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

[4] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir. Test development for communication protocols: Towards automation. (Elsevier) Comput. Networks, 31:1835--1872, 1999.

[5] M. A. Fecko, P. D. Amer, M. U. Uyar, and A. Y. Duale. Test Generation in the presence of Conflicting Timers. In *TESTCOM Ottawa, Canada,* August-September 2000.

[6] 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, Newport Beach, California*, February 1997.

[7] Rachel Cardell-Oliver and Tim Glover. A Practical and Complete Algorithm for Testing Real-Time Systems In *FTRTFT1998 – Formal Techniques for Real-Time Fault Tolerant Systems, Lyngby, Danmark*, 1998.

[8] D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365-398, November 1995.

[9] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In *5th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems FTRTFT' 98*, September 1998.

[10] 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.

[11] Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek and Abdelkader Elqortobi Testing Embedded Real-Time Systems, RTCSA'2000, Cheju Island Korea, 2000.

[12] A. Khoumsi, M. Akalay, R. Dssouli, A. En-Nouaary, and L. Granger. An Approach For Testing Real-Time Protocols. In *TESTCOM Ottawa, Canada*, August-September 2000.

[13] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. In *Proceedings of the International Workshop on Testing Communicating Systems (IWTCS' 99), Budapest, Hungary*, 1999.

[14] J. Springintveld, F. Vaadranger, and P. Dargenio. Testing Timed Automata. Journal of Theoretical Computer Science, 254, pages 225-257, 2001.

[15] R. Castanet, O. Kone, and P. Laurencot. On the Fly Test Case Generation for Real Time Protocols. ICN'98, Louisiana, USA, 1998.

[16] ITU-T, Message Sequence Charts--MSC-2000, ITU-T Recommendation Z.120, November 1999.

[17] C.Heimeyer, R. Jeffords, and B.Labaw. Comparing Different Approaches for Specifying and Verifying Real-Time Systems. In *Proc. 10th IEEE Workshop on Real-Time Operating Systems and Sofware*, pages 122-129, May 1993.

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

[19] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. IN *SDL' 93*, October 1993.

[20] K.G. Larsen and W. Yi. Time Abstracted Bisimulation: Implicit Specification and Decidability. In *Proceedings Mathematical Foundations of Programming Semantics (MFPS 9), volumn 802 of Lecture Notes in Computer Science*, New Orleans, USA, April 1993. Springer-Verlag.

[21] C.Heimeyer, R. Jeffords, and B.Labaw. Comparing Different Approaches for Specifying and Verifying Real-Time Systems. In Proc. 10th IEEE Workshop on Real-Time Operating Systems and Sofware, pages 122-129, May 1993.

[22] Stephen R. Schach Vanderbilt University. Object-Oriented And Classical Sofware Engineering. Published by McGraw-Hill, a business unit of The McGraw-Hill Companies,Inc., 1221 Avenue of the Americas, New York, NY10020.

[23] ISO Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[24] Martha Gray, Alan Goldfine, Lynne Rosenthal, Lisa Carnahan: Conformance Testing. http://www.oasis-open.org/cover/conform20000112.html

[25] J. Tretmans An Overview of OSI Conformance Testing. Translated and adapted from: J. Tretmans and J. van de Lagemaat, Conformance Testen, in Handboek Telematica, Vol. II, pages 4400 1--19. Samson, 1991.

[26] A. Petrenko, G.v. Bochmann, and M. Yao, On Fault Coverage of Tests for Finite State Specifications, Computer Networks and ISDN Systems, vol. 29, pp. 81-106, Dec. 1996.

[27]Ana R. Cavalli, Jean Philippe Favreau, Marc Phalippou: Formal Methods for Conformance Testing: Results and Perspectives. Protocol Test Systems 1993: 3-17

[28] D. P. Sidhu and T. K. Leung, Formal Methods for Protocol Testing: A Detailed Study, IEEE Trans. SE-15, No.4, April 1989, pp. 413-425.

[29]G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo, Fault Models in Testing, IFIP Transactions, Protocol Test Systems, IV (the Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991), Ed. by Jan Kroon, Rudolf J. Heijink and Ed Brinksma, 1992, North-Holland.

[30]A. Petrenko and N. Yevtushenko, Test Suite Generation for a FSM with a Given Type of Implementation Errors, IFIP Transactions, Protocol Specification, Testing, and Verification, XII (the Proceedings of IFIP TC6 12th International Symposium on Protocol Specification, Testing, and Verification, 1992), Ed. by R.J. Linn. Jr. and M.U. Uyar, 1992, North-Holland, pp. 229-243.

[31]H. Ural, Formal Methods for Test Sequence Generation, Computer Comm., Vol. 15,No. 5, 1992, pp. 311-325.

[32]A. Petrenko, G. v. Bochmann, and R. Dssouli, Conformance Relations and Test Derivation, IFIP Transactions, Protocol Test Systems, VI, (the Proceedings of IFIP TC6 Sixth International Workshop on Protocol Test Systems, 1993), Ed. by O. Rafiq, 1994, North-Holland, pp. 157-178.

[33] A. Faro and A. Petrenko, Sequence Generation from EFSMs for Protocol Testing, Participants' Proc. of COMNET'90, Budapest, 1990.

[34] R. Probert and F. Guo, Mutation Testing of Protocols: Principles and Preliminary Results, Protocol Test Systems, III, (Proceedings of IFIP TC6 Third International Workshop on

Protocol Test Systems, 1990), Ed. by I. Davidson and D. W. Litwack, 1991, North-Holland, pp. 57-76.

[35] R. E. Miller and S. Paul, Generating Conformance Test Sequences for Combined Control and Data of Communication Protocols, IFIP Transactions, Protocol Specification, Testing, and Verification, XII (the Proceedings of IFIP TC6 12th International Symposium on Protocol Specification, Testing, and Verification, 1992), Ed. by R.J. Linn. Jr. and M.U. Uyar, 1992, North-Holland, pp. 1-15.

[36] S. T. Chanson and J. Zhu, A Unified Approach to Protocol Test Generation, in Proc. Of the IEEE INFOCOM'93, pp.106-114.

[37]C.-J. Wang and M. T. Liu, Generating Test Cases for EFSM with Given Fault Models, Proc. of the IEEE INFOCOM'93, pp. 774-781.

[38] ISO/IEC. Information Technology – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN). International Standard ISO/IEC 9646-3. ISO/IEC, Geneve, 1997. Second Edition.

[39] Lars Mats, Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, 1998, IEEE Computer Society Washington, DC, USA

[40] Bhaskar Rao.G and Albee Vimal, SDL Based Test Automation for Real Time Systems Testing, Motorola India Electronics Ltd.

[41] A. Petrenko, N. Yevtushenko, G. v. Bochmann, R. Dssouli, Testing in context: A framework and test derivation, Special Issue on Protocol Engineering of Computer Communication, 1997.

[42] L.P. Lima, A. Cavalli, A pragmatic approach to generating test sequences for embedded systems, in: Proc. IWTCS'97, Cheju islands, Korea, 1997.

[43] Zhang Xiang and Abdeslam En-Nouaary, Test Cases Generation for Embedded Real-Time Systems Based on Test Purpose, in: NOTERE' 2004, Saidia, Maroc, 2004.

[44] Jens Grabowski, The Generation of TTCN Test Cases from MSCs, Technical Report IAM-94-004, University of Berne, Institute for Informatics, Berne, Switzerland, May 1994.

[45] The Telelogic Tau TTCN Suite, http://www.telelogic.com/products/tau/ttcn/index.cfm

Appendix A
Regular Expression Table of Lex

```
%{

  #include <stdio.h>

  #include <stdlib.h>

  #include <string.h>

  #include "MyLib.h"

  #include "lexical.cpp.h"

  #include "y.tab.h"

%}

%e2000

%p8000

%n1500

%a 30000

%o 40100

LETTER            [A-Za-z]

DECIMAL_DIGIT         [0-9]

OTHER_CHARACTER     ["?"|"%"|"+"|"\-"|"!"|"/"|">"|"*"|"\\"|"<"|"="]

%%

";"        {return ENDL;}

"&"        {return REL_TIME_MARK;}

"@"        {return ABS_TIME_MARK;}
```

```
"["     {return LEFT_CLOSED;}

"]"     {return RIGHT_CLOSED;}

"("     {return LEFT_OPEN;}

")"     {return RIGHT_OPEN;}

":"     {return DBL_DOT;}

","     {return PUNCT;}

"<<"    {return QUALIFIER_LEFT;}

">>"    {return QUALIFIER_RIGHT;}

[aA][cC][tT][iI][oO][nN]  {return ACTION;}

[aA][fF][tT][eE][rR]        {return AFTER;}

[aA][lL][lL]        {return ALL;}

[aA][lL][tT]        {return ALT;}

[aA][sS] {return AS;}

[bB][eE][fF][oO][rR][eE]  {return BEFORE;}

[bB][eE][gG][iI][nN]        {return BEGINX;}

[bB][lL][oO][cC][kK]        {return BLOCK;}

[bB][yY]            {return BY;}

[cC][aA][lL][lL]  {return CALL;}

[cC][oO][mM][mM][eE][nN][tT]    {return COMMENT;}

[cC][oO][nN][cC][uU][rR][rR][eE][nN][tT] {return CONCURRENT;}

[cC][oO][nN][dD][iI][tT][iI][oO][nN]         {return CONDITION;}

[cC][oO][nN][nN][eE][cC][tT]       {return CONNECT;}
```

```
[cC][rR][eE][aA][tT][eE]  {return CREATE;}

[dD][aA][tT][aA]{return DATA;}

[dD][eE][cC][oO][mM][pP][oO][sS][eE][dD]          {return DECOMPOSED;}

[dD][eE][fF]       {return DEF;}

[eE][mM][pP][tT][yY]      {return EMPTY;}

[eE][nN][dD]       {return END;}

[eE][nN][dD][aA][fF][tT][eE][rR]  {return ENDAFTER;}

[eE][nN][dD][bB][eE][fF][oO][rR][eE]       {return ENDBEFORE;}

[eE][nN][dD][cC][oO][nN][cC][uU][rR][rR][eE][nN][tT]       {return ENDCONCURRENT;}

[eE][nN][dD][eE][xX][pP][rR]       {return ENDEXPR;}

[eE][nN][dD][iI][nN][sS][tT][aA][nN][cC][eE]         {return ENDINSTANCE;}

[eE][nN][dD][mM][eE][tT][hH][oO][dD]     {return ENDMETHOD;}

[eE][nN][dD][mM][sS][cC]           {return ENDMSC;}

[eE][nN][dD][mM][sS][cC][dD][oO][cC][uU][mM][eE][nN][tT]          {return ENDMSCDOCUMENT;}

[eE][nN][dD][sS][uU][sS][pP][eE][nN][sS][iI][oO][nN]         {return ENDSUSPENSION;}

[eE][nN][vV]       {return ENV;}

[eE][qQ][uU][aA][lL][pP][aA][rR] {return EQUALPAR;}

[eE][sS][cC][aA][pP][eE]  {return ESCAPE;}

[eE][xX][cC]       {return EXC;}

[eE][xX][pP][rR] {return EXPR;}

[eE][xX][tT][eE][rR][nN][aA][lL]  {return EXTERNAL;}

[fF][iI][nN][aA][lL][iI][zZ][eE][dD]          {return FINALIZED;}
```

```
[fF][oO][uU][nN][dD]      {return FOUND;}

[fF][rR][oO][mM]          {return FROM;}

[gG][aA][tT][eE] {return GATE;}

[iI][nN]  {return IN;}

[iI][nN][fF]        {return INF;}

[iI][nN][hH][eE][rR][iI][tT][sS]     {return INHERITS;}

[iI][nN][lL][iI][nN][eE]     {return INLINE;}

[iI][nN][sS][tT]    {return INST;}

[iI][nN][sS][tT][aA][nN][cC][eE]    {return INSTANCE;}

[iI][nN][tT]_[bB][oO][uU][nN][dD][aA][rR][yY]      {return INT_BOUNDARY;}

[lL][aA][bB][eE][lL]        {return LABEL;}

[lL][aA][nN][gG][uU][aA][gG][eE]            {return LANGUAGE;}

[lL][oO][oO][pP] {return LOOP;}

[lL][oO][sS][tT]  {return LOST;}

[mM][eE][tT][hH][oO][dD]           {return METHOD;}

[mM][sS][cC]      {return MSC;}

[mM][sS][cC][dD][oO][cC][uU][mM][eE][nN][tT]   {return MSCDOCUMENT;}

[mM][sS][gG]      {return MSG;}

[nN][eE][sS][tT][aA][bB][lL][eE]  {return NESTABLE;}

[nN][oO][nN][nN][eE][sS][tT][aA][bB][lL][eE]       {return NONNESTABLE;}

[oO][fF][fF][sS][eE][tT]   {return OFFSET;}

[oO][pP][tT]       {return OPT;}
```

```
[oO][rR][dD][eE][rR]        {return ORDER;}

[oO][tT][hH][eE][rR][wW][iI][sS][eE]        {return OTHERWISE;}

[oO][uU][tT]        {return OUT;}

[pP][aA][rR]        {return PAR;}

[pP][aA][rR][eE][nN][tT][hH][eE][sS][iI][sS]        {return PARENTHESIS;}

[pP][rR][oO][cC][eE][sS][sS]        {return PROCESS;}

[rR][eE][cC][eE][iI][vV][eE]        {return RECEIVE;}

[rR][eE][dD][eE][fF][iI][nN][eE][dD]        {return REDEFINED;}

[rR][eE][fF][eE][rR][eE][nN][cC][eE]        {return REFERENCE;}

[rR][eE][lL][aA][tT][eE][dD]        {return RELATED;}

[rR][eE][pP][lL][yY][iI][nN]        {return REPLYIN;}

[rR][eE][pP][lL][yY][oO][uU][tT]  {return REPLYOUT;}

[sS][eE][qQ]        {return SEQ;}

[sS][eE][rR][vV][iI][cC][eE]        {return SERVICE;}

[sS][hH][aA][rR][eE][dD] {return SHARED;}

[sS][tT][aA][rR][tT][aA][fF][tT][eE][rR]    {return STARTAFTER;}

[sS][tT][aA][rR][tT][bB][eE][fF][oO][rR][eE]        {return STARTBEFORE;}

[sS][tT][aA][rR][tT][tT][iI][mM][eE][rR]    {return STARTTIMER;}

[sS][tT][oO][pP]  {return STOP;}

[sS][tT][oO][pP][tT][iI][mM][eE][rR]        {return STOPTIMER;}

[sS][uU][sS][pP][eE][nN][sS][iI][oO][nN]   {return SUSPENSION;}

[sS][yY][sS][tT][eE][mM] {return SYSTEM;}
```

```
[tT][eE][xX][tT]          {return TEXT;}

[tT][iI][mM][eE] {return TIME;}

[tT][iI][mM][eE][oO][uU][tT]       {return TIMEOUT;}

[tT][iI][mM][eE][rR]       {return TIMER;}

[tT][oO] {return TO;}

[uU][nN][dD][eE][fF]       {return UNDEF;}

[uU][sS][iI][nN][gG]       {return USING;}

[uU][tT][iI][lL][iI][tT][iI][eE][sS]  {return UTILITIES;}

[vV][aA][rR][iI][aA][bB][lL][eE][sS]         {return VARIABLES;}

[vV][iI][aA]        {return VIA;}

[vV][iI][rR][tT][uU][aA][lL]         {return VIRTUAL;}

[wW][hH][eE][nN]          {return WHEN;}

[wW][iI][lL][dD][cC][aA][rR][dD][sS]        {return WILDCARDS;}

[0-9]+         {yylval.numbers=atoi(yytext);return NUMBER;}

[A-Za-z_]+[A-Za-z0-9"_""."]*       {strcpy(yylval.strs,yytext);return NAME;}

"".*""   {strcpy(yylval.strs,yytext);return CHARACTER_STRING;}

[A-Za-z0-9]+([A-Za-z0-9]|{OTHER_CHARACTER})*        {strcpy(yylval.strs,yytext);return WORDS;}

"/*".*"*/"          {;}

[\t\n\r]   {;}

%%
```

Appendix B
Textual Grammar For TIOA

- Lexical Rules:

<lexical unit> ::= <character string> | <decimal digit> | <special> | <keyword>

<character string> ::= <letter>{<letter>|<decimal digit>}*

<letter>::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X
| Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<decimal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special> ::= <relational operators> | <and> | <or> | <input sign> | <output sign> | <left open> |
<right open> | <end> | <assignment>

<relational operators> ::= > | < | = | >= | <=

<logical operators> ::= <and> | <or>

<and> ::= &

<or> ::= |

<input sign> ::= ?

<output sign> ::= !

<left open> ::= (

<right open> ::=)

<end> ::= ;

<assignment> ::= =

<keyword> ::= **begin** | **Clock** | **end** | **EndSystem** | **ENV** | **from** | **Input** | **Locations** | **Output** |
System | **to** | **Transitions**

- Concrete textual grammar

<specification> ::= **begin** <tioa>+ **end**<end>

<tioa> ::= **System** <tioa head> <tioa body> **EndSystem**;

<tioa head> ::= <tioa ID><end>

<tioa body> ::= <inputs><outputs><locations><clocks><transitions>

<inputs> ::= **Input**:<input item>[,<input item>]*<end>

<input item> ::= <message ID> **from** <tioa ID> | **ENV**

<outputs> ::= Output:<output item>[,<output item>]*<end>

<output item> ::= <message ID> **to** <tioa ID> | **ENV**

<locations> ::= **Locations**:<location ID> [,<location ID>]<end>

<clocks> ::= **Clock**:<clock item>*

<clock item> ::= <clock ID><upbound><end>

<upbound> ::= <left open><value><right open>

<transitions> ::= **Transitions**:<transition item>+

<transition item> ::= <transition ID><assignment><left open><source location ID>, <destination location ID>,{<input sign>|<output sign>}<message ID>,[<logical expression>],[<clock reset>]<right open>

< logical expression> ::= <relational expression>[< logical operators> < relational expression>]*

<relational expression> ::= <left open><clock ID><relational operators><value><right open>

<clock reset> ::= (<clock ID>:=<value>)

<ID> ::= <character string>

<value> ::= <decimal digit>+