AUTOMATED TEST GENERATION FROM FORMAL SPECIFICATIONS OF REAL-TIME REACTIVE SYSTEMS

Mao Zheng

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

April 2002

0-612-68212-9

Abstract

Automated Test Generation from Formal Specifications of Real-Time Reactive Systems

Mao Zheng, Ph.D.
Concordia University, 2002

Real-time reactive systems are among the most difficult systems to design and implement because of their size and complex functional and timing requirements. They are often used in safety-critical contexts. Consequently, the correction of such systems must be assured before they are deployed. This thesis addresses the quality assurance of real-time reactive systems through rigorous testing methods. The thesis gives methods to generate test cases from the formal specifications of real-time reactive systems developed in TROMLAB framework.

The scope of this thesis encompasses two major components. A Homomorphism Theorem is given that serves as a basis for automated test case generation from the grid automaton associated with the extended state machine formalism. A number of algorithms are also given for generating test cases for black-box testing of reactive class implementations, implementations of class refinements and system configurations. The testing methodologies are theoretically sound, yet being practical for automated test case generation.

To my parents.

Acknowledgments

I would like to deeply thank my supervisor, Dr. V.S. Alagar. He guided me throughout my study at Concordia University and played a significant role in shaping my research career. His rigorous research attitude and professional shrewdness were fundamental to the success of this research work. Without his wisdom, his unfailing guidance, constant encouragement and insightful comments, this work would not have been possible. I am really fortunate to have had such a great supervisor to go through this long journey.

I am grateful to the TROMLAB research group for thought-provoking discussions.

I thank the faculty, staff and students of the Computer Science Department at Concordia University for providing me with a stimulating, and yet personal environment to work.

I gratefully acknowledge the financial support provided by my supervisor during my study at Concordia University. I also thank Concordia University for awarding me the Graduate Fellowships, International Tuition Fee Remission and Carolyn and Richard Renaud Teaching Assistantship.

On a personal level, I thank my parents and my brother for giving me all the best that anyone can ever ask for. Their love and support accompany me at all times.

I would also like to bring attention to Catherine and Richard who have brought significance to this accomplishment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real-Time Reactive System

The term *reactive* was introduced by Harel and Pnueli [HP85] to designate systems that continuously interact with their environment through stimulus and response. The term reactive also distinguishes the stimulus and response synchronizations in reactive systems from those that are available in *interactive* and *transformational* systems. The real-time behavior of a reactive system is regulated by time and persists across some interval of time during which it responds to inputs as they occur. Two important properties characterize real-time reactive systems:

- *stimulus synchronization*: the process always reacts to a stimulus from its environment;
- *response synchronization*: the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment, so that the environment is still receptive to the response.

A real-time system is a software system that maintains an ongoing and timely interaction with its environment. The behavior of a real-time reactive system encompasses concurrency, communication through sensors and actuators, and relations between input and output over time. Such systems often operate in *safety-critical* contexts, with applications in control systems for nuclear reactors, air traffic, railroad crossings, telecommunications, and medicine. In safety-critical applications, the consequences of failure are extremely high, usually with a threat to human life. Safety-critical systems are expected to perform as desired and should never fail.

Generally, the behavior of a reactive system is *infinite*: the process in a reactive system is usually continuously responding to the stimuli from its environment. The correct behavior of non-real time systems is founded on the *functional correctness* of the result. By contrast, real-time reactive systems require both the functional correctness of the result and its *timing correctness*. Hence, *safety* of such systems, which concerns prevention of risks to human life or property, relies on good analysis of the functional and timing properties of those systems. Nevertheless, real-time reactive systems are large and complex and consequently understanding or describing the behavior of such systems becomes very difficult.

In recent studies [RP95], formal methods have been applied to manage the complexity in modeling and implementing real-time reactive systems. Software-development techniques based on mathematics are called formal methods, and they are beginning to become both practical to apply and capable of providing substantial improvements in software quality. Two broad categories of techniques for assuring high-quality software are *formal verification* [Mut00] and *testing* [BG93, Heg89]. Although formal verification is based on a sounder theoretical foundation than testing, it is generally considered impractical to formally verify large complex software products. Therefore testing has remained the most commonly used technique for ensuring software quality.

1.2 Research Goals

The goal of this research is the investigation of testing methods for real-time reactive systems founded on formal methods. Softwares that operate and control safety-critical applications must be thoroughly tested before the software is installed in the operational environment. This thesis proposes a testing methodology based on the formal software specification and gives algorithms for automated test case generation of real-time reactive systems. We provide a theoretical basis to justify the algorithms for test case generation and propose methods for efficiently implementing them.

The factors that contribute to the complexity of real-time reactive systems include largeness, concurrency, time constraints on stimuli and responses, complex sequencing of events, and an incomplete knowledge of the environment wherein it is supposed to be embedded. The complexity of the system permeates from requirement analysis to design, verification, implementation and testing. Time constraints, in addition to functionality, must be correctly carried through design refinements. The design must be verified for safety properties and validated for time-dependent functionalities. Verification deals with the specification of the system under consideration and aims to ensure that design specification satisfies predefined functional and timing requirements. A mechanical verification approach for reactive systems, proposed by Muthiayen and Alagar [MA99], complements the testing method discussed in this thesis.

The goal of testing is to uncover errors and improve the quality of software. An implementation of a system that has been formally verified for safety properties should still be tested. This is to guard against the errors due to resource constraints that might forbid the timed behavior at certain instances in the operational environment of the software. In this thesis, a black-box testing approach is investigated, limited to generating test cases from formal specifications. This approach is *conformance-directed*, therefore the intent is to provide a method for system developer to demonstrate conformance of the implementation to the design specification. Conformance-directed testing need not consider potential implementation faults in detail, but must establish that a test suite is sufficiently representative of the properties/features for a system. It is our view that black-box testing can adequately complement formal verification and validation effort in assuring an acceptable level of accuracy in an implementation.

Black-box testing relies on the *quality* of the specification of the system under test. If the specification is loose, the effectiveness of testing will be weak. Usually the results obtained from an implementation are compared with an oracle and the correctness of the implementation is certified only when they match in all test cases. When a *good* formal specification of the system is available, it can be used both as an oracle and a medium to generate test cases. The motivation for this research work comes from the following considerations:

- The correctness and safety requirements of real-time reactive systems must be assured in a safety-critical context. Test cases generated from a design that correctly incorporates the safety requirements are sufficient to certify the safety in an implementation.

- The object-oriented software development life cycle and its artifacts adopted by *Timed Reactive Object Model* (TROM) formalism [Ach95] are different from traditional real-time software development practices. Traditional software testing does not apply directly to object-oriented software.

- There is considerable value added to the entire development process when a rigorous testing is correctly combined with software development.

- There is an inherent limitation of formal verification for real-time reactive systems: methods based on *model checking* and *theorem proving* have exponential time complexity even when the model of time is discrete.

- Although specification-based testing methods for untimed systems have been studied in depth, not much work has been done for rigorous testing of timed systems. We take up the discussion on related work in Chapter 4.

1.3 Major Contributions and Thesis Outline

In this thesis, a methodology is given for testing real-time reactive systems developed in TROMLAB [AAM98] framework. The methodology includes methods for testing classes (unit testing), refinements of classes (inheritance testing), and system configurations (system testing). The methodology, although strictly follows TROM formalism, is adaptable for testing real-time reactive systems whose behavior can be abstracted as timed labeled transition systems.

Since the behavior of a real-time reactive system may be infinite, testing its behavior is a non-trivial problem. The significance of the research reported in this thesis lies in construction of a grid automaton with finite number of states and whose behavior can be homomorphically mapped to the infinite timed behavior of the labeled transition system that models the behavior of a reactive class. This construction provides the basis for an algorithm to generate test cases for unit testing.

Chapter 2 reviews TROMLAB, an environment for developing real-time reactive systems according to TROM formalism and gives the context of the research described in this thesis. Test case generation algorithms and a test driver for injecting test cases into a Java implementation are being integrated with the automatic code generation tool [Zha00]. Chapter 3 reviews TROM formalism and Real Time UML

(RTUML) [Mut00] to illustrate that UML models of reactive systems consistent with TROM formalism can be constructed. The motivation here is twofold: to introduce TROM formalism on which the testing methodology is founded, and to suggest that our testing methodology is applicable to UML models of reactive systems that have TROM as the semantic basis. Chapter 4 provides a survey of related work on testing real-time reactive systems with particular emphasis on black-box testing. Although a number of methods have been reported for testing object-oriented softwares, no efficient method has been reported yet for a rigorous testing of real-time reactive systems. Chapter 5 contains several major results:

- Test Adequacy: Two test adequacy criteria are given. These are used in test case generation algorithms.
- Homomorphism: A homomorphism theorem is stated as the basis for the construction of a finite state grid automaton.
- Algorithms: Algorithms are given for generating sufficient number of test cases for a reactive class.

Chapter 6 provides methods for incrementally generating test cases for inherited classes. Three forms of inheritance are defined in [Ach95]. Chapter 7 gives an algorithm for generating test cases for testing a system configured with two interacting objects that are instances of classes for which test cases have already been generated. The algorithm involves two steps:

- the synchronous product machine of the two instantiated objects is constructed, and
- test cases are generated from the grid automaton of the synchronous product machine.

Chapter 8 discusses a method for minimizing the complexity of testing a real-time reactive system composed with n ($n > 2$) objects. The system configuration is decomposed into smaller components in such a way that

- each component, abstracted as a directed graph, is complete,
- the combined behavior of the components is equivalent to the behavior of the whole system.

An algorithm is given for generating test cases of the whole system as a composition of test cases generated for smaller components. The crucial step is in the utilization of the test cases generated for pairs of interacting objects in generating test cases for each component. Chapter 9 concludes the thesis with a discussion of future directions of research.

Chapter 2

TROMLAB - Context of Testing

2.1 Introduction

The research work reported in this thesis has been done in the context of TROMLAB [AAM98], a formal framework for rigorous development of real-time reactive systems. The prototype TROMLAB environment, shown in Figure 1, has been constructed over the last six years and used as a testbed for real-time reactive systems development.

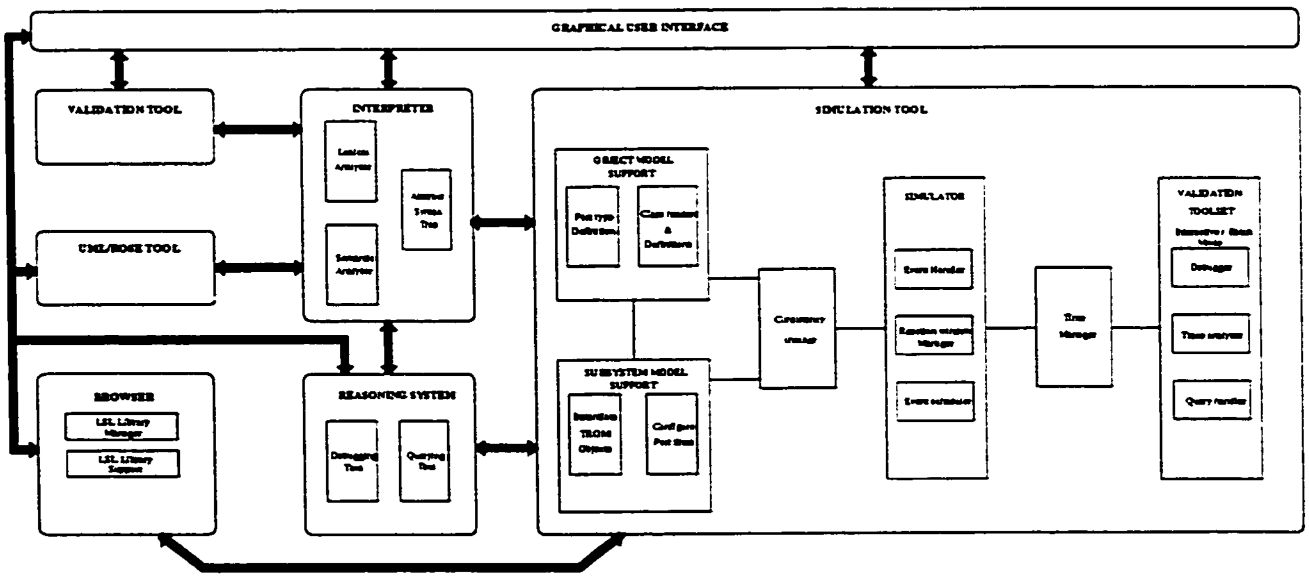


Figure 1: TROMLAB Architecture

The following components of TROMLAB are currently operational:

- **Interpreter** - [Tao96] A parser, syntax checker and internal representation constructor;
- **Simulator** - [Mut96] A tool that simulate a subsystem based on the internal representation and enables a systematic validation of the specified system;
- **Browser** - [Nag99] A tool that help users navigate, query and access various system components for reuse during system development;
- **UML-RT Support** - [Pop99] A translator to generate TROM specification from Real-time UML;
- **Verification Assistant** - [Pom99] A tool to generate PVS theory from TROM specification for proving timing properties;
- **Graphical User Interface** - [Sri99] A visual modeling and interaction facility for a developer using the TROMLAB environment;
- **Reasoning System** - [Hai99] A tool to provide a means of debugging the system during animation by facilitating interactive queries of hypothetical nature on system behavior.
- **Code Generator** - [Zha00] It automatically generate Java code from TROM specifications.

The testing tool built as part of this thesis will be integrated with the interpreter, the code generator, and the metrics tool being built by Ormandjieva [Orm02].

Software development activity in TROMLAB framework, assisted by the above tools, is carried out according to TROM formalism, integrated in the process model shown in Figure 2. The primary distinction between traditional process models and the process model of TROMLAB is in the integration of rigorous methods through the different stages of development. The Graphical User Interface provides complete transparency of the formalism to software developers in the framework.

The process model requires a formal model of the environment to be produced and integrated with system elements. Environment objects are abstracted and their interface to system elements are formally defined. System requirements which include functional and timing requirements are identified and their formal descriptions are produced. A reactive system model is composed from the software unit and the model

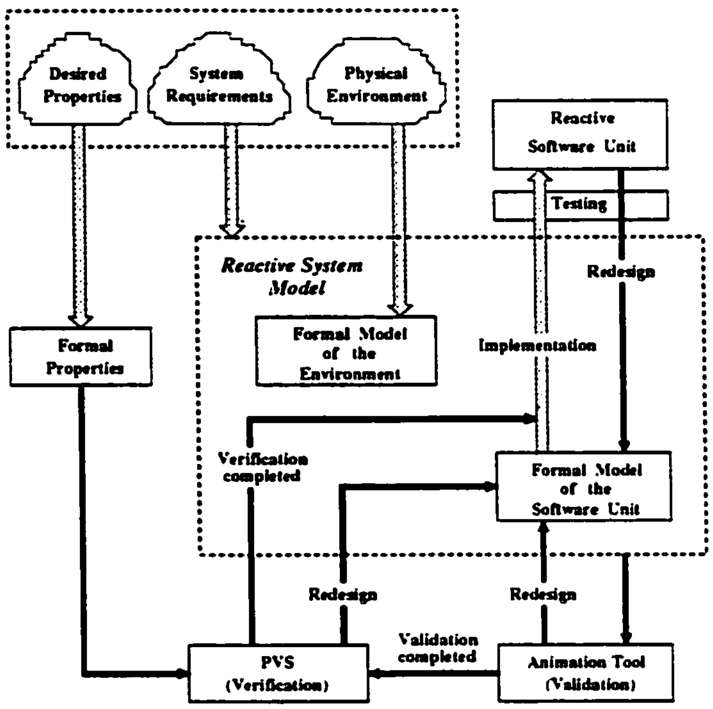


Figure 2: Process Model for Developing Complex Reactive Systems

of the environment. Integration of Rational Rose with initial stages of model building activities enables the construction of visual models of reactive systems.

The formal model of the reactive unit is not implemented until several iterations of design take place. Validation is done by animating the reactive system design. Simulation and reasoning are two techniques used to debug the design and predict its behavior. Simulation and reasoning tools use the formal model and hence is independent from any implementation constraints such as resource and process speed. This experimentation gives better insight into what the crucial properties of the design are and how these are directly related to the requirements. If flaws due to incorrect functions or inconsistent timing behavior are noticed during system simulation and reasoning, the process model allows an iterative inner cycle for redefining the formal model of the reactive unit and validating it.

System verification takes place at the next stage of the process model. Time critical properties, such as safety properties, are verified formally at this stage. The system design is mechanically translated to a set of PVS theories consisting of axioms describing the timed behavior of the system. The desired properties are formalized and are included as lemmas in PVS theories. A mechanized verification procedure is discussed in [Mut00]. An iterative cycle enables redesigning the system when the design does not satisfy the desired properties.

Test suits are generated from the verified and validated design specification, and are independent from implementation decisions. In TROMLAB framework black-box testing is used to test the conformance of the code generated by [Zha00] with respect to the verified and validated design specifications. Thus, the generated code certified by the black-box testing can be trusted to conform to the initial requirements.

2.2 TROM Formalism

In this section we review TROM formalism [Ach95] which is founded on merging object-oriented and real-time technologies. Our testing approach is modular to suit the TROM formalism's abstract three-tiered structure. Figure 3 shows the three-tiered structure of the TROM methodology. Abstract models from a lower tier can be included in upper tiers. The formalism is sufficiently expressive for modeling large real-time reactive systems.

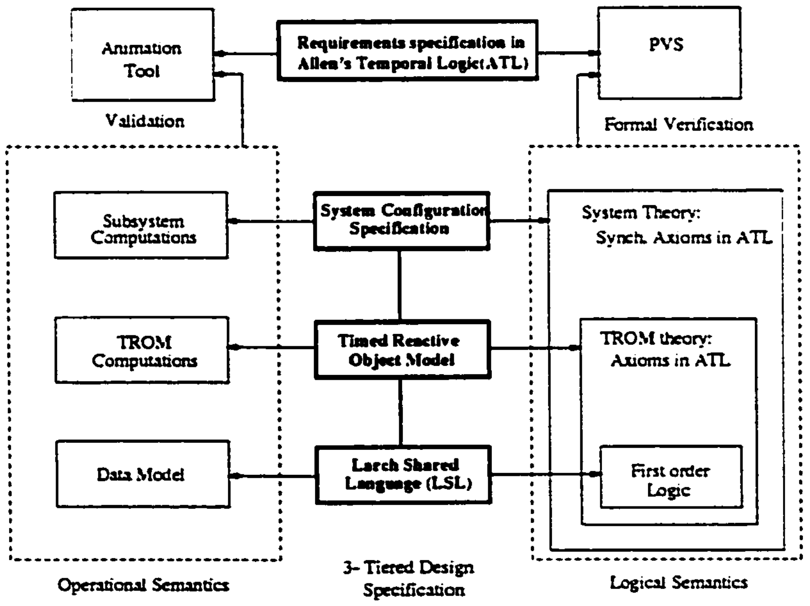

Animation Tool
Requirements specification in Allen's Temporal Logic(ATL)
PVS
Validation
Formal Verification
Subsystem Computations
System Configuration Specification
System Theory: Synch. Axioms in ATL
TROM Computations
Timed Reactive Object Model
TROM theory: Axioms in ATL
Data Model
Larch Shared Language (LSL)
First order Logic
Operational Semantics
3-Tiered Design Specification
Logical Semantics

Figure 3: Overview of TROM Methodology

2.2.1 First Tier - Abstract Data Types

Abstract data types and theories can be specified in the first tier in Larch [GH93] style. For instance, the abstract data type *Set* is defined by the LSL (Larch Shared Language Trait) in Figure 4. The different sections of a trait define included traits, the signature for operations on the sort defined by the trait, the assertions constraining the operations, and the logical consequences that can be inferred from the assertions. In Figure 4, the *introduces* clause declares a set of operators, each with its *signature* (the sorts of its domain and range). The *body* of the trait contains, following the *asserts* clause, equations between terms containing operations and variables. The *generated by* clause states that all values of sort C can be represented by terms $\{\}$ and *insert*. The operators listed in the *partitioned by* clause are sufficient to distinguish unequal set values. The *implies* section describes additional properties of sort C that follow from the assertions part. The theory of a trait is the set of all logical consequences of its assertions. It contains everything that logically follows from its assertions, but nothing else. All operators listed in the *converts* clause are defined for terms in their domains in the exceptions noted in the *exempting* clause.

The Larch language supports a family of *Interface Specification* languages. Each interface specification language is designed for a specific programming language. For instance Larch/C++ [GH93], the interface specification language for C++ programming language, can be used to specify how C++ class interfaces must be invoked by a client program. An implementation of a data type specification is thus linked to the interface specification. Based on LSL traits and Larch/C++ interface specifications Celer [AC95] and Protopsaltou [Pro96] have given black-box testing methods for data type implementation. The code generator implemented by Zhang [Zha00] uses a small Larch/Java interface specification language while mechanically generating Java code in TROMLAB framework. The test suit generation methods discussed in this thesis will assume that data type implementations have been tested by the above methods.

2.2.2 Second Tier - Timed Reactive Object Model

A TROM defines a *Generic Reactive Class* (GRC) parameterized with port types. A port type is an abstract interface defining the set of messages (events) that can be received or sent by an object of the class at a port of this type. The signature of a port-type P gives the set of events that can occur at the port-type P, denoted by $\mathcal{E}^P$,

```
Set(E, C) : trait
    % Essential finite-set operators
    includes Integer
    introduces
        {} :→ C
        insert : E, C → C
        __ ∈ __ : E, C → Bool
        delete : E, C → C
    asserts
        C generated by {}, insert
        C partitioned by ∈
        ∀ s : C, e, e1, e2 : E
            ¬(e ∈ {})
            e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s
            delete(e1, insert(e2, s)) == if (e1 = e2) then s
                else insert(e2, delete(e1, s))
    implies
        ∀ e, e1, e2 : E, s : C
            insert(e, s) ≠ {}
            insert(e, insert(e, s)) == insert(e, s)
            insert(e1, insert(e2, s)) ==
                insert(e2, insert(e1, s))
        converts ∈, delete
            exempting ∀ x : E
            delete(x, {})
```

Figure 4: LSL Trait for Set

$\mathcal{E}^P = \mathcal{E}^P_{in} \bigcup \mathcal{E}^P_{out}$, where $\mathcal{E}^P_{in}$ is the set of input events, $\mathcal{E}^P_{out}$ is the set of output events, and $\mathcal{E}^P_{in} \bigcap \mathcal{E}^P_{out} = \emptyset$. Message exchange can occur only between compatible ports. Two port type P and Q are *compatible* if

- $e? \in \mathcal{E}^P_{in} \Leftrightarrow e! \in \mathcal{E}^Q_{out}$
- $e! \in \mathcal{E}^P_{out} \Leftrightarrow e? \in \mathcal{E}^Q_{in}$

A GRC may include attributes of two kinds: (1) abstract data types imported from the first tier, and (2) port types. The attribute functions define the association of attributes to states. For each state, the attribute function defines a subset of attributes that are active in the state. For a computation associated with a transition entering a state, only the attributes associated with that state are modified and all other attributes will be read-only in that computation.

The semantic model of reactive objects belonging to a GRC is a timed labeled transition system, extended with hierarchical states, logical assertions involving clock variables and variables of included traits. A transition between two states is specified by a guard and an action. A guard is a conjunction of two predicates, one specifying the port-condition for the event labeling the transition, and the other specifying the enabling condition on the attributes in the pre-state of the transition. An action is specified as a predicate involving the attributes in the post-state of the transition. A clock may be initialized as part of an action, if the event labeling the transition triggers a future action. The guard of a constrained transition will include a predicate involving clock variables and time constants.

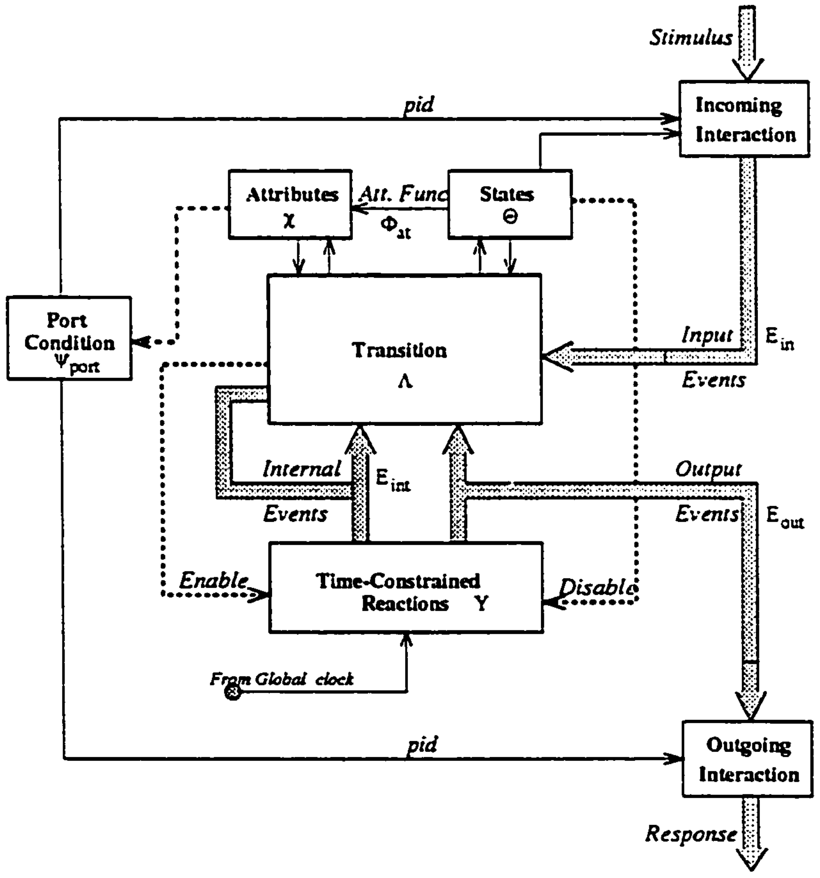


Figure 5: Anatomy of a Reactive Object

Figure 5 shows the behavior of a reactive object. The filled arrows in the figure indicate flow of events. An input event (?) is the result of an incoming interaction defined by the external stimulus, the current state of the object, and the port constrained by the port-condition. Every event occurrence causes a state transition and may also involve a computation. A computation updates the object's state and attributes, shown by the arrow labeled 'Att. Func.'. The dotted arrow connecting the block of computation to that of time-constrained reaction signifies the enabling of a reaction due to a computation. Based on the reading of the clock, an outstanding reaction is fired by the object, thereby generating either an internal event or an external event. All generated output events (!) will result as a response at the port specified by the port-condition. A state update may also result in the disabling of an outstanding reaction. A formal definition of TROM and its semantics appear in [Mut00]. Figure 6 shows the template for a generic reactive class specification.

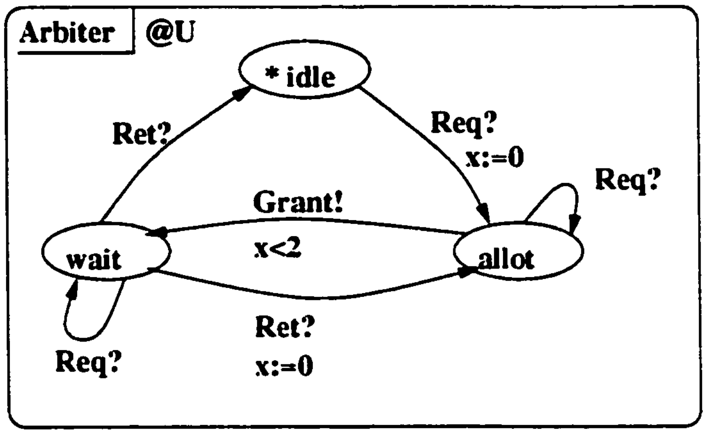
```
Class < name >
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 6: Template for Class Specification

Figure 7 is an example of the formal specification of the TROM class *Arbiter*. An arbiter allocates shared resources to processes requesting them. It enqueues the requests for a resource received from processes and allocates the resource to the next process waiting in the queue. The specification uses the functions *insert* and *tail* from the trait *Queue*($@U$, *UQueue*), imported by the *Arbiter* class, to add and delete requests made at a port of type $@U$. The attribute *hold* in class *Arbiter* denotes the most recent port at which the resource was granted. The output event *Grant*! is time constrained and must occur within 2 time units from the instant the input events *Req*? and *Ret*? have occurred.

The algorithms for testing the correctness of a class implementation are based on the semantics of TROM formalism. The two important requirements for conformance are the following:

- Every state in the design specification must be observable in the implementation: the attribute values and outstanding reactions in an implemented program



Class *Arbiter* [@U]

Events: *Req?U, Grant!U, Ret?U*

States: **idle, allot, wait*

Attributes: *rqQueue: UQueue; hold:@U*

Traits: *Queue[@U, UQueue]*

Attribute-function:

allot $\mapsto$ *rqQueue; wait* $\mapsto$ *rqQueue, hold*;

Transition-Specifications:

R_1 : $\langle$*idle, allot*$\rangle$; *Req?(true)*;

true $\Longrightarrow$ *rqQueue'* = *insert*(pid, {});

R_2 : $\langle$*allot, wait*$\rangle$; *Grant!*(pid $\in$ *rqQueue*);

true $\Longrightarrow$ *rqQueue'* = *tail(reQueue))* $\wedge$ *(hold'* = pid);

R_3 : $\langle$*allot, allot*$\rangle$, $\langle$*wait, wait*$\rangle$; *Req?(not* pid $\in$ *rqQueue)*;

true $\Longrightarrow$ *rqQueue'* = *insert*(pid, *rqQueue*);

R_4 : $\langle$*wait, allot*$\rangle$; *Ret?* (pid *hold*);

‾ *isEmpty(rqQueue)* $\Longrightarrow$ *equal(rqQueue', rqQueue)*;

R_5 : $\langle$*wait, idle*$\rangle$; *Ret?* (pid = *hold*);

isEmpty(rqQueue) $\Longrightarrow$ *true*;

Time-Constraints:

TC_1 : (R_1, *Grant*, [0,2], $\emptyset$)

TC_2 : (R_4, *Grant*, [0,2], $\emptyset$)

end

Figure 7: Formal Specification of Class Arbiter

must match the specification.

- Every transition in the state machine description of the design must have an implementation: the abstract computation specified in the action part must have a correct implementation.

2.2.3 Third Tier - System Configuration Specification

A system configuration is the collaboration of a finite number of objects instantiated from the second tier. Objects collaborate through message passing. Only compatible ports of collaborating objects can exchange messages. The specification template shown in Figure 8 has three sections:

```
Subsystem < name >
    Include:
    Instantiate:
    Configure:
end
```

Figure 8: Template for System Configuration Specification

Include Predefined subsystems are included in the Include section. Ports that are free in the included subsystems may be linked to ports of objects instantiated in the subsystem being defined.

Instantiate Reactive objects are instantiated from the GRCs defined in the second tier. An instantiation has the syntax $(c : X[@P : 3, @Q : 2])$. The effect of this declaration is the creation of object c having three ports p_1, p_2, p_3 of type $@P$, and two ports q_1, q_2 of type $@Q$ from class X.

Configure Object collaboration is specified by defining links between compatible ports of instantiated objects and objects in the included subsystem. The syntax for defining a communication link between the port p_1 (type P) of object c and the port s_1 (type S) of object a is $c.@p_1{:}@P <-> a.@s_2{:}@S$.

A formal operational semantics for system configurations is given in [Mut00]. The semantics is based on system traces, sequences of computational paths in the synchronous product of the state machines associated with the objects in the system configuration. Our test algorithms generate test cases that are valid partial traces, inject them in an implementation and check whether the state information and time constrained reactions are timely, as specified.

2.3 Case Study

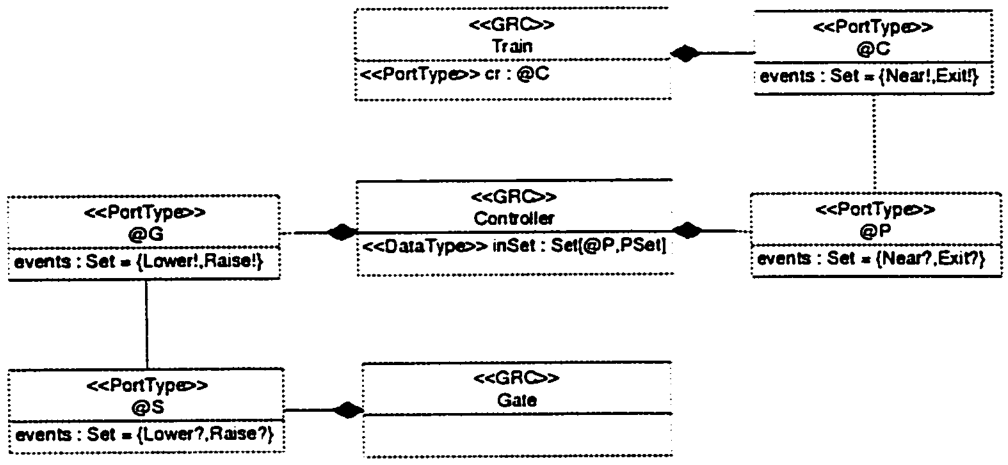
In this section, we introduce the railroad crossing problem, and will use it as a running example throughout this thesis to illustrate our testing methodology.

The railroad crossing problem has been considered as a bench mark example by researchers in real-time systems community [HM96]. A generalized version of this problem has been considered by Muthiayen and Alagar [MA99] to formally prove safety properties in their design. We take their verified design and generate test cases to test the correctness of code generated by Zhang [Zha00].

According to their design, several trains cross a gate independently and simultaneously using non-overlapping tracks. A train chooses the gate it intends to cross; there is a unique controller monitoring the operations of each gate. When a train approaches a gate, it sends a message to the corresponding controller, which then commands the gate to close. When the last train crossing a gate leaves the crossing, the controller commands the gate to open. The safe operation of the controller depends on the satisfaction of certain timing constraints, so that the gate is closed before a train enters the crossing, and the gate is opened after the last train leaves the crossing.

> A train enters the crossing within an interval of 2 to 4 time units after having indicated its presence to the controller. The train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message. The controller instructs the gate to close within 1 time unit of receiving an approaching message from the first train entering the crossing, and starts monitoring the gate. The controller continues to monitor the closed gate if it receives an approaching message from another train. The controller instructs the gate to open within 1 time unit of receiving a message from the last train to leave the crossing. The gate must close within 1 time unit of receiving instructions from the controller.

The gate must open within an interval of 1 to 2 time units of receiving instructions from the controller.



<<GRC>>
Train
<<PortType>> cr : @C
<<PortType>>
@C
events : Set = {Near!,Exit!}
<<GRC>>
Controller
<<DataType>> inSet : Set[@P,PSet]
<<PortType>>
@G
events : Set = {Lower!,Raise!}
<<PortType>>
@P
events : Set = {Near?,Exit?}
<<PortType>>
@S
events : Set = {Lower?,Raise?}
<<GRC>>
Gate

Figure 9: Class Diagram for Train, Gate, and Controller Entities

Figure 9 shows the class specification for the Train-Gate-Controller system. An input event is decorated with the suffix ? and an output event is decorated with the suffix !. The port types for reactive classes and the events associated with the port types are shown. The associations between the Port Type classes indicate communication channels between instances of the GRC classes.

Figures 11, 13, 15 show the statechart diagrams for the controller, train and gate. Together, they specify the behavior of a Train-Gate-Controller system. When there is no train in the system, the train is in the idle state, the controller is in the idle state, and the gate is in the opened state. When a train wants to pass through a crossing, it sends the message to the controller, the train and controller change their states simultaneously. In the state activate, the controller may receive "Near" messages from other trains or it sends the message "lower" to the gate. In the latter case, the controller and gate change their states simultaneously. In the state monitor, the controller continues to receive "Near" messages from other trains or monitor the gate. The train which is in the crossing may send an "Exit" message to the controller before exiting the gate. Both the controller and the train synchronize on the event "Exit". The train leaves the gate within 6 unit of time from the instant originally sends the

```
Class Controller [@P, @Y]
   Events: Lower!@Y, Near?@P, Raise!@Y, Exit?@P
   States: *idle, activate, deactivate, monitor
   Attributes: inSet:PSet
   Traits: Set[@P,PSet]
   Attribute-Function:
      activate →   {inSet}; deactivate →   {inSet};
      monitor →   {inSet}; idle →   {};
   Transition-Specifications:
      R1: <activate,monitor>; Lower(true);
         true => true;
      R2: <activate,activate>; Near(NOT(member(pid,inSet)));
         true => inSet' = insert(pid,inSet);
      R3: <deactivate,idle>; Raise(true);
         true => true;
      R4: <monitor,deactivate>; Exit(member(pid,inSet));
         size(inSet) = 1 => inSet' = delete(pid,inSet);
      R5: <monitor,monitor>; Exit(member(pid,inSet));
         size(inSet) > 1 => inSet' = delete(pid,inSet);
      R6: <monitor,monitor>; Near(!(member(pid,inSet)));
         true => inSet' = insert(pid,inSet);
      R7: <idle,activate>; Near(true);
         true => inSet' = insert(pid,inSet);
   Time-Constraints:
      TCvar1: R7, Lower, [0, 1], {};
      TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 10: Formal Specification for GRC Controller

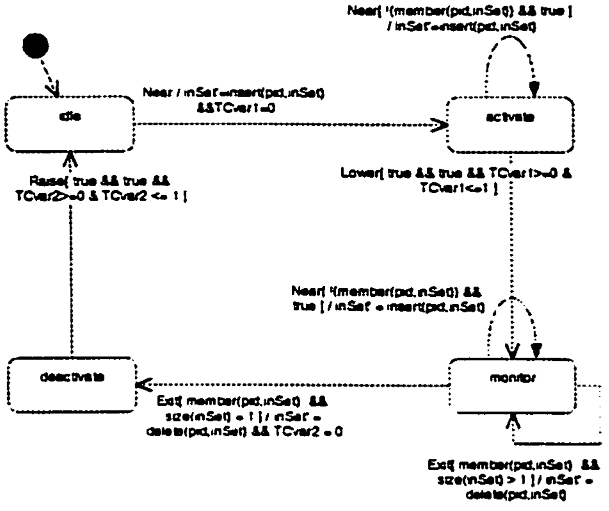

Near[!(member(pid,inSet)) && true] / inSet'=insert(pid,inSet)
idle
Near / inSet'=insert(pid,inSet) &&TCvar1=0
activate
Raise[true && true && TCvar2>=0 & TCvar2 <= 1]
Lower[true && true && TCvar1>=0 & TCvar1<=1]
Near[!(member(pid,inSet)) && true] / inSet' = insert(pid,inSet)
deactivate
monitor
Exit[member(pid,inSet) && size(inSet) = 1] / inSet' = delete(pid,inSet) && TCvar2 = 0
Exit[member(pid,inSet) && size(inSet) > 1] / inSet' = delete(pid,inSet)

Figure 11: Statechart Diagram for Controller

message "Near" to the controller. The controller changes from monitor state only when all the train have exited from the gate. At that instant, it sends the message to the gate and returns to the idle state. The events "Down" and "Up" are time constrained internal events for the gate. The evaluation for the local clocks for the controller are the variables "TCvar1" and "TCvar2". Similarly, the timed constrained events for the controller and train are governed by their local clocks shown in the extended statechart diagrams. Figures 10, 12, 14 show the textual specifications corresponding to their extended statechart diagrams.

Figure 17 shows the collaboration diagram for a system with 5 Trains, 2 Controllers, and 2 Gates. In this configuration, the object *train3* is allowed to interact with both controllers, while the other train objects can only interact with one of the controllers. Hence the object *train3* has two ports, one for each controller with which it will communicate. Each of the other trains have only one port. Each gate communicates with its associated controller through one port. Figure 16 shows the textual specification of the Train-Gate-Controller system shown in Figure 17.

```
Class Train [@R]
   Events: Near!@R, Out, Exit!@R, In
   States: *idle, cross, leave, toCross
   Attributes: cr:@C
   Traits:
   Attribute–Function:
      idle →   {}; cross →   {};
      leave →   {}; toCross →   {cr};
   Transition–Specifications:
      R1: <idle,toCross>; Near(true); true=>cr′=pid;
      R2: <cross,leave>; Out(true); true => true;
      R3: <leave,idle>; Exit(pid = cr); true => true;
      R4: <toCross,cross>; In(true); true => true;
   Time–Constraints:
      TCvar2: R1, Exit, [0, 6], {};
      TCvar1: R1, In, [2, 4], {};
end
```

Figure 12: Formal Specification for GRC Train

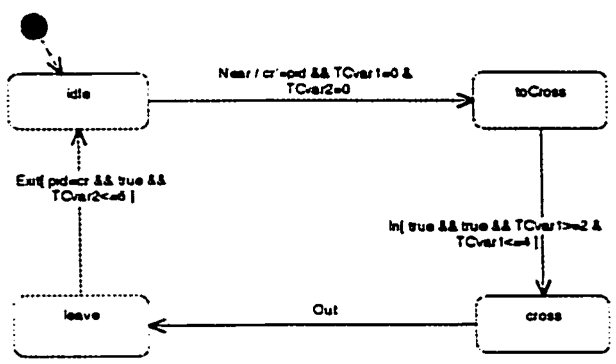

idle
toCross
Near / cr'=pid && TCvar1=0 & TCvar2=0
Exit[pid=cr && true && TCvar2<=6]
In[true && true && TCvar1>=2 & TCvar1<=4]
leave
Out
cross

Figure 13: Statechart Diagram for Train

```
Class Gate [@S]
   Events: Lower?@S, Down, Up, Raise?@S
   States: *opened, toClose, toOpen, closed
   Attributes:
   Traits:
   Attribute–Function:
      opened →   {}; toClose →   {};
      toOpen →   {}; closed →   {};
   Transition–Specifications:
      R1: <opened,toClose>; Lower(true); true => true;
      R2: <toClose,closed>; Down(true); true => true;
      R3: <toOpen,opened>; Up(true); true => true;
      R4: <closed,toOpen>; Raise(true); true => true;
   Time–Constraints:
      TCvar1: R1, Down, [0, 1], {};
      TCvar2: R4, Up, [1, 2], {};
end
```

Figure 14: Formal Specification for GRC Gate

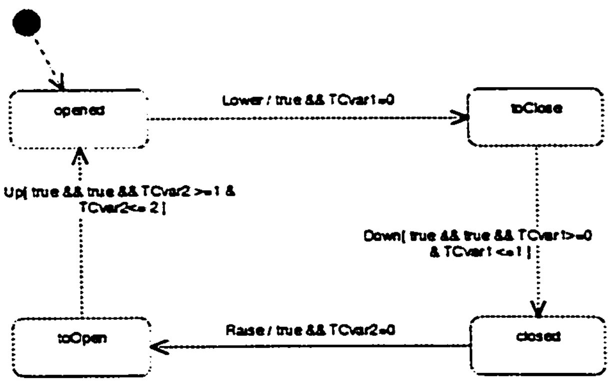

opened
Lower / true && TCvar1=0
toClose
Up[true && true && TCvar2 >=1 & TCvar2<= 2]
Down[true && true && TCvar1>=0 & TCvar1 <=1]
toOpen
Raise / true && TCvar2=0
closed

Figure 15: Statechart Diagram for Gate

```
SCS TrainGateController
  Include:
  Instantiate:
    gate1::Gate[@S:1];
    gate2::Gate[@S:1];
    train1::Train[@C:1];
    train2::Train[@C:1];
    train3::Train[@C:2];
    train4::Train[@C:1];
    train5::Train[@C:1];
    controller1::Controller[@P:3, @G:1];
    controller2::Controller[@P:3, @G:1];
  Configure:
    controller1.@G1:@G <-> gate1.@S1:@S;
    controller2.@G2:@G <-> gate2.@S2:@S;
    controller1.@P1:@P <-> train1.@C1:@C;
    controller1.@P2:@P <-> train2.@C2:@C;
    controller1.@P3:@P <-> train3.@C3:@C;
    controller2.@P4:@P <-> train3.@C4:@C;
    controller2.@P5:@P <-> train4.@C5:@C;
    controller2.@P6:@P <-> train5.@C6:@C;
end
```

Figure 16: Formal Specification for Train-Gate-Controller Subsystem

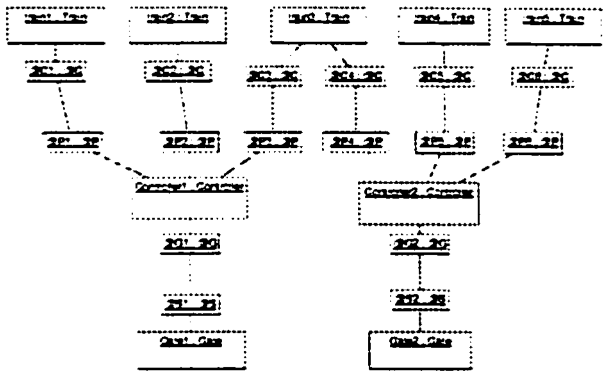


Figure 17: Collaboration Diagram for 5 Train - 2 Gate - 2 Controller

Chapter 3

Formal Basis of Testing UML Models of Real-Time Reactive Systems

3.1 Introduction

UML is widely used in industries to model untimed object-oriented systems. Selic [Sel99] has pointed out that UML has sufficient expressive power to model real-time systems, without requiring additional language constraints. Douglass [Dou98] has given a naive introduction to real-time modeling in UML. However, it is stated by Selic [Sel99] that *Time*, although defined in UML standards, is not given any semantics. Without semantics, time-constrained actions cannot be specified precisely. Given the general purpose modeling features of UML and the lack of formal semantics, it is possible to have several interpretations of one UML model even for untimed systems, thus giving rise to different implementations with different behaviors. Consequently, it is impossible to claim a conformance relation between a given UML model and its implementation. The goal of this chapter is to review RTUML, the language introduced by Muthiayen [Mut00] which has a formal syntax within UML language and formal semantics based on TROM formalism, and claim that our testing methodology is applicable to real-time UML models having TROM semantics.

3.2 RTUML Types and Domains

The basic types are GRCTypes, DataTypes, and PortTypes.

- $\mathcal{GRC}$ is the universal set of GRCTypes.

- $\mathcal{D}$ is the universal set of DataTypes.

- $\mathcal{P}$ is the universal set of PortTypes.

- P_0 is the *null PortType*. $\mathcal{P}$ includes the *null PortType* P_0, $P_0 \in \mathcal{P}$. P_0 is a *singleton* set, containing the *null port* p_0, $P_0 = \{p_0\}$.

The *Event Domain* formally defines the event set. Let $\mathcal{E}$ be the universal set of events, $\mathcal{E}_{internal}$ be the universal set of *internal* events, $\mathcal{E}_{external}$ be the universal set of *external* events, $\mathcal{E}_{input}$ be the universal set of *input* events, and $\mathcal{E}_{output}$ be the universal set of *output* events. An *input* event corresponds to an *external* event suffixed with the symbol "?". We use "e?" to denote the *input* event obtained by suffixing the *external* event "e".

$$\mathcal{E}_{input} = \{e? \mid e \in \mathcal{E}_{external}\}$$

An *output* event corresponds to an *external* event suffixed with the symbol "!". We use "e!" to denote the *output* event obtained by suffixing the *external* event "e".

$$\mathcal{E}_{output} = \{e! \mid e \in \mathcal{E}_{external}\}$$

The following properties apply to the universal sets of *events*.

1. The universal sets of *internal* and *external* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{external} = \emptyset$$

2. The universal sets of *internal* and *input* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{input} = \emptyset$$

3. The universal sets of *internal* and *output* events are disjoint.

$$\mathcal{E}_{internal} \cap \mathcal{E}_{output} = \emptyset$$

4. The universal sets of *input* and *output* events are disjoint.

$$\mathcal{E}_{input} \cap \mathcal{E}_{output} = \emptyset$$

5. The universal set of events $\mathcal{E}$ corresponds to the union of the universal sets of *internal*, *input* and *output* events.

$$\mathcal{E} = \mathcal{E}_{internal} \cup \mathcal{E}_{input} \cup \mathcal{E}_{output}$$

The following functions are defined on the *Event* domain.

- The bijective function σ_{in} converts a set of *input* events into a set of *external* events, by removing the suffix "?" from each *input* event.

  $$\sigma_{in} : \mathbb{P}(\mathcal{E}_{input}) \rightarrow \mathbb{P}(\mathcal{E}_{external})$$

  The function σ_{in} is defined as follows:

  $$\forall \ E_{in} \in \mathbb{P}(\mathcal{E}_{input}) \bullet \sigma_{in}(E_{in}) = \{e \ \mid \ e? \in E_{in}\}$$

- The bijective function σ_{out} converts a set of *output* events into a set of *external* events, by removing the suffix "!" from each *output* event.

  $$\sigma_{out} : \mathbb{P}(\mathcal{E}_{output}) \rightarrow \mathbb{P}(\mathcal{E}_{external})$$

  The function σ_{out} is defined as follows:

  $$\forall \ E_{out} \in \mathbb{P}(\mathcal{E}_{output}) \bullet \sigma_{out}(E_{out}) = \{e \ \mid \ e! \in E_{out}\}$$

The *State Domain* defines the set of states and their properties. Let S be the universal set of states, S_{simple} be the universal set of *simple* states, and $S_{complex}$ be the universal set of *complex* states. The following properties apply to the universal sets of *states*.

1. The universal sets of *simple* and *complex* states are disjoint.

$$S_{simple} \cap S_{complex} = \emptyset$$

2. the universal set of states S corresponds to the union of the universal sets of *simple* and *complex* states.

$$S = S_{simple} \cup S_{complex}$$

The following functions are defined on the *State* domain.

* A *complex* state contains a set of *substates* that are either simple or complex. The function *substates* returns the set of substates of a state.

$$substates : S \rightarrow \mathbb{P}(S)$$

  1. The function *substates* returns an empty set when applied to a *simple* state.

$$\forall \ s \in S_{simple} \bullet substates(s) = \emptyset$$

  2. The set of *substates* of a *complex* state is nonempty.

$$\forall \ s \in S_{complex} \bullet substates(s) \neq \emptyset$$

* A complex state has a unique *initial* state, which is a simple state. The total function *initial* identifies the initial state of a complex state.

$$initial : S_{complex} \rightarrow S_{simple}$$

  1. The *initial* state of a complex state is a member of the *substates* of the *complex* state.

$$\forall \ s \in S_{complex} \bullet initial(s) \in substates(s)$$

  2. The set of *substates* of a complex state contains at least 2 elements.

$$\forall \ s \in S_{complex} \bullet substates(s) \setminus \{initial(s)\} \neq \emptyset$$

* The hierarchy function $\mathcal{H}$ returns the set of all states in the hierarchy of a state.

$$\mathcal{H} : S \rightarrow \mathbb{P}(S)$$

  1. The hierarchy function $\mathcal{H}$ is recursively defined as follows.

$$\mathcal{H}(s) = \{s\} \cup \bigcup_{s \in substates(s)} \mathcal{H}(s)$$

3.2.1 Time Semantics

Time is defined as the set of nonnegative real numbers.

$$Time = \{t \mid t \in \mathbb{R}^{\geq 0}\}$$

TimeConstant is defined as the set of nonnegative rationals.

$$TimeConstant = \{m \mid m \in \mathbb{Q}^{\geq 0}\}$$

Clock Valuation

Time is fully encapsulated in a class. Clocks are locally defined and clock variables assume time values as measuring clock times.

- *Clock Valuation* v for a set C of clocks is a map that at any instant assigns a nonnegative real value to each clock.

  $$v : C \rightarrow (\mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0})$$

  That is: $v(C) : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$, $v(c)(x)$ is the valuation for a clock c, $c \in C$ at instance x, $x \in \mathbb{R}^{\geq 0}$

- For each $x \in \mathbb{R}^{\geq 0}$, and $k \in \mathbb{Z}$, we define $k + x$ to be a k-shift of x.

- For any valuation v and for all clocks $c \in C$, the clock valuation has the following properties:

  - $v(c)(0) = 0$
  - $v(c)(x) > v(c)(y)$, *if* $x > y$, $x, y \in \mathbb{R}^{\geq 0}$
  - $v(c)(k + x) = k + v(c)(x)$.

A time constraint is a predicate, involving clock valuation. If m is a time constant, $m \in \mathbb{Q}^{\geq 0}$, then $v(c)(x) \leq m$, $v(c)(x) \geq m$ are time constraints, where $c \in C$, $x \in \mathbb{R}^{\geq 0}$. A time constraint may also involve conjunction, disjunction and negation. For instance, if m_1, m_2 are time constants, then $v(c)(x) \geq m_1 \wedge v(c)(x) \leq m_2$ is a time constraint.

3.2.2 Reactive Object Model in RTUML

The reactive object model is one of GRCTypes, consisting of a generic reactive class with an associated statechart extended to capture timing constraints on transitions. This definition captures our notion of what a reactive object is.

Definition for *GRCClass*

A *GRCClass* is a 3-tuple $< \mathcal{P}, \mathcal{X}, \Omega >$, with the following definition:

- $\mathcal{P}$ is a set of *PortTypes*

$$\mathcal{P} : \mathbb{P}(\mathcal{P})$$

- $\mathcal{X}$ is a set of *attributes*, where each attributes is either an instance of a PortType from the set $\mathcal{P}$, or an instance of a DataType from the universal set $\mathcal{D}$.

$$\mathcal{X} = \{x \mid x : \mathcal{P}\} \cup \{a \mid a : \mathcal{D}\}$$

- Ω is a mapping from the set of PortTypes $\mathcal{P}$ to the power set of the universal set of events $\mathcal{E}$, defining the set of *allowed* input and output events at instances of the PortType.

$$\Omega : \mathcal{P} \rightarrow \mathbb{P}(\mathcal{E})$$

The following properties apply to a *GRC* class $< \mathcal{P}, \mathcal{X}, \Omega >$.

1. The sets of *input* and *output* events associated with a PortType, other than the *null PortType* P_0, correspond to disjoint sets of *external* events.

   $\forall P_i \in \mathcal{P} \bullet P_i \neq P_0 \rightarrow$

   $\Omega(P_i) = E_{in_i} \cup E_{out_i} \wedge E_{in_i} \subseteq \mathcal{E}_{input} \wedge E_{out_i} \subseteq \mathcal{E}_{output} \wedge$

   $\sigma_{in}(E_{in_i}) \cap \sigma_{out}(E_{out_i}) = \emptyset$

2. The sets of events associated with two PortTypes of a GRC class are disjoint.

$$\forall P_i, P_j \in \mathcal{P} \bullet P_i \neq P_j \rightarrow \Omega(P_i) \cap \Omega(P_j) = \emptyset$$

3. The events associated with the *null PortType* P_0 are *internal* events.

$$\forall P_i \in P \bullet P_i = P_0 \rightarrow \Omega(P_i) \subseteq \mathcal{E}_{internal}$$

4. The set of *internal* events $\mathcal{E}_{internal}$ of a GRC class corresponds to the set of events associated with the *null PortType* P_0.

$$P_0 \in \mathcal{P} \rightarrow \mathcal{E}_{internal} = \Omega(P_0) \wedge P_0 \notin \mathcal{P} \rightarrow \mathcal{E}_{internal} = \emptyset$$

5. The events associated with a PortType P_i of a GRC class, excluding the *null PortType* P_0, are *input* and *output* events.

$$\forall \ P_i \in \mathcal{P} \bullet P_i \neq P_0 \rightarrow \Omega(P_i) \subseteq (\mathcal{E}_{input} \cup \mathcal{E}_{output})$$

6. The set of *input* and *output* events E_{io} of a GRCClass corresponds to the distributed union of the sets of events associated with the PortTypes of the GRCClass, excluding the *null PortType* P_0.

$$E_{io} = \bigcup_{P_i \in \mathcal{P} \wedge P_i \neq P_0} \Omega(P_i)$$

Compatibility of PortTypes

Given two *GRCClasses* G_a and G_b, such that
$G_a =< \mathcal{P}_a, \mathcal{X}_a, \Omega_a >$, and
$G_b =< \mathcal{P}_b, \mathcal{X}_b, \Omega_b >$,

PortType P_{a_i} from the set $\mathcal{P}_a$ and PortType P_{b_j} from the set $\mathcal{P}_b$ are *compatible* iff the set of *input* events associated with PortType P_{a_i} is equal to the set of *output* events associated with PortType P_{b_j}, and the set of *output* events associated with PortType P_{a_i} is equal to the set of *input* events associated with PortType P_{b_j}. The comparison for equality events to the *external* events corresponding to the respective *input* and *output* events.

- The predicate *compatible* defines the compatibility of two PortTypes.

$$compatible : \mathcal{P} \times \mathcal{P} \rightarrow Bool$$

1. The predicate *compatible* has the following definition.

$$\forall \ P_{a_i} \in \mathcal{P}_a, P_{b_i} \in \mathcal{P}_b \ \bullet \ compatible(P_{a_i}, P_{b_j}) \leftrightarrow$$
$$\Omega_a(P_{a_i}) = E_{in_a} \cup E_{out_a} \wedge E_{in_a} \subseteq \mathcal{E}_{input} \wedge E_{out_a} \subseteq \mathcal{E}_{output} \wedge$$
$$\Omega_b(P_{b_j}) = E_{in_b} \cup E_{out_b} \wedge E_{in_b} \subseteq \mathcal{E}_{input} \wedge E_{out_b} \subseteq \mathcal{E}_{output} \wedge$$
$$\sigma_{in}(E_{in_a}) = \sigma_{out}(E_{out_b}) \wedge \sigma_{in}(E_{in_b}) = \sigma_{out}(E_{out_a})$$

Definition for *Extended StateChart*

An *ExtendedStatechart* is a 8-tuple $< S, C, R, \Upsilon, \Psi, \Gamma, \Xi >$, with the following definition.

- S is a set of states.

$$S : \mathbb{P}(S)$$

  1. The set S consists of *simple* and *complex* states. The set S_s denotes the subset of all simple states from the set S, and the set S_c denote the subset of all complex states from the set S.

$$S = S_s \cup S_c \wedge S_s \subseteq S_{simple} \wedge S_c \subseteq S_{complex}$$

  2. The distinguished simple state s_0 from the set S is the *initial* state of the statechart.

$$s_0 \in S_s$$

  3. The *initial* state of a *complex* state from the set S_c is a *simple* state from the set S_s.

$$\forall s_c \in S_c \bullet initial(s_c) \in S_s$$

.

  4. The set of *substates* of a complex state from the set S_c is a subset of the set S.

$$\forall s_c \in S_c \bullet substates(s_c) \subset S$$

  5. The sets of substates of two complex states from the set S_c are disjoint.

$$\forall s_i, s_j \in S_c \bullet s_i \neq s_j \rightarrow substates(s_i) \cap substates(s_j) = \emptyset$$

- C is a set of *logical clocks* for defining *time constraints* on reactions to transitions.

- R is a set of *transitions*, where each transition is a 5-tuple $< s, d, e, g, a >$, such that

  - s is the *source* state; s is a member of the set S.

$$s \in S$$

- d is the *destination* state; d is a member of the set S.

$$d \in S$$

- e is the *trigger* event; e is a member of the universal set of events $\mathcal{E}$

$$e \in \mathcal{E}$$

- g is a predicate representing the *guard* condition for the transition.
- a is a predicate representing the *action* resulting from the transition.

1. A valid transition satisfies one of the following four conditions.

   * The destination state d is a state which is not a substate of another state.

   $$d \in S \setminus \bigcup_{s_c \in S_c} substates(s_c)$$

   * The destination state d is the *initial* state of a complex state which is not a substate of another state.

   $$\exists\, d_c \in S_c \setminus \bigcup_{s_c \in S_c} substates(s_c) \bullet d = initial(d_c)$$

   * There exists a complex state s_c such that the source state s is in the hierarchy of the state s_c and the destination state d is a substate of the state s_c.

   $$\exists\, s_c \in S_c \bullet s \in \mathcal{H}(s_c) \wedge d \in substates(s_c)$$

   * There exists a complex state s_c such that the source state s is in the hierarchy of the state s_c and the destination state d is the *initial* state of a complex substate of the state s_c.

   $$\exists\, s_c, d_c \in S_c \bullet s \in \mathcal{H}(s_c) \wedge d_c \in substates(s_c) \wedge d = initial(d_c)$$

2. If the source state s is a complex state, then the transition corresponds to a set of transition R_s such that the source state s_h, of each transition in the set R_s, is in the hierarchy of the state s.

$$s \in S_c \rightarrow \{< s_h, d, e, g, a >|\ s_h \in \mathcal{H}(s)\} \subseteq R$$

3. If the destination state d is a complex state, then the transition corresponds to a transition whose destination state d_e is the initial state of the state d.

$$d \in S_c \rightarrow \{< s, d_e, e, g, a >| \ d_e \in initial(d)\} \subseteq R$$

- Υ is a total function that maps a clock from the set C to a transition from the set R, such that the clock is initialized to 0 when the transition occurs.

$$\Upsilon : \ C \rightarrow R$$

If $\Upsilon(c) = r$, then $v(c)(0) = 0$ for the transition r.

- Φ is a function that maps a transition from the set R to a set of clocks from the set C, such that the timing constraint on the transition is specified in terms of the values of these clocks.

$$\Phi : \ R \rightarrow \mathbb{P}(C)$$

1. If the trigger event e of the transition $r =< s, d, e, g, a >$ is an *input* event, then the transition r is not time-constrained.

$$\forall \ r \in R \bullet r =< s, d, e, g, a > \wedge \ e \in \mathcal{E}_{input} \rightarrow \Phi(r) = \emptyset$$

2. If the transition $r =< s, d, e, g, a >$ is time-constrained, then the trigger event e is either an *internal* or an *output* event.

$$\forall \ r \in R \bullet r =< s, d, e, g, a > \ \wedge \ \Phi(r) \neq \emptyset \rightarrow e \in (\mathcal{E}_{internal} \cup \mathcal{E}_{output})$$

3. If the transition $r =< s, d, e, g, a >$ is time-constrained, then the trigger event e corresponds to a reaction to transitions other than r from the set R, such that the corresponding clocks are initialized to 0 when those transition occur.

$$\forall \ r \in R \bullet \Phi(r) \neq \emptyset \rightarrow \forall \ c \in \Phi(r) \bullet \exists \ r_2 \in R \bullet \Upsilon(c) = r_2 \wedge r_2 \neq r \wedge v(c)(0) = 0$$

- Ψ is a partial function that gives the value of a clock from the set C when a transition from the set R occurs.

$$\Psi : \ C \times R \rightarrow Time$$

* The function Ψ is *defined* on the clock c and the transition r iff
  * the clock c is initialized to 0 when the transition r occurs, or
  * the transition r corresponds to a time-constrained reaction, such that the timing constraint is defined in terms of the clock c, or
  * the destination state of the transition r is a *disabling state* for a time-constrained reaction, such that the timing constraint is defined in terms of the clock c.

$$\Psi(c,r) = \begin{cases} 0, & \text{if } \Upsilon(c) = r, \\ v(c)(x),\ \textit{where}\ 0 < v(c)(x) < \infty, & \text{if } c \in \Phi(r),\ x \in \mathbb{R}^{\geq 0} \\ \infty, & \text{if } r =< s,d,e,g,a > \wedge\ d \in \Xi(c) \end{cases}$$

- Γ is a partial function that gives the *lower* and *upper bounds* for the value of a clock from the set C, relative to the activation time of the time-constrained reaction, within which the corresponding transition from the set R can occur.

$$\Gamma :\ C \times R \rightarrow \textit{TimeConstant} \times \textit{TimeConstant}$$

  * The function Ψ is *defined* on the clock c and the transition r iff the transition r corresponds to a time-constrained reaction, such that the timing constraint is defined in terms of the clock c.

$$\Gamma(c,r) =< l,u > \quad \textit{iff} \quad c \in \Phi(r) \wedge (l \leq v(c)(x) \leq u)$$

  where $x \in \mathbb{R}^{\geq 0}$

- Ξ is a function that maps a clock c from the set C to the set of *disabling* states for the time-constrained reaction defined in terms of the clock c.

$$\Xi :\ C \rightarrow \mathbb{P}(s)$$

Definition for *GRCType*

A *GRCType* is a 2-tuple $< G, B >$, with the following definition.

- G is a *GRCClass*.

$$G =< \mathcal{P}, \mathcal{X}, \Omega >,$$

- B is an *Extended Statechart.*

$$B =< S, C, R, \Upsilon, \Phi, \Psi, \Gamma, \Xi >$$

- A transition r from the set R, such that $r =< s, d, e, g, a >$, satisfies the following properties.

  * The *trigger* event e is associated with a PortType from the set of PortTypes P.

$$\exists P_i \in \mathcal{P} \bullet e \in \Omega(P_i)$$

  * The *guard* condition g is a *conjunction* of three predicates $g_{port\_cond}, g_{enabling\_cond}$ and $g_{time\_constraint}$.

$$g = g_{port\_cond} \wedge g_{enabling\_cond} \wedge g_{time\_constraint}$$

  * The *port condition* $g_{enabling\_cond}$ is a logical assertion on the values of *attributes* from the set $\mathcal{X}$ and an instance p_j of a PortType P_i from the set P. The PortType P_i corresponds to the PortType with which event e is associated; the *null PortType* P_0 for internal events. The instance p_j of PortType P_i corresponds to the port through which the message is channeled; the *null port* p_0 for internal events.

$$g_{port\_cond} : \mathbb{P}(\mathcal{X}) \times P_i \rightarrow Bool$$

  where

$$P_i \in \mathcal{P} \wedge e \in \Omega(P_i)$$

  1. If the trigger event e an *internal* event, the *port condition* is *true.*

$$if\ e \in \mathcal{E}_{internal},\ then\ g_{port\_cond} \triangleq true$$

  - The *enabling condition* $g_{enabling\_cond}$ is a logical assertion on the values of *attributes* from the set $\mathcal{X}$.

$$g_{enabling\_cond} : \mathbb{P}(\mathcal{X}) \rightarrow Bool$$

- The *time constraint condition* $g_{time\_constraint}$ is a logical assertion on the values of *clocks* from the set C, that are used in specifying the timing constraint on transition r.

$$g_{time\_constraint} : \mathbb{P}(C) \rightarrow Bool$$

For instance, if $x \in \delta$, $g_{time\_constraint}(x) = true \mid false$

2. If there is no timing constraint r, the *time constraint condition* is *true.*

$$if\ \Phi(r) = \emptyset\ then\ g_{time\_constraint} \triangleq true$$

3. If the trigger event e is an *input* event, the *time constraint* is *true.*

$$If\ e \in \mathcal{E}_{input}\ then\ g_{time\_constraint} \triangleq true$$

* The *action* a is a *conjunction* of two predicates a_{post_cond} and $a_{clock_{init}}$.

$$a = a_{post\_cond} \wedge a_{clock\_init}$$

- The *post condition* $a_{post_{cond}}$ is a logical assertion on the values of *attributes* from the set $\mathcal{X}$ after transition r is taken.

$$a_{post\_cond} : \mathbb{P}(\mathcal{X}) \rightarrow Bool$$

- The *clock initialization expression* a_{clock_init} initializes the value of a *clock* to 0 for each *time-constrained* reaction associated with transition r.

$$a_{clock\_init} \triangleq \bigwedge_{c \in C \wedge \Upsilon(c) = r} \Psi(c, r) = 0$$

- For each state s from the set S, the *disabling state expression* $e_{disabling\_state}$ defines the time-constrained reactions that are disabled when a transition leading to the state s occurs. The *disabling state expression* $e_{disabling\_state}$ is defined as a predicate on clocks from the set C that are used for specifying the timing constraints on the reactions.

$$e_{disabling\_state} : \mathbb{P}(C) \rightarrow Bool$$

* The *disabling state expression* $e_{disabling\_state}$ sets the values of the clocks defining timing constraints on reactions that are disabled in state s to infinity(∞).

$$e_{disabling\_state} \triangleq \bigwedge_{c \in C \wedge d \in \Xi(c) \wedge r \in R \wedge r = <s,d,e,g,a>} \Psi(c, r) = \infty$$

Operational Semantics

The *status* of a TROM at any time t, is the tuple $(s;\ \vec{a};\ \mathcal{R})$, where the current state θ is a simple state of the TROM, $\vec{a}$ is the assignment vector, and $\mathcal{R}$ is the vector of outstanding reactions. A *computational step* of a TROM occurs when the object with status $(s;\ \vec{a};\ \mathcal{R})$, receives a signal $\langle e, p_i, t\rangle$ from its environment and there exists a transition specification that can change the status of the TROM. A computation τ of a TROM object A is a sequence, possibly infinite, of alternating statuses and signals, $\mathcal{OS}_0 \overset{\langle e_0,p_0,t_0\rangle}{\rightarrow} \mathcal{OS}_1 \overset{\langle e_1,p_1,t_1\rangle}{\rightarrow} \ldots$

A computation of an extended statechart $B =< S, C, R, \Upsilon, \Psi, \Gamma, \Xi >$ can be written in terms of *extended states*, where an extended state is a pair $< s, v >$, $s \in S$, and v is a clock valuation at that state. For instance, let $\mathcal{OS}_i = (s_i;\ \vec{a_i};\ \mathcal{R}_i)$, $\mathcal{OS}_{i+i} = (s_{i+1};\ \vec{a_{i+1}};\ \mathcal{R}_{i+1})$ are two successive statues of a TROM. The states change due to the occurrence of the signal $< e_i, p_i, t_i >$ at the state s_i. If this TROM receive its next signal $< e_{i+1}, p_{i+1}, t_{i+1} >$ in the state s_{i+1}, then $v_{i+1} = v_i + t_{i+1} - t_i$. That is, the local clocks run at the same rate as the global clock. Hence, one step of computation can be written as $< s_i, v_i > \overset{<e_i,p_i,t_i>}{\rightarrow} < s_{i+1}, v_{i+1} >$.

3.2.3 Reactive System Model

The reactive system model consists of generic reactive classes each with an associated extended statechart, configurations of instances of reactive object models and scenarios of interaction among the reactive objects. This definition captures our notion of what a reactive system is.

Definition for *Configuration*

A *configuration* is a 4-tuple $< V, I, W, L >$, with the following definition.

- V is a set of *reactive objects.*

  1. A reactive object from the set V is an instance of a GRCType.

$$\forall\ v \in V \bullet \exists\, G \in \mathcal{GRC} \bullet v : G$$

- I is a set of *port objects.*

1. A port object from the set I is an instance of a PortType.

$$\forall\ p \in I \bullet \exists P_i \in \mathcal{P} \bullet p : P_i$$

- W is a function that defines a set of *port ownership* associations, identifying port objects from the set I that are owned by a reactive object from the set V.

$$W : V \rightarrow \mathbb{P}(I)$$

1. A port object from the set I is owned by a unique reactive object from the set V. The sets of port objects associated with two reactive objects are disjoint.

$$\forall\ r_i, r_j \in V \bullet r_i \neq r_j \rightarrow W(r_i) \cap W(r_j) = \emptyset$$

- L is a partial injective function that defines a set of *communication channels* between port objects from the set I.

$$L : I \rightarrow I$$

1. A port object from the set I is linked to at most one port object from the set I. The port objects associated with two port objects p_i and p_j are distinct port objects.

$$\forall\ p_i, p_j \in I \bullet p_i \neq p_j \rightarrow L(p_i) \neq L(p_j)$$

2. A *communication channel* associates two port objects only if they are instances of compatible PortTypes.

$$\forall\ p \in I \bullet p \in P_i \wedge L(p) \in P_j \rightarrow compatible(P_i, P_j)$$

Definition for *Scenario*

A *scenario* is a 2-tuple $< V, M >$, with the following definition.

- V is a set of reactive objects.

1. A reactive object from the set V is an instance of a GRCType.

$$\forall\ v \in V \bullet \exists G \in \mathcal{GRC} \bullet v : G$$

- M is a sequence of messages. A message is a 4-tuple $< v_s, v_r, e, t >$, such that

  1. the *sender* object v_s is a number of the set V

  $$v_s \in V$$

  2. the *receiver* object v_r is a member of the set V.

  $$v_r \in V$$

  3. the *event* e is a member of the universal set of events $\mathcal{E}$

  $$e \in \mathcal{E}$$

  4. the *time* t corresponds to the global time at which the event occurs.

  $$t \in Time$$

Definition for *Reactive System Model*

A Reactive System Model is a 3-tuple $< Y, F, N >$, with the following definition.

- Y is a set of *GRCTypes.*
- F is set of *configurations.*
- N is a set of *scenarios.*

The following properties apply to a *Reactive System Model.*

1. In a configuration $f =< V, I, W, L >$ from the set F, a reactive object v from the set V is an instance of a GRCType G from the set Y.

$$\forall < V, I, W, L > \in F \bullet \forall \; v \in V \, \exists \, G \in Y \bullet v : G$$

2. In a configuration $f =< V, I, W, L >$ from the set F, a port object p from the set I is an instance of a PortType P such that PortType P is owned by a GRCType G from the set Y.

$$\forall < V, I, W, L > \in F \bullet \forall \; p \in I \bullet \exists < \mathcal{P}, \mathcal{X}, \Omega > \in Y \bullet p : \mathcal{P}$$

3. In a scenario $n =< V, M >$ from the set N, a reactive object v from the set V is an instance of a GRCType G from the set Y.

$$\forall \ < V, M > \in N \bullet \forall v \in V \bullet \exists G \in Y \bullet v : G$$

4. In a scenario $n =< V, M >$ from the set N, for every message $m =< v_s, v_r, e, t >$ from the set M, there exists a configuration f from the set F, such that port object p_i and p_j are in the configuration, and there exists GRCTypes G_i and G_j from the set Y, such that PortType P_i is owned by GRCType G_i, PortType P_j is owned by GRCType G_j, port object p_i is an instance of PortType P_i, port object p_j is an instance of PortType P_j, event e is allowed at PortTypes P_i and P_j, and PortTypes P_i and P_j are *compatible*.

   $\forall \ < V, M > \in N \bullet \forall \ < v_s, v_r, e, t > \in M \bullet$

   $\exists \ < V, I, W, L > \in F, p_i, p_j \in I \ \bullet$

   $\exists < \mathcal{P}_a, \ \mathcal{X}_a, \Omega_a >, < \mathcal{P}_b, \mathcal{X}_b, \Omega_b > \in Y, P_i \in \mathcal{P}_a, P_j \in \mathcal{P}_b \ \bullet$

   $\exists \ e_{in} \in \Omega_a(P_i), \ e_{out} \in \Omega_b(P_j) \ \bullet$

   $p_i : P_i \wedge p_j : P_j \wedge e \in \sigma_{in}(\{e_{in}\}) \wedge e \in \sigma_{out}(\{e_{out}\}) \wedge compatible(P_i, P_j)$

3.3 Automated Testing of RTUML Models

Muthiayen [Mut00] gives a rigorous semantics for RTUML models, relating it to TROM semantics. Our testing methodology is founded on TROM semantic models, and hence is valid for RTUML models. Figure 18 shows how RTUML models are linked to testing process. Based on the RTUML models in analysis and design stages, test case generation can be developed in parallel with the implementation. Test execution will apply generated test cases to implementation and evaluate the test results. When a RTUML model changes, test cases must be altered to match the change. That is, we automatically regenerate test cases that would match the current RTUML model.

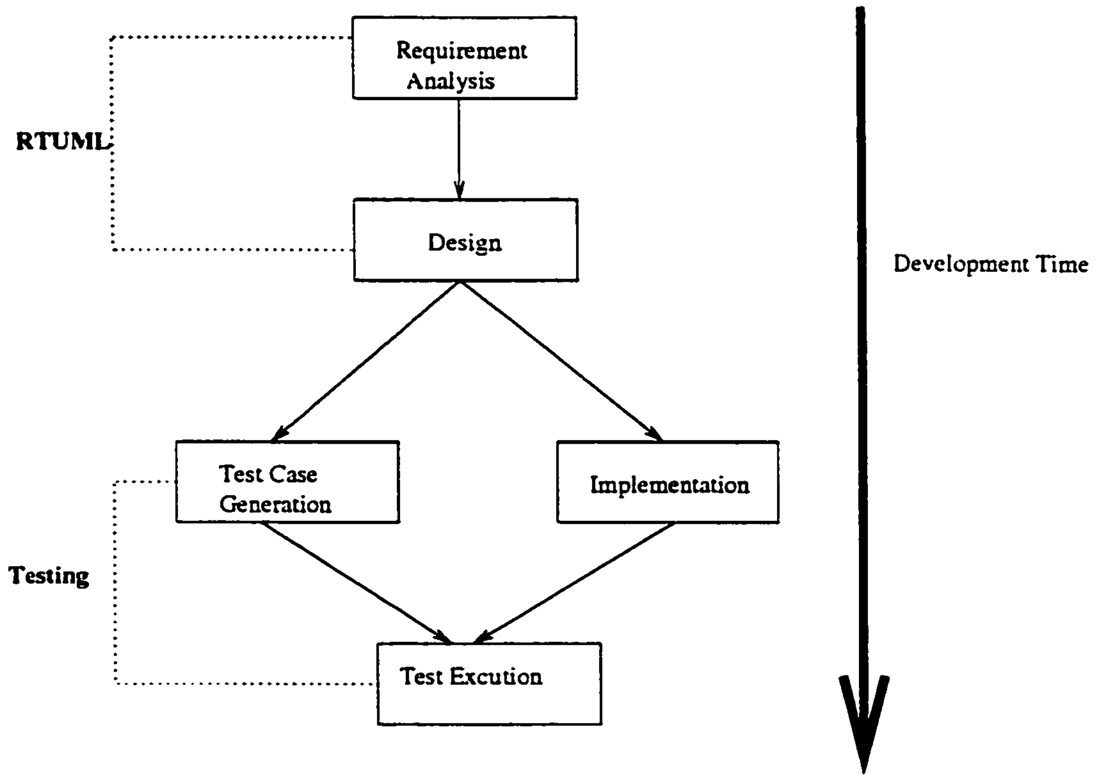


Figure 18: RTUML and Automated Testing

Chapter 4

Related Work

4.1 Introduction

In recent years concerted efforts have been made to elevate software testing from an intuitive *ad-hoc* collection of methods into a unified discipline of formalized and systematized techniques. Real-time reactive systems have been widely and increasing used throughout society, but testing methods to such systems have not been deeply studied. There is a need for a rigorous testing method, as well as automated testing method which promises to save a great deal of human effort so that testing is effective and repeatable. This especially applies to real-time reactive systems which have complex behaviors over time and require long test sequences.

There are a number of methods related to software testing, in particular to object-oriented testing. We will focus on those testing methodologies that are specific to real-time reactive systems, and those test suits generation based on formal specifications.

This chapter describes works related to specification-based testing.

4.2 Untimed System

Several rigorous methods exist [CTCC98, CRS96, Don97] for testing untimed systems in a black-box fashion. These methods differ in the choice of formalism used to specify the requirements, and the techniques used to generate test cases from the formal specification. Weyuker's test generation method [WGS94] uses boolean propositions to specify the requirements. Donat [Don97] has extended Weyuker's method by using predicate logic to state the requirements. Kirani [Kir94] has used BNF notation for the same purpose. These three notations lack the expressive power needed to formally specify the different states of abstract data types and formulate test cases based on them. Consequently, they are not suitable for testing large-scale software systems.

Algebraic specification methods are well suited for interface specification in object-oriented systems. However, a pure algebraic style does not support the specification of state information. Due to this limitation, a model based language, which can specify state changes, is often used as specification language for black-box testing. Rigorous methods exist to generate test cases from Z specification [SC96], VDM specification [DF93] and Larch specification [AC95] languages.

P. Stocks and D. Carrington [SC96] derive test templates from Z specifications and provide a test template framework for specification-based testing. The framework can be defined using any model-based specification notation and used to derive tests from model-based specifications. The framework formally defines test data sets and their relation to the operations in a specification and to other test data sets, providing structure to the testing process. In essence, an operation or functional unit under test is a relation. It represents some transformation of input values to output values. A test template is the basic unit for defining data and can be expressed by constraints over the input variables defined in the specification. The authors use a structured approach to build a hierarchy of test templates. Coarser templates are iteratively divided into smaller templates using test strategies. The valid input space is the starting point of a hierarchy. Once the valid input space of the functional unit is determined, the next step is to subdivide the valid input space into the desired subsets, or partitions, called *domains*. The terminal nodes in a hierarchy represent the final test classes as determined by the human tester. Choice of domains is not determined by the test templates framework. Rather, testing strategies and heuristics are used to subdivide the valid input space. Domains must be chosen so that each element of a domain has the same error-detecting ability, and so the result of testing one element of the domain applies to all elements of the domain. After applying all the desired strategies to derive test templates, the templates hierarchy is considered complete. Instances of the templates in the hierarchy represent test data. The methodology of Stocks and Carrington provides a general framework for specification-based testing, however applies only to untimed systems.

4.3 Timed System

Periyasamy and Alagar [PA00] extend Stocks and Carrington's method [SC96] to generate test cases from the composite operations in Object-Z for testing the conformance of an object-oriented program implementing the specified system. Timed extensions of Z [MH92] and Object-Z [MD98, PA01] specification languages have recently been introduced, but no method has yet been formulated to use them to generate test cases for timed systems.

In general, not much work has been done for rigorously testing timed systems. This is perhaps due to the lack of appropriate formal models that lend themselves to black-box testing. Recently, Springintveld, Vaandrager, and D'Argenio [SVD97] have proposed a black-box testing method for dense real-time systems. This approach is based on Timed Input/Output Automata(TIOA), which is an extension of the Timed Automata(TA). It discretizes the infinite state space to obtain a finite set of tests. The dependence between the length of the test sequence and the granularity of the grid automata is that the longer the sequence the finer the grid automata. The testing algorithm assumes the behavior of the timed system is accurately modeled by a TIOA *Impl*, an automaton that implements the TIOA specification, and the conformance testing is that the timed system conforms to the specification *Spec* if *Impl* is bisimilar to *Spec*. It is the first algorithm that yields a finite and complete set of tests for dense real-time systems. However, the complexity of the algorithm itself is highly exponential and cannot be claimed to be useful in practice. The testing method is not object-oriented, and modeling a timed system as a TIOA is restricted by the system's complexity and scalability.

The testing algorithm [SVD97] for TIOA provides the basic inspiration for the testing work on TROM, however there are significant differences in the semantics of the formalism TA, TIOA, and TROM. The testing methodology presented in this thesis is theoretically sound as well as practical. The complexity barrier is broken by modular testing of reactive units and reusing the test cases for system testing. Moreover, test cases for inherited classes are generated incrementally from the test cases of the parent class.

Timed (finite) Automata [AD94] model the behavior of real-time systems by annotating state-transition graphs with timing constraints using finite real-valued clocks. Timed automata provide a natural way of expressing timing delays of a real-time system. The entire system is specified as an automaton, the synchronous product composition of all the individual automaton within the system. The timed automaton for the system will invariably be huge and complex. It is impossible to construct such an automaton in practice. TA approach is not object oriented, and therefore, instances of one model cannot be instantiated within the formalism. For instance, in the railroad crossing problem [HM96], a bench mark case study in real-time system community, TA approach can model only a simple system with one train, one gate and one controller. That is, a generalized Train-Gate-Controller system in which an arbitrary number (finite) of trains are monitored by a controller cannot be modeled by TA formalism.

In TA, the time domain is non-negative real numbers. It is a natural model for physical processes operating over continuous time. Time constant, which is used in a time constraint for comparing a clock value, is the set of nonnegative rationals. The semantics of the transition specification $\langle q_1, q_2, e, c, \delta\rangle$, where q_1 is the source state, q_2 is the destination state, e is the labeling event, c is a clock, and δ is the time constraint is:

> the clock valuation v_2 of the clock c in state q_2 and the clock valuation v_1 of the clock c in state q_1 satisfy the relation $v_2 = v_1 + \epsilon$ and δ is true when the clock variable c is replaced by its evaluation v_2.

Here ϵ is the time taken for the transition from state q_1 to state q_2.

Timed I/O Automata (TIOA) are an extension of Timed Automata in which the events are partitioned into inputs and outputs. Each state has either (a) a single outgoing transition labeled with an output action or (b) both outgoing delay transition and outgoing input transitions (one for each input action), but no outgoing output transitions. In TIOA, time constant is a non-negative integer. For every admissible clock valuation, each state can be assigned an *invariant*, a predicate involving clock variables and integer constants. If the predicate δ constrains the transition between the states q_1 and q_2, only when the $inv(q_1) \wedge \delta$ is *true* the transition can occur. After the state change, $inv(q_2)$ becomes *true*.

A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi [EDKE98] address the issue of generating timed test cases from a timed system specification based on TIOA for real-time reactive systems. A state of a TIOA A is a pair (l, v) consisting of a location $l \in L_A$ and a clock valuation $v \in V(C_A)$. A clock valuation over a set of clocks C is a map v that assigns to each clock $x \in C_A$ a value in $\mathbf{R}^{+\infty}$. The semantic model of a TIOA A is given by a timed labeled transition system $S_t(A)$ that consists of the state set S_A, the label set $\mathbf{R}^{\geq 0} \cup \Sigma_A$, both input/output actions and time increments, and the transition relation $\xrightarrow{a}$, for $a \in \mathbf{R}^{\geq 0} \cup \Sigma_A$. Since the timed labeled transition system $S_t(A)$ is infinite, because of the infinite delay transitions, it cannot be used for test generation. The solution is therefore to reduce the number of states in the system. An equivalence relation on the set of clock valuations $V(C_A)$ is used to cluster equivalent states of $S_t(A)$ into equivalent classes. A partition of the uncountable state of the timed automaton produces a finite number of clock regions(a set of equivalent states in other words), called a *region* automaton. In order to generate test cases, a subautomaton of the region automaton, called the Grid Automaton, is derived from the region automaton. Since the region automaton is formed with locations of TIOA and clock regions, a set of clock valuations has the same behavior. The idea behind the construction of the grid automaton is to represent each clock region with a finite set of clock valuations, referred to as the representatives of the clock region. For this reason, they define the grid points with granularity $1/k$, where k is the number of clocks. In fact, the granularity of grid points constitutes the steps by which the clocks are authorized to pass from one clock region to another one, allowing thereby the automaton to make transitions from one location to another.

The TIOA models a reactive system is not object-oriented. So, the size of systems that can be modeled will be restricted and the complexity of the test case generation is quite high. Also no theoretical foundation for the construction of region automaton has been given.

A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaidi [CLRZ99] present a new algorithm, Hit-or-Jump, for embedded testing of components of communication system that can be modeled by communicating extended finite state machine. The new algorithm avoids the construction of a complete system reachability that would be exposed to the well-know state space explosion. Instead, it conducts a local search from the current state in a neighborhood of the reachability graph. If an untested part is found(a Hit), they test that part and continue the process from there. Otherwise, they move randomly to the frontier of the neighborhood searched(Jump), and continue the process from there. It is a new technique for system state search and constructs test sequences efficiently with a high fault coverage. Recently, A. Cavalli, C. Gervy and S. Prokopenko [CGP01] present two new approaches for passive testing using an Extended Finite State Machine(EFSM). The first approach extracts the invariants describing characteristic behaviors from specification EFSM, then studies the trace resulting from the implementation to determine if it conforms with those behaviors. The second approach expresses properties from specification EFSM in constraints, the trace from the implementation in automaton, then check if the states in the automaton satisfy the constraints. The new algorithm and approaches are more suitable for the communication networks where EFSM models both control flow and data flow of the systems.

4.4 Comparison With Our Work

TROM is an Object Oriented formalism introduced in [AAM98] for modeling real-time reactive systems. We discuss the test case generation method for this formalism. Although TROM semantics is based on timed labeled transition systems, there are significant differences between TROM and TA, and between TROM and TIOA.

In TROM formalism, a reactive entity, such as controller or arbiter, is formalized as a generic reactive class encapsulating the structure and time-dependent functionality of objects of the class. All objects of a reactive class have the same behavior, as given by the computational sequences of the extended statechart diagram associated with the class. The pair (*class*, *state machine*) defines a TROM type.

A reactive class can include attributes of basic types or abstract data types. For instance, a controller class may include a data type *queue* to receive and service objects from its environments. Several objects may be instantiated from a reactive class. They all have the same state machine for their behavior. There are other significant differences as well:

- A class is parameterized with port types. An object instantiated from this class can have finite number of ports of any port type. That is, objects having the same behavior may interact in different contexts.

- A state can be simple or complex. A complex state is the root of a hierarchy of substates. This feature provides a means of refining the behavior of reactive classes.

- An event may trigger a time-constrained event. If an event e triggers an event f and $P(t)$ is a time-constrained predicate associated with f, then the semantics requires the event f to occur during the interval of time in which $P(t)$ is *true*. If the event f does not occur during the interval of time in which $P(t)$ is *true*, the object enters into a *disabled* state and the event f is *disabled*.

- Two objects in the system communicate by exchanging a message through their compatible ports. The result of this communication is a synchronized transition, as in TA, for the objects. At any instant, some of the objects in the system may interact, and the rest of the objects may be engaged in some internal activity. Consequently, there is parallelism in the system composed of TROM objects.

- The events are partitioned into input (?), output (!) and internal events. Systems continuously receive input from and react to their environment. An internal event is initiated and controlled by an object and does not have any effect on other objects in the system. Hence, the internal behavior of an object is described more explicitly in TROM formalism than in TA or TIOA formalism. Although in TROM model, communication mechanism between TROMs is based on synchronous message passing, also known as *rendezvous* it can also model systems in which the objects operate asynchronously since events are classified into different categories.

- Input events to an object come either from the environment or from other objects in the system with which it interacts. Consequently, an input event is not time-constrained. An input or output or an internal event can occur at a state, cause state change, and may trigger time-constrained future events to happen in the object.

- In TROM model, there is a very clear distinction between those events whose performance is under the control of TROM object, and those events whose performance is under the control of its environment. A TROM object generates output and internal events autonomously, and transmits output instantaneously to its environment. In contrast, the input event is generated by the environment and transmitted instantaneously to the TROM object. Our distinction between the input and other events is fundamental, and based on who determines when the event is performed: a TROM object can have restrictions on when it will perform an output or internal event, but it is unable to block the performance of an input event.

Chapter 5

Unit Testing

5.1 Introduction

There are three stages in testing a real-time reactive system developed in TROM-LAB framework. Each stage focuses on testing the artifacts developed in one tier of TROM architecture. During the first stage abstract data type implementations are tested [AC95, Pro96]. The second stage has two parts: the generic reactive class implementations are tested in the first part of testing, and implementations of inherited classes are tested in the second part. During the third stage a reactive system composed of classes is tested. In this chapter we discuss a method for testing reactive classes. Using the Object Oriented terminology, we refer to testing a reactive class implementation as *unit testing.*

We propose two test adequacy criteria, called *state cover* and *transition cover* criteria and use them to generate test cases that form a minimal as well as an exhaustively test set for an implementation. Minimality implies that all test cases in the set are necessary to check conformity. Exhaustivity implies that the test cases in the generated set are sufficient.

We provide a method to discretize the timed statechart of a TROM class and construct a *grid automaton* with finite number of states, having no clocks, and only having untimed transitions. Each state of the grid automaton represents a durational TROM behavior, where the duration, called grid size, would be chosen based on the number of clocks in TROM specification. We give an algorithm to construct the grid automaton, state and prove a Homomorphism Theorem relating the behavior of the grid automaton to the behavior of its corresponding TROM. This theorem is to justify that testing the grid automaton implementation is equivalent to testing the TROM implementation. Finally we present the algorithms for deriving test cases from grid automaton satisfying the two test adequacy criteria. Homomorphism Theorem is the theoretical basis to justify the correctness of our testing strategy, the testing method is also practical.

5.2 Test Adequacy Criteria

An *exhaustive test suit* requires that every possible value and sequence of inputs be applied in every possible state of the system under test, thereby exercising every possible execution path. This approach results in an astronomical number of test cases, even with small programs. Exhaustive testing is a practical impossibility. Software testing is therefore concerned with small subsets of the exhaustive test suite.

The completeness of a test suite with respect to a particular test case design method is measured by coverage. Coverage is the percentage of elements required by a test strategy that have been exercised by a given test suite.

We consider state coverage and transition coverage as the two test adequacy criteria in our testing method.

- State Cover SC: A *state cover* is a set of test cases required to identify each state in the design to some states in the implementation. Each test case is a sequence of events $e_0.e_1.\cdots.e_i$, such that the transitions triggered by the sequence of events form a path from the initial state to the test state. We are interested in a state cover SC of minimum size.

- Transition Cover TC: A *transition cover* is a set of test cases required to identify each transition in the design to some actions in the implementation. The transition $s_i \xrightarrow{e} s_j$ can be tested by the test case $t.e$, where t is a test case in the state cover set SC covering the state s_i and $t.e \notin SC$. We are interested in a transition cover TC of minimum size.

The *test suite* for testing a TROM is defined by $T = SC \cup TC$. It forms a minimal set and can exhaustively test an implementation. Minimum implies that all test cases in the set are necessary. Exhaustivity implies that the test cases in the generated set are sufficient.

5.3 Homomorphism

In this section we define the grid automaton corresponding to a TROM and give the Homomorphism Theorem to relate them.

5.3.1 Clock Regions

Informally, a clock region characterizes a set of clock valuations and is used to define a grid automaton state.

A *term* over C is an expression generated by the grammar $e ::= c \mid n \mid e + n$, where C is a finite set of clocks, c is a clock in C, and n is a nonnegative integer, $n \in \mathbb{Z}^{\geq 0}$. In the obvious way, a clock valuation v is lifted to a function $\overline{v}$ that takes a term and returns a value. That is,

$$\overline{v}(c_1 + c_2 + k) = v(c_1) + v(c_2) + k.$$

A clock valuation is a high-order function: $v : C \to (\mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0})$

- $v(c)(0) = 0$
- $v(c)(x) \; > \; v(c)(y), \; \textit{if } x > y, \; x, y \in \mathbb{R}^{\geq 0}$
- For $x \in \mathbb{R}^{\geq 0}$, and $k \in \mathbb{Z}$, $v(c)(k + x) = k + v(c)(x)$.

A time constraint is a predicate, if m, m_1, m_2 is a time constant, $m, m_1, m_2 \in \mathbb{Q}^{\geq 0}$, then $v(c)(x) \leq m,\; v(c)(x) \geq m$, $v(c)(x) \geq m_1 \wedge \; v(c)(x) \leq m_2$ are time constraints.

A clock valuation v satisfies a time constraint δ, $\; v \models \delta$, iff δ evaluates to *true* using the values given by v.

The integral parts of clock values can become arbitrarily large. But if a clock c is never compared with a time constant greater than m, then its actual value, once it exceeds m, is of no consequence in deciding the allowed paths. We define the *intv* of a clock $c \in C$ to be an interval over $\mathbb{R}^{\geq 0}$ with infimum and supremum in $\mathbb{Q}^{\geq 0}$.

$$\mathit{intv}(c) \triangleq \mathit{domain}(c) - \{\infty\}$$

where the time domain of a clock, *domain*(c), is the set of nonnegative real numbers. If $d \in \mathbb{R}^{\geq 0}$, $x \in \mathbb{R}^{\geq 0}$, then $v \oplus d$ is the clock valuation defined by

$$(v \oplus d)(c)(x) \triangleq \begin{cases} v(c)(x) + d & \text{if } v(c)(x) + d \in intv(c) \\ \infty & \text{otherwise} \end{cases}$$

For all clock valuations of C, the equivalence relation $\cong$ is defined as follows: $v \cong v'$ iff, for all $c_1,\ c_2 \in C,\ x \in \mathbb{R}^{\geq 0}$,

1. $v(c_1)(x) = \infty$ iff $v'(c_1)(x) = \infty$
2. if $v(c_1)(x) \neq \infty$ then

   $\lfloor v(c_1)(x) \rfloor = \lfloor v'(c_1)(x) \rfloor$ and $(\mathit{fract}(v(c_1)(x)) = 0$ iff $\mathit{fract}(v'(c_1)(x)) = 0)$,
3. if $v(c_1)(x) \neq \infty \neq v(c_2)(x)$ then

   $\mathit{fract}(v(c_1)(x)) \leq \mathit{fract}(v(c_2)(x))$ iff $\mathit{fract}(v'(c_1)(x)) \leq \mathit{fract}(v'(c_2)(x))$.

A *clock region* is an equivalence class of clock valuations induced by equivalence relation $\cong$. We say that a clock region α satisfies a clock constraint δ iff every $v \in \alpha$ satisfies δ. Each region can be uniquely characterized by a (finite) set of clock constraints it satisfies. Each region can be represented by specifying

(1) for every clock c, one clock constraint from the set $\{v(c)(x) = m \mid m = 0, 1, \ldots, m_c\} \cup \{m - 1 < v(c)(x) < m \mid m = 1, \ldots, m_c\} \cup \{v(c)(x) > m_c\}$, where m_c is the upper bound of the time constant, $x \in \mathbb{R}^{\geq 0}$

(2) for every pair of clocks c_1 and c_2 such that $m_1 - 1 < v(c_1)(x) < m_1$ and $m_2 - 1 < v(c_2)(x) < m_2$ appear in (1) for some m_1,m_2, whether $\mathit{fract}(v(c_1)(x))$ is less than, equal to, or greater than $\mathit{fract}(v(c_2)(y))$.

For instance, if $m_1 = 4$, $m_2 = 6$, there exists 59 clock regions, as shown in Figure 19. Each region is interpreted by the clock values according to the equivalence relation definition. For instance, open regions $\alpha 1$ and $\alpha 16$ are defined by the inequalities:

$$\alpha 1: \ 0 < v(c_1)(x) < 1, 0 < v(c_2)(y) < v(c_1)(x)$$

$$\alpha 16: \ 3 < v(c_1)(x) < 4, v(c1)(x) - 2 < v(c_2)(y) < 2$$

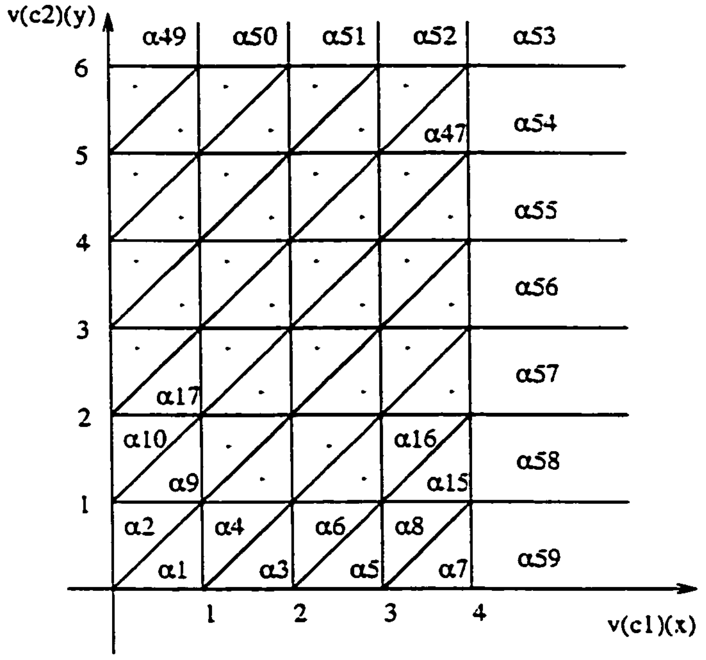


Figure 19: Clock Regions

A clock region α' is a *time-successor* of a clock region α iff for each $v \in \alpha$, there exists a positive $t \in \mathbb{R}$ such that $v+t \in \alpha'$. The time-successors of a clock region α are all the clock regions that will be visited by a clock valuation $v \in \alpha$ as time progresses. The time-successors of a region α can be derived by moving along a line drawn from some point in α in the diagonally upwards direction(parallel to the line $x = y$). The concepts of time-successor describes the relation over the clock regions, jumping from one region to the other region. For instance, in Figure 19, the successors of region $\alpha 1$ are : $\alpha 4$, $\alpha 11$, $\alpha 14$, $\alpha 21$, $\alpha 24$, $\alpha 31$, $\alpha 53$, $\alpha 54$, $\alpha 55$, $\alpha 56$.

The following Lemma states if two clock valuations are equivalent, then they produce the same value on any predicate.

Lemma 1

If $v \cong v'$ then $v \models \delta \iff v' \models \delta$

Proof.

The proof follows from the definition.

Based on the above discussion about time region, we know at every point in time the future behavior of a TROM object is determined by its states and the values of its clocks. This motivates the definition of *extended state*: For a TROM, an *extended state* is a pair $< s, v >$, where $s \in S$ and v is a clock valuation for C.

5.3.2 Definition of Grid Automaton

We have discussed two test adequacy criteria in previous section for specification-based testing: *state cover* and *transition cover*. The test case generation method that enforces these criteria will ensure *minimality* and *exhaustivity* of test cases generated. However, due to the possible occurrence of real numbers in continuous notion of time in TROM model, the state space of a TROM object is infinite: it is impossible in practice to generate an infinite number of test coverage sequences. Therefore, we construct a finite automaton using the fundamental concept of *region*. The key idea behind the definition of a region is that, even though the number of states is infinite, not all of these states are distinguishable via a constraint. The distinguishable states are those that correspond to the clock regions. When time constraints involve only bounded intervals, the number of clock regions of interest is only finite. Hence, the number of distinguishable states is also finite and are the states of the *grid automaton* associated with the TROM state machine.

For each TROM object A, the grid size, called *granularity* is $d = \frac{1}{k}$, where $k > 1$ is the number of clocks in A. If $k = 1$, we choose $d = \frac{1}{2}$. The grid automaton $\mathcal{G}_d(A)$, corresponding to the TROM state machine A is the automaton (Θ, θ_0, L, T) defined as follows:

States $\Theta = \{< s, v' > | < s, v >$ is an extended state of A and $v' = v + kd$, $v' < \infty$, $k \geq 0$ $\}$

Initial State $\theta_0 =< s_0, v_0 >$, the extended initial state in A

Transition Labels $L = \mathcal{E} \cup \{d\}$

Transition Function $T : \Theta \times L \rightarrow \Theta$ defined by

$$T(< s_i, v_i' >, e) =< s_{i+1}, v_i' > \text{ if } \; e \in \mathcal{E}$$
$$T(< s_i, v_i' >, d) =< s_i, v_i' + d >$$

subject to the following conditions:

- The transition

  $$< s_i, v_i > \xrightarrow{e} < s_{i+1}, v_i >$$

  in A, where e is not a time constraint event, with or without a clock initialization, corresponds to the transition

  $$< s_i, v_i' > \xrightarrow{e} < s_{i+1}, v_i' >$$

  of $\mathcal{G}_d(A)$.

- If the transition

  $$< s_i, v_i > \xrightarrow{e} < s_{i+1}, v_{i+1} >$$

  in A, where e is the event constrained to occur within the time shift $[l, u]$, then $v_i + l \leq v_{i+1} \leq v_i + u$ holds. The states and transitions corresponding to this are determined by the time at which the state s_i is reached:

  Assume η is minimal time required to reach s_i from s_0.
  Case 1: $l \leq \eta$
  b.1 $< s_i, \eta > \xrightarrow{d} < s_i, \eta + d > \xrightarrow{d} \ldots \xrightarrow{d} < s_i, \eta + u >$
  b.2 $< s_i, v_i' > \xrightarrow{e} < s_{i+1}, v_{i+1}' >$
  where $v_i' = v_{i+1}',\ v_i', v_{i+1}' \in \{\eta, \eta + d, \ldots, \eta + u\}$
  Case 2. $l > \eta$
  b.1 $< s_i, \eta > \xrightarrow{d} < s_i, \eta + d > \xrightarrow{d} \ldots \xrightarrow{d} < s_i, \eta + l > \xrightarrow{d} \ldots \xrightarrow{d} < s_i, \eta + u >$
  b.2 $< s_i, v_i' > \xrightarrow{e} < s_{i+1}, v_{i+1}' >$ where $v_i' = v_{i+1}',\ v_i', v_{i+1}' \in \{\eta + l, \eta + l + d, \ldots, \eta + u\}$

The enabling condition, port condition and post condition for the transition $< s_i, v_i >$ to $< s_{i+1}, v_{i+1} >$ in $\mathcal{G}_d(A)$, are the same as those in A for the transition from s_i to s_{i+1}, except that in the grid automaton, there is no time constraint and no clock.

For a grid automaton, the following lemmas can be proved:

Lemma 2
If $\theta_1, \theta_2, \theta_3 \in \Theta$, $\theta_1 \xrightarrow{d} \theta_3$, and $\theta_2 \xrightarrow{d} \theta_3$, then $\theta_1 = \theta_2$

Lemma 2 states that no state in the grid automaton can have more than one incoming transition labeled by d.

Lemma 3

For a grid automaton, there exists a state that it can have both outgoing transition labeled by $e \in \mathcal{E}$ and outgoing transition labeled by d.

Lemma 4

In a grid automaton, if there exists two transitions: $< s_i, v_i' > \xrightarrow{e} < s_j, v_j' >$ and $< s_k, v_k' > \xrightarrow{f} < s_j, v_j' >$, where $e, f \in \mathcal{E}$, then at any time instant, either $v_i' = v_j'$ or $v_k' = v_j'$ but not both are true.

5.3.3 Homomorphism

A *homomorphism* between two sets is a mapping of the elements of one set to the other such that their respective binary operations are preserved:

$$H : (X, \circ) \rightarrow (Y, *)$$

such that $H(x)$ is an element of y and for any pair $x_1, x_2 \in X$

$$H(x_1 \circ x_2) = H(x_1) * H(x_2).$$

In defining a grid automaton we carry over the transition specifications, ports, and attributes from its corresponding TROM machine. However, the transition specification for a transition in the TROM specification may be copied to more than one transition in the grid automaton. So, it is sufficient to define a mapping that maps the state of $\mathcal{G}_d(A)$ to the state of A preserving the transition relations:

$$H : \mathcal{G}_d(A) \rightarrow A$$

has the following properties:

- a state θ of $\mathcal{G}_d(A)$ is mapped to a state s of A, and
- a transition between two states θ_1 and θ_2 of $\mathcal{G}_d(A)$ is mapped either to a *silent transition* within the state s_1 or to a transition between the states s_1 and s_2 of A.

That is, H is defined as follows:

- For each state $< s, v' > \in \Theta$, there exists a unique extended state $< s, v >$ in A. The initial state $< s_0, v_0 >$ of the grid automaton is mapped onto the initial state of A.

- $L = \mathcal{E} \cup \{d\}$ is mapped onto $\mathcal{E}$, and $\frac{1}{d}$ is the number of clocks in A.
- Transition Preservation (extended TROM states are shown):
  - $H(< s_i, v'_i > \xrightarrow{e} < s_{i+1}, v'_{i+1} >) = < s_i, v_i > \xrightarrow{e} < s_{i+1}, v_i >$
  - $H(< s_i, v'_i > \xrightarrow{e} < s_{i+1}, v'_{i+1} >) = < s_i, v_i > \xrightarrow{e} < s_{i+1}, v_{i+1} >$, where $v'_i = v'_{i+1}$, $v'_i, v'_{i+1} \in \{\eta, \eta+d, \ldots, \eta+u\}$ or $\{\eta+l, \eta+l+d, \ldots, \eta+u\}$, $l \leq v_i = v_{i+1} \leq u$
  - $H(< s_i, v'_i > \xrightarrow{d} < s_i, v'_i + d >) = < s_i, v_i > \xrightarrow{i} < s_i, v_i >$, where i is a silent transition
  - $H(< s_i, v'_i > \xrightarrow{e} < s_i, v'_i >) = < s_i, v_i > \xrightarrow{e} < s_i, v_i >$

5.3.4 Homomorphism Theorem

A *computational path* in a grid automaton is a sequence of transitions starting from the initial state of the automaton. The sequence need not be finite. The homomorphical mapping H can be extended in a natural manner to map a computational path of $\mathcal{G}_d(A)$ to a computational path in A. That is, H can be extended to a map

$$H^* : Comp(\mathcal{G}_d(A)) \to Comp(A).$$

If σ' is a computational path in the grid automaton $\mathcal{G}_d(A)$, then $H^*(\sigma')$ can be inductively defined as follows:

$$\sigma' = (< s_0, v'_0 > \xrightarrow{e_0} < s_1, v'_1 >).\sigma'_1$$

where σ'_1 is the path following the transition caused by e_0.

$$H^*(\sigma') = H^*((< s_0, v'_0 > \xrightarrow{e_0} < s_1, v'_1 >).\sigma'_1)$$

$$= H(< s_0, v'_0 > \xrightarrow{e_0} < s_1, v'_1 >).H^*(\sigma'_1)$$

The map H^* is defined over all computational paths in the grid automaton. Hence, the behavior of the grid automata $\mathcal{G}_d(A)$ is *homomorphicto* the behavior of A. We summarize these results as a theorem.

Homomorphism Theorem

A computational path in a grid automaton $\mathcal{G}_d(A)$ is homomorphic to a computational path in its corresponding TROM A. The behavior of A is a homomorphic image of the behavior of its grid automaton $\mathcal{G}_d(A)$.

Based on the homomorphism result, we can define a notion of *simulation* of TROM behavior by its grid automaton. Since the states of the grid automaton $\mathcal{G}_d(A)$ are clock regions, the equivalence classes of clock valuations, the behavior of the grid automaton is equivalent to the timed behavior of A. That is, the timed behavior of A is *simulated* by the grid automaton with clock valuations as observations.

5.4 Unit Testing

If P is the program under test for conformance to the specification A, according to the *Homomorphism Theorem*, it is sufficient to test the program P for conformance to the specification $\mathcal{G}_d(A)$. We provide a method for constructing the grid automaton $\mathcal{G}_d(A)$ for a given A and an algorithm for generating test cases from the grid automaton, meeting the coverage criteria.

Assume that we are given a program that implements a reactive class. The program is tested by generating test cases from the grid automaton corresponding to the state machine of the reactive class. A test case is a generic template, which when instantiated with specific values (time, attribute values) gives a collection of tests to be administered to the program. For each test, the output from the program is filtered to contain only global variables, such as external events and attribute values. These are matched against the post-condition for the test case.

5.4.1 Algorithm GA: Grid Automaton Construction

The algorithm accepts a TROM class specification, and outputs a grid automaton specification. The algorithm chooses the granularity d of the TROM with $k > 1$ clocks to be $d = \frac{1}{k}$; However, if the TROM class has only one clock, the granularity is chosen as $\frac{1}{2}$. Since each region is defined by k constraints, one constraint for each clock, it is sufficient to choose k samples from each region. Hence, we justify the quantization $d = \frac{1}{k}$. For each time-constraint $TC_i = \{R_i, e, [t_i, t_j], \{\}\}$ in a TROM class specification, the following steps are done:

Step 0. Initialize: ES: events of TROM; SS: states of TROM; TS: transition specifications of TROM.

Step 1. Find the transition specification of TROM constrained by TC_i

$R_i :< s_i, s_j >$; e(port condition); enabling condition => post condition

Step 2. Compute the grid points: $< s_i, v'+0 >, < s_i, v'+d >, < s_i, v'+2d >, \ldots, < s_i, v'+md >$, where m is the largest positive integer for which $md = t_j$;

Step 3. Add the internal event d to ES; remove s_i from SS, and add the grid points computed in Step 2 to SS. Attribute-function section (active attributes) are not changed.

Step 4. Do the following changes to TS:

Step 4.1. For n, $0 \leq n \leq t_j - d$, increased by d each time, add transitions(time increment) to TS:

$<< s_i, v'+n >, < s_i, v'+n+d >>$; $d(true)$; $true => true$;

Step 4.2. Remove the transition from TS:

$R_i :< s_i, s_j >$; e (port condition); enabling condition => post condition

Step 4.3. For n, $t_i \leq n \leq t_j$, increased by d each time, add the transitions(time constraint) to TS:

$<< s_i, v'+n >, < s_j, v'+n >>$; e (port condition); enabling condition => post condition;

The correctness of the algorithm can be proved to establish that the grid automaton constructed by the algorithm is homomorphic to the given TROM object.

5.4.2 The complexity of the algorithm GA

We measure the complexity of the algorithm in terms of the number of new states and number of new transitions introduced in the construction. Only a time constrained transition in the object A introduces new states and transitions in $\mathcal{G}_d(A)$. Consider a transition for which $[l, u]$ is the time constraint. The new states introduced are

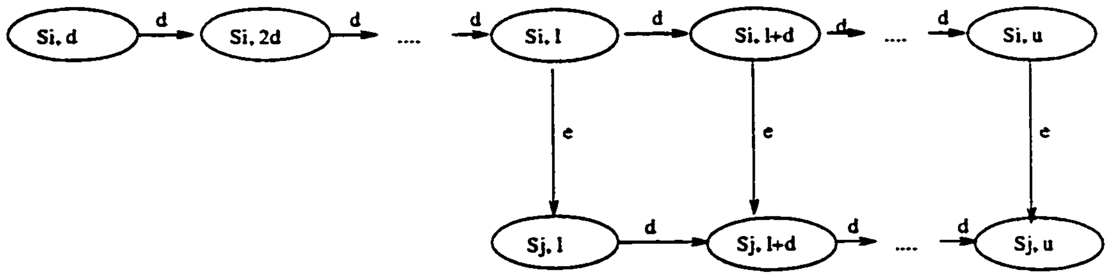


Figure 20: Grid Automaton for a Time Constrained Transition

$< s_i, d >, < s_i, 2d >, \ldots, < s_i, l >, < s_i, l+d >, \ldots, < s_i, u >$. That is, $\frac{u}{d}$ new states are introduced in the construction. See Figure 20. In between a pair of new states, there is a new transition, and from each state in the sequence $< s_i, l >, < s_i, l+d >$ $, \ldots, < s_i, u >$, there is a transition to the state $< s_{i+1}, v >$. Hence, the number of new transitions (including time increase) is $\frac{u}{d} + (\frac{u-l}{d} - 1) = (\frac{2u-l}{d}) - 1$. If l_{min}, u_{max} denote the minimal of the lower bounds and maximal of upper bounds of all the time constraints, and k is the number of clocks, $k = \frac{1}{d}$, then the number of new states is bounded by $k\frac{u_{max}}{d} = k^2 u_{max}$, the number of new transitions is bounded by $k^2(2u_{max} - l_{min}) - k$. The complexity of the algorithm GA is $O(k^2 u_{max} + k^2(2u_{max} - l_{min}) - k) = O(k^2 u_{max}) = O(u_{max})$. Hence, the size of the grid automaton increases *linearly* with the size of the maximum time constraint, assuming the number of clocks is fixed.

5.4.3 Example

The derived grid automata for train object, controller object and gate object are shown in Figures 21, 22, and 23 respectively, where the state machine is augmented with the label of time increment $\frac{1}{2}$ and the corresponding grid points to represent time increments. The state $< toCross, 1/2 >$ in Figure 21 is in the open region $\alpha 1$ in Figure 19.

5.4.4 Algorithm TC: Generating Test Cases from a Grid Automaton

The algorithm is in two parts: first, a set of test sequences based on state coverage are generated; next, test sequences based on transition coverage are generated, and

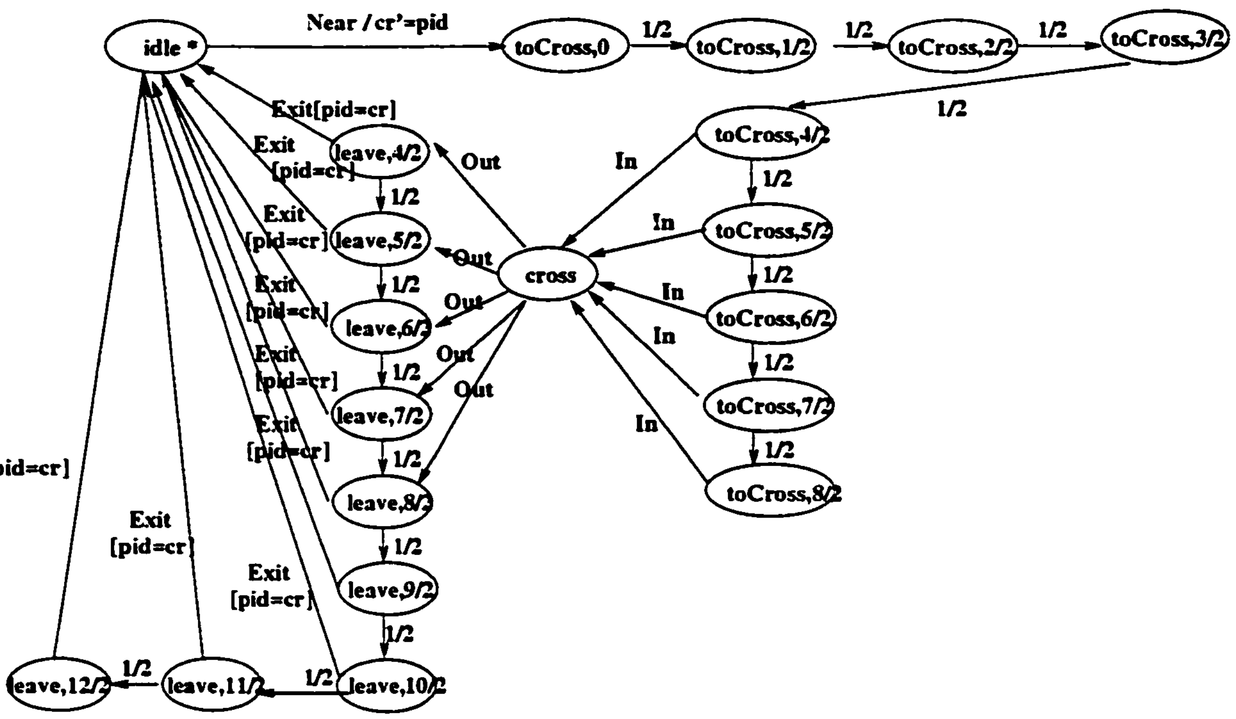


Figure 21: Grid Automaton for Train

are added to the state cover. The resulting set of test cases has the minimality and exhaustivity properties.

- **STC: State Coverage Algorithm**

For every state θ_i, the algorithm generates a set of sequences of events that brings the grid automaton from its initial state θ_0 to θ_i.

Step 1. initially, Θ: set of States; Θ_0: initial states

$\Theta = \{\theta_0, \theta_1, \cdots, \}$, $\Theta_0 = \{\theta_0\}$, $i = 0$, C_i: set of event sequences covering θ_i, $C_0 = \{\epsilon\}$

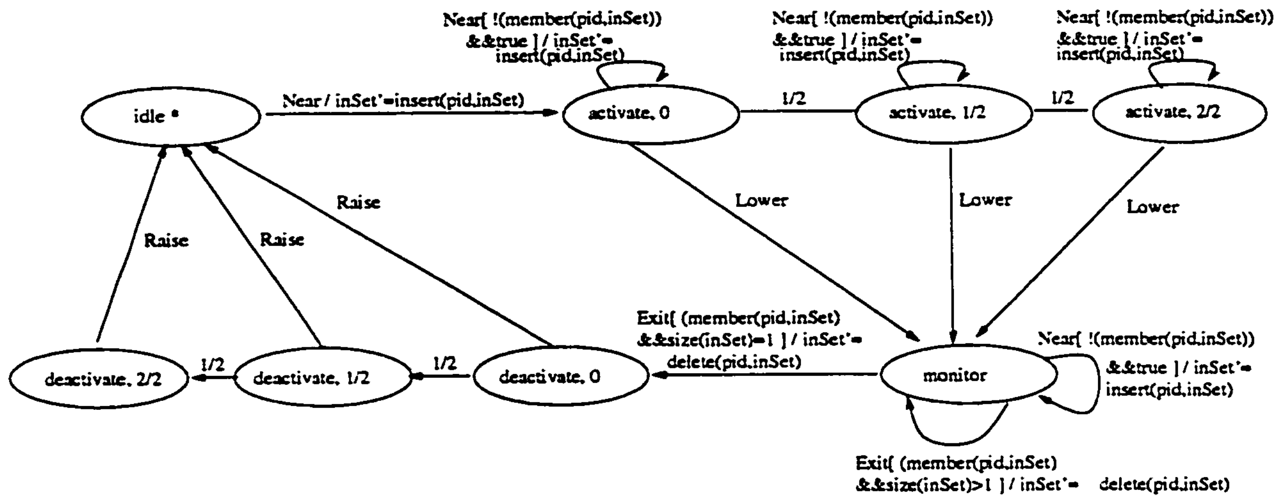
Step 2. remove θ_i from Θ, if Θ become empty, stop, $SC = C$, $C = \bigcup_{k=0}^{i} C_k$

Step 3. $i = i + 1$

$C_i = \{c = p.q \mid p \in C_{i-1}, p$ is the test sequence for covering θ_{i-1} and q is labeling the transition $\theta_{i-1} \rightarrow \theta_i\}$

Step 4. go to *Step 2.*

- **TRC: Transition Coverage Algorithm**



idle *
Near / inSet'=insert(pid,inSet)
activate, 0
activate, 1/2
activate, 2/2
1/2
Near[!(member(pid,inSet)) &&true] / inSet'= insert(pid,inSet)
Lower
Raise
deactivate, 2/2
deactivate, 1/2
deactivate, 0
monitor
Exit[(member(pid,inSet) &&size(inSet)=1] / inSet'= delete(pid,inSet)
Exit[(member(pid,inSet) &&size(inSet)>1] / inSet'= delete(pid,inSet)

Figure 22: Grid Automaton for Controller

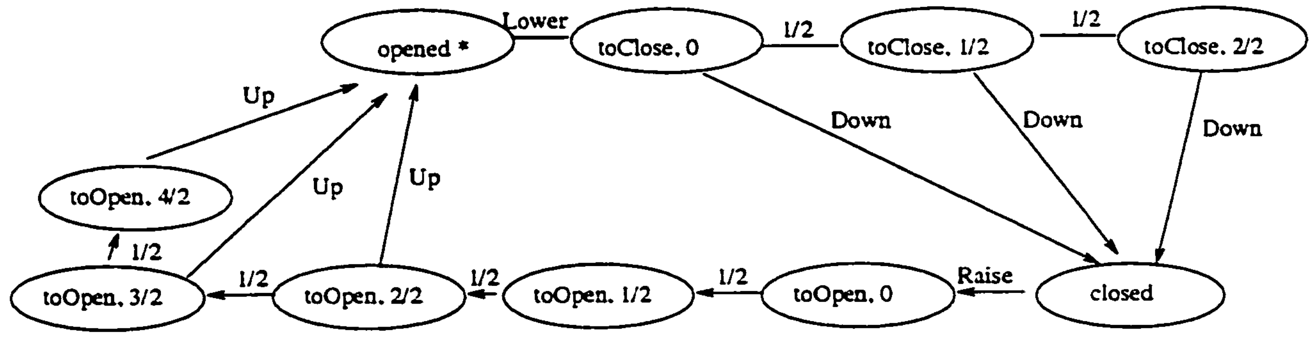


Figure 23: Grid Automaton for Gate

For every transition, $\theta_i \xrightarrow{e} \theta_j$, the algorithm generates a set of sequences of events x, $y = x.e$ such that x brings the grid automaton from its initial state θ_0 to θ_i. The event sequences will completely cover all the transitions except those that are already covered in the state coverage SC.

Step 1. initially, T: set of transitions; i=0,

for every transition $R_i : \theta_i \xrightarrow{e} \theta_j$ D_i is the set of event sequences covering the transition R_i: $D_i = \{t.e \mid t \in C_i, t.e \notin SC\}$

Step 2. remove transition R_i from T, if T become empty, stop, $TC = D$, $D = \bigcup_{k=0}^{i} D_k$

Step 3. $i = i + 1$

Step 4. go to *Step 2.*

The properties of state and transition cover sequences can be characterized as follows:

- $SC(A)$: set of state covers for $\mathcal{G}_d(A)$
- $TC(A)$: set of transition covers for $\mathcal{G}_d(A)$
- $T_d(A) = SC(A) \cup TC(A)$
- For each state $\theta \in \mathcal{G}_d(A)$, there exists a state cover of θ: $sc(\theta) \in SC(A)$
- If $\theta_i \xrightarrow{e} \theta_j$ is a transition in $\mathcal{G}_d(A)$, there exists a transition cover $tc(\theta_i \xrightarrow{e} \theta_j) \in TC(A)$ such that $tc(\theta_i \xrightarrow{e} \theta_j) = sc(\theta_1).e$

Our method implicitly generates sequential and concurrent test sequences; however our method does not differentiate between them.

Chapter 6

Testing Derived Classes

6.1 Introduction

Test cases for derived classes must be generated incrementally from the test cases of the base class, otherwise the complexity of the testing process increases. In TROM methodology, there are strict rules for refining a base class [Ach95]. The test methods discussed in this chapter are based on these rules.

Inheritance is one of the most essential tools through which we organize the hierarchical structure of a complex system. According to Booch [Boo91]:

> inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes.

There are two ways of looking at the inheritance relationship in a system: *essential* inheritance, which implies inheritance of specifications, and *incidental* inheritance, which implies inheritance of implementation. The inheritance of specification provides specialization-generalization hierarchies in the system, whereas the inheritance of implementation is primarily a code-reuse mechanism. Here, we look at the impact of inheritance of specification on the testing of real-time reactive systems. The testing strategy must consider the possibility of substitution of a base class where a derived class has been used and so on.

In this chapter, we first discuss the three types of inheritance in TROM, and next we explain the test case generation for every type of inheritance. The test cases generated from the base class are used for test case generation of the derived class.

6.2 Concept of Inheritance in TROM

Inheritance in TROM is based on *subtyping* principles. They are used in two basic forms:

- as a means of modeling conceptual generalization/specialization relationship, and
- as a means of reusing existing class specifications in the definition of new classes.

In order to accommodate these needs, three kinds of subtype relations have been introduced in [Ach95]. These are called *behavioral subtyping, extensional subtyping,* and *polymorphic subtyping.* A class $\mathcal{A}$ that is a behavioral subtype of another class $\mathcal{A}_B$ ensures both substitutability and behavior preservation. That is, an object A of class $\mathcal{A}$ has the same behavior as an object A_B of class $\mathcal{A}_B$ and A can be substituted in any configuration where A_B is expected to participate. A class $\mathcal{A}$ that is an extensional subtype of another class $\mathcal{A}_E$ ensures context-independent preservation of behavior. That is, A preserves the behavior of A_E, but it cannot be substituted where A_E is expected to participate. A class $\mathcal{A}$ that is a polymorphic subtype of another class $\mathcal{A}_P$ ensures context-dependent preservation of behavior and substitutability. That is, A preserves the behavior of A_P only in the context of A_P, and it can be substituted where A_P is expected to participate. Inheritance mechanisms that achieve these effects in class refinements are respectively called *behavior inheritance*(BI), *extensional inheritance*(EI) and *polymorphic inheritance*(PI).

Extensional inheritance and polymorphic inheritance allow the addition of new transitions. The added transition may involve a new event, new states, new time-constraints, and new attributes. In addition, enabling condition, port-condition and post-condition may be introduced in the added transition. The difference between extensional inheritance and polymorphic inheritance is that extensional inheritance adds new items only related to make an existing simple state as a compound state while polymorphic inheritance may add new items without relating to a compound state. The addition of new time constraints to the new transitions must be consistent with the time constraints in the base class.

Assuming that the base class has been tested, the goal of testing derived classes is to avoid constructing the grid automaton for the inherited classes, but derive test cases of the derived classes from the test cases of the base class. Otherwise, complexity of testing process increases. For instance, by adding a new time constraint, the granularity $d = \frac{1}{k}$, where k is the number of clocks, for the grid automaton, it is increased, thus forcing more grid points in the grid automaton of the derived class. That is, the size of the grid automaton increases, thus requiring more work in generating test sequences for state and transition covers. More importantly, in evolving systems where new derived classes are created, it is important to test them as optimally as possible.

The testing methods discussed in this chapter are based on the following observations:

- the state machine description of a class that describes the behavior of objects of the class contains sufficient information for test case generation;
- an object derived by behavioral inheritance from a base object should be tested for behavior preservation and substitutability; that is, every test case of the base object, when applied to the inherited object, must produce the same result as it would on the base object;
- test cases of an object obtained by extensional inheritance of a base object should include the test cases of the base object, and when a common test case is applied to the derived object, it should produce the same result as the base object;
- test cases of an object obtained by polymorphic inheritance of a base object should produce the test cases of the base object when projected on the elements of the base class.

In testing derived classes, the following principles play a crucial role. Let A be the base object and A_I be an inherited object (any one of the three kinds). Let $\mathcal{G}_d(A)$ and $\mathcal{G}_d(A_I)$ be respectively their grid automaton.

1 By the Homomorphism Theorem, for every computation τ_d of $\mathcal{G}_d(A)$, there exists a computation τ of A such that $H(\tau_d) = \tau$, and for every computation τ_{I_d} of $\mathcal{G}_d(A_I)$, there exists a computation τ_I of A_I such that $H_I(\tau_{I_d}) = \tau_I$.

2 If MI is the inheritance mapping (any one of the three kinds) then $MI(\tau_I) = \tau$

3 From 1 & 2, we have $MI(H_I(\tau_{I_d}) = H(\tau_d)$. This means that there exists a mapping $\varpi : \mathcal{G}_d(A_I) \rightarrow \mathcal{G}_d(A)$ such that $H(\varpi(\tau_{I_d})) = MI(H_I(\tau_{I_d}))$. That is, $\tau_d = \varpi(\tau_{I_d})$. See Figure 24.

$$\begin{tikzcd}
A & & A_I \arrow[ll, "MI"'] \\
\mathcal{G}_d(A) \arrow[u, Rightarrow, "H"] \arrow[d, dash, dotted] & & \mathcal{G}_d(A_I) \arrow[u, Rightarrow, dashed, "H_I"'] \arrow[ll, dashed, "MI'"'] \arrow[d, dash, dotted] \\
T_d(A) & & T'_d(A_I) \equiv T_d(A_I) \arrow[ll, dashed, "MI'"']
\end{tikzcd}$$

Figure 24: The Principle of Inheritance

4 The Homomorphism Theorem can be applied to map the sets of test sequences, since a test sequence is either a state cover or a transition cover, which in turn is a partial computational path.

The following notations are used: $T_d(A)$ denotes the set of test sequences generated from $\mathcal{G}_d(A)$, it will be used to test object A; $T_d(A) = SC(A) \cup TC(A)$, where $SC(A)$ is the set of state cover sequences, and $TC(A)$ is the set of transition cover sequences; $T_d(A)$ is derived from $\mathcal{G}_d(A)$ using the test case generation algorithm. BI, EI, and PI denote respectively the mappings defining the behavioral, extensional, and polymorphic inheritances. $T_d(A_B)$, $T_d(A_E)$, $T_d(A_P)$ denote respectively the set of test sequences that will be generated by the unit testing algorithm from the grid automatons $\mathcal{G}_d(A_B)$, $\mathcal{G}_d(A_E)$, $\mathcal{G}_d(A_P)$. $T'_d(A_B)$, $T'_d(A_E)$, $T'_d(A_P)$ denote respectively the set of test sequences derived from $T_d(A)$ by the construction discussed in the sections below.

In the light of the observations and principles enunciated above, the following equivalences can be shown :

$$T_d(A_B) \equiv T'_d(A_B)$$

$$T_d(A_E) \equiv T'_d(A_E)$$

$T_d(A_P) \equiv T'_d(A_P)$

6.3 Behavioral Inheritance

The mechanism for deriving behavioral inheritance TROM objects is defined in [Ach95] as follows:

- *Attribute redefinition*: A port type may be renamed; The data model of an attribute may be redefined, provided there exist coercion functions for each attribute from the redefined trait to the original trait.
- *Transition redefinition*: Redefinition of an inherited transition specification may be done such that:
    - The *post condition* may be strengthened.

      $$a'_{post\_cond} \Rightarrow a_{post\_cond}$$

    - The *port condition* of a transition may be strengthened.

      $$g'_{port\_cond} \Rightarrow g_{port\_cond}$$

    - The *enabling condition* of a transition may be strengthened.

      $$g'_{enabling\_cond} \Rightarrow g_{enabling\_cond}$$

- *Time-constraint redefinition*: The minimal time delay may be increased or maximal time delay may be decreased.

No new clock is defined in the inherited object. Hence, the grid size for $\mathcal{G}_d(A)$ and $\mathcal{G}_d(A_B)$ are the same. However, due to the strengthening of some time constraints, $\mathcal{G}_d(A_B)$ will not have some of the states and transitions of $\mathcal{G}_d(A)$. Based on this observation a method is outlined to construct test sequences of the derived object from the test sequences of the base object. We consider test case generation for each one of the rules defined above. Our hypothesis is that object A has been tested, and $T_d(A)$ is available.

6.3.1 Attribute Redefinition

Renaming a port type will have no effect on the set of test cases. Although abstract data type redefinition will not affect the set of test cases, it will have an impact on the evaluation of the test results. For simplicity, assume that only one attribute has been refined. Let $a \in \mathcal{X}$, $a' \in \mathcal{X}'$, $\chi(f'(a')) = f(a)$, where χ is the coercion function for the attribute a': $a' \xrightarrow{\varphi} a$, $\mathcal{X}$ is the set of attributes in TROM A, $\mathcal{X}'$ is the set of attributes in TROM A_B, f' and f are the operations in $\mathcal{X}'$ and $\mathcal{X}$, respectively. Since the attribute is redefined, we will have to look at whether it has affect on the transition in terms of *enabling condition*, *port condition* and *post condition*. Since only attribute trait is redefined, the test cases generated for the base object are sufficient to test the behavior of the inherited object. The only check we have to make is that the transitions for which the *enabling condition*, *port condition* and *post condition* involve the refined attribute retain the same behavior of the corresponding transition in the base class. This check does not require any new test case. It requires that the result of an existing test case satisfies the refined post condition. Assume that the transition $r : g \Rightarrow a$ of TROM A becomes the transition $r' : g' \Rightarrow a'$ of TROM A_B. Both g and g' can be expressed in the form $g = g_{enabling\_cond} \wedge g_{port\_cond} \wedge g_{time\_constraint}$, and $a = a_{post\_cond} \wedge a_{clock\_init}$. Four situations arise:

Case 1: $g = g'$, $a = a'$: We don't have to test r'.

Case 2: $g \neq g'$, $a = a'$: If $g' \Rightarrow g$, then the test case used to test the transition $r : g \Rightarrow a$ can be used to test the transition $r' : g' \Rightarrow a'$.

Proof

(1): $g' \Rightarrow g$ (hypothesis)
(2): $g \Rightarrow a$ (base object assertion)
(3): from (1) and (2) infer: $g' \Rightarrow a$
(4): $a' = a$(hypothesis)
(5): from (3) and (4) infer: $g' \Rightarrow a'$

Case 3: $g = g'$, $a \neq a'$: If $a' \Rightarrow a$, then the test case c' used to test the transition $r' : g' \Rightarrow a'$ can be mapped to c that tests the transition $r : g \Rightarrow a$.

Proof

(1): c tests $r : g \Rightarrow a$ (hypothesis)
(2): c' tests $r' : g' \Rightarrow a'$ (hypothesis)

(3): $a' \Rightarrow a$(hypothesis)

(4): from (2) and (3) infer: c' tests $g' \Rightarrow a$

(5): $g = g'$(hypothesis)

(6): from (4) and (5) infer: c' tests $g \Rightarrow a$

therefore, the coericion function maps c' to c.

Case 4: $g \neq g'$, $a \neq a'$: If $g' \Rightarrow g \wedge a \Rightarrow a'$, then the test case c used to test the transition $r : g \Rightarrow a$ can be used to test the the transition $r' : g' \Rightarrow a'$.

Proof

(1): $g \Rightarrow a$ (base object assertion)

(2): $a \Rightarrow a'$(hypothesis)

(3): from (1) and (2) infer: $g \Rightarrow a'$

(4): $g' \Rightarrow g$ (hypothesis)

(5): from (3) and (4) infer: $g' \Rightarrow a'$

Hence we conclude that $T_d(A_B) = T_d(A)$ with the provision that they are applied and result evaluated according to the four situations discussed above.

6.3.2 Transition Redefinition

The redefinition of a transition involves strengthening one or more of the three predicate expressions: enabling condition, port condition, post condition. Using proofs similar to the proofs in the previous section, we can show that test set is not changed.

6.3.3 Time-constraint Redefinition

A time constraint can be redefined by increasing the minimal time delay or by decreasing the maximal time delay.

The minimal time delay is increased

Let e, where $s_i \xrightarrow{e} s_{i+1}$, be the event constrained by the minimum and maximum time delays given by $[l, u]$. If the lower bound is increased to l', the time constraint for event e is changed to $[l', u]$, where $l, l', u \in \mathbb{Q}^{\geq 0}$. Let η be the minimal time required to reach the source state s_i from the original state s_0, d is the unit of time increment. If $\eta \geq l'$, then $T_d(A_B) = T_d(A)$; otherwise, $T_d(A_B)$ is computed as follows.

Case 1 $l \leq \eta \leq l'$

The test suite $T_d(A_B)$ for object A_B is constructed in three steps.

1. Let $X = \{< s_i, \eta >, < s_i, \eta + d > \ldots, < s_i, l' - d >\}$. We delete from $TC(A)$ all the test sequences for covering transitions whose source states are in X and labeling events are e :

$$T_d(A_B)_1 = T_d(A) \setminus \{tc(e) \in T_d(A) \mid tc(e) = sc(\theta).e \wedge \theta \in X\}$$

2. If there exists a state $\theta \in \mathcal{G}_d(A)$ and the state cover $sc(\theta) \in T_d(A)$ is of the form $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{\eta, \eta + d, \ldots, l' - d\}$, p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new state cover of θ, $sc(\theta) = sc(< s_i, l' >).e.p$:

$$T_d(A_B)_2 = (T_d(A_B)_1 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p\}) \cup \{sc(< s_i, l' >).e.p\}$$

3. If there exists a state θ with an outgoing transition e' such that $tc(e') \in TC(A)$, $tc(e') = sc(\theta).e'$, $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{\eta, \eta + d, \ldots, l' - d\}$, p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new transition cover of e', $tc(e') = sc(< s_i, l' >).e.p.e'$:

$$T_d(A_B) = (T_d(A_B)_2 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p.e'\}) \cup \{sc(< s_i, l' >).e.p.e'\}$$

Case 2 $l > \eta$

The test suite $T_d(A_B)$ for object A_B is constructed in three steps.

1. Let $X = \{< s_i, l >, < s_i, l + d > \ldots, < s_i, l' - d >\}$. We delete from $TC(A)$ all the test sequences for covering transitions whose source states are in X and labeling events are e :

$$T_d(A_B)_1 = T_d(A) \setminus \{tc(e) \in T_d(A) \mid tc(e) = sc(\theta).e \wedge \theta \in X\}$$

2. If there exists a state $\theta \in \mathcal{G}_d(A)$ and the state cover $sc(\theta) \in T_d(A)$ is of the form $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{l, l + d, \ldots, l' - d\}$,

p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new state cover of θ, $sc(\theta) = sc(< s_i, l' >).e.p$:

$$T_d(A_B)_2 = (T_d(A_B)_1 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p\}) \cup \{sc(< s_i, l' >).e.p\}$$

3. If there exists a state θ with an outgoing transition e' such that $tc(e') \in TC(A)$, $tc(e') = sc(\theta).e'$, $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{l, l+d, \ldots, l'-d\}$, p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new transition cover of e', $tc(e') = sc(< s_i, l' >).e.p.e'$:

$$T_d(A_B) = (T_d(A_B)_2 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p.e'\}) \cup \{sc(< s_i, l' >).e.p.e'\}$$

The maximal time delay is decreased

Let e, where $s_i \xrightarrow{e} s_{i+1}$, be the event constrained by the minimum and maximum time delays given by $[l, u]$. If the upper bound is decreased to u', the time constraint for event e is changed to $[l, u']$, where $l, u', u \in \mathbb{Q}^{\geq 0}$. Let η be the minimal time required to reach the source state s_i from the original state s_0, d is the unit of time increment. $T_d(A_B)$ is computed as follows.

The test suite $T_d(A_B)$ for object A_B is constructed in four steps.

1. Let $X = \{< s_i, u'+d >, < s_i, u'+d+d > \ldots, < s_i, u >\}$. We delete from $TC(A)$ all the test sequences for covering transitions whose source states or destination states are in X :

$$T_d(A_B)_1 = T_d(A) \setminus \{tc(r) \in TC(A) \mid source(r) \in X \vee destin(r) \in X\}$$

2. We delete from $SC(A)$ all the test sequences for covering any state θ, where $\theta \in X$:

$$T_d(A_B)_2 = T_d(A_B)_1 \setminus \{sc(\theta) \in SC(A) \mid \theta \in X\}$$

3. If there exists a state $\theta \in \mathcal{G}_d(A)$ and the state cover $sc(\theta) \in T_d(A)$ is of the form $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{u'+d, u'+d+d, \ldots, u\}$, p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new state cover of θ, $sc(\theta) = sc(< s_i, u' >).e.p$:

$$T_d(A_B)_3 = (T_d(A_B)_2 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p\}) \cup \{sc(< s_i, u' >).e.p\}$$

4. If there exists a state θ with an outgoing transition e' such that $tc(e') \in TC(A)$, $tc(e') = sc(\theta).e'$, $sc(\theta) = sc(< s_{i+1}, v'_{i+1} >).p$, where $v'_{i+1} \in \{u' + d, u' + d + d, \ldots, u\}$, p is a path from state $< s_{i+1}, v'_{i+1} >$ to state θ, then replace it by the new transition cover of e', $tc(e') = sc(< s_i, u' >).e.p.e'$:

$$T_d(A_B) = (T_d(A_B)_3 \setminus \{sc(< s_{i+1}, v'_{i+1} >).p.e'\}) \cup \{sc(< s_i, u' >).e.p.e'\}$$

6.3.4 Example

We illustrate the test case generation for *OrdChannel* object which is a behavioral inheritance of *Channel* object.

```
Class Channel [@P,@Q]
 Events: SendA?P, SendB?P, DelvA!Q, Delv!Q
 States: *idle, active
 Attributes: inChn: MBag, x: msg
 Attribute-function:
  idle ↦ {inChn}; active ↦ {inChn,next};
 Transition-Specifications:
  R1 : ⟨idle, active⟩, ⟨active, active⟩;
      SendA?(true); true ⟹ inChn' = insert(A,inChn)
      ∧ x' = A;
  R2 : ⟨idle, active⟩, ⟨active, active⟩;
      SendB?(true); true ⟹ inChn' = insert(B,inChn)
      ∧ x' = B;
  R3 : ⟨active, active⟩; DelvA!(size(inChn) > 1 ∧ x = A);
      true ⟹ inChn' = delete(A,inChn);
  R4 : ⟨active, active⟩; DelvB!(size(inChn) > 1 ∧ x = B);
      true ⟹ inChn' = delete(B,inChn);
  R5 : ⟨active, idle⟩; DelvA!(size(inChn) = 1 ∧ x = A);
      true ⟹ inChn' = {};
  R6 : ⟨active, idle⟩; DelvA!(size(inChn) = 1 ∧ x = B);
      true ⟹ inChn' = {};
end
```

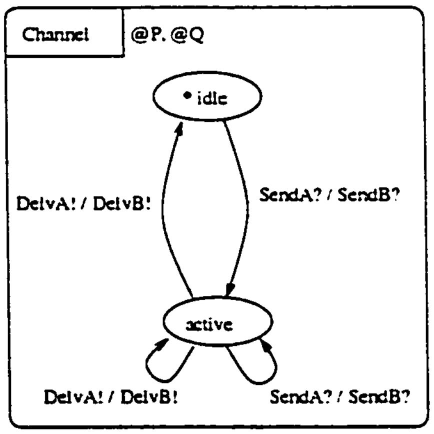


Figure 25: Class Specification of Channel

Figure 25 shows TROM class specification for a communication channel. The channel receives and delivers two types of message, *A-message* and *B-message*. Sending an *A-message* (*B-message*) is represented by the input event *SendA*(*SendB*). Similarly, the delivery of an *A-message*(*B-message*) to the receiver is represented by the output event *DelvA*(*DelvB*). The channel can be either *idle* or *active*. The attribute *inChn* is of sort *Bag* and keeps the message that are currently in the channel. In order to deliver a message, an arbitrary message from the bag is picked and delivered. Thus the channel does not preserve the order of message sent. Figure 26 shows the definition of the class OrdChannel derived from the class Channel in behavioral inheritance. The sort for the attribute *inChn* is redefined from *Bag* to a *Queue*. Accordingly, the *post condition* and the *enabling condition* of the transitions are modified. The communication channel modeled by the TROM class OrdChannel delivers messages in the order in which they are received. Every TROM object instantiated from the TROM class OrdChannel inherits the behavior of a TROM object instantiated from the TROM class Channel. There exists a coercion function that maps the trait *Queue* to the trait *Bag*, and a proof that an OrdChannel object is a behavior subtype of a Channel object are given in [Ach95]. We remark that a TROM object of the class OrdChannel can be substituted for a TROM object of the class Channel in every context.

```
Class OrdChannel [@P,@Q]
 Inherits: Channel
 Attributes: inChn: MQue
 Traits: Queue[msg,MQue]
 Transition-Specifications:
   $R_1$ : $\langle idle, active\rangle, \langle active, active\rangle$;
       $SendA?(true);\ true \Longrightarrow inChn' = insert(A,inChn) \wedge x' = A$ ;
   $R_2$ : $\langle idle, active\rangle, \langle active, active\rangle$;
       $SendB?(true);\ true \Longrightarrow inChn' = insert(B,inChn) \wedge x' = B$ ;
   $R_3$ : $\langle active, active\rangle;\ DelvA!(len(inChn) > 1 \wedge x = A)$ ;
       $true \Longrightarrow inChn' = tail(inChn) \wedge x' = front(inChn')$;
   $R_4$ : $\langle active, active\rangle;\ DelvB!(len(inChn) > 1 \wedge x = B)$ ;
       $true \Longrightarrow inChn' = tail(inChn) \wedge x' = front(inChn')$;
   $R_5$ : $\langle active, idle\rangle;\ DelvA!(len(inChn) = 1 \wedge x = A)$ ;
       $true \Longrightarrow isempty(inChn')$;
   $R_6$ : $\langle active, idle\rangle;\ DelvA!(len(inChn) = 1 \wedge x = B)$ ;
       $true \Longrightarrow isempty(inChn')$;
end
```

Figure 26: Class Specification of OrdChannel

From Channel to OrdChannel, the only data model of the attribute is changed. The test suite does not change. Now we check if it has the same effect on the transitions' enabling condition, port condition and post condition.

- For transition R_1: $g' = g$, $a' = a$, no test needed.
- For transition R_2: $g' = g$, $a' = a$, no test needed.
- For transition R_3: $g' = g$, $a' \neq a$, but we have
  $a' = (inChn' = tail(inChn) \wedge x' = front(inChn')) \Rightarrow a = (inChn' = tail(inChn))$
- For transition R_4: $g' = g$, $a' \neq a$, but we have
  $a' = (inChn' = tail(inChn) \wedge x' = front(inChn')) \Rightarrow a = (inChn' = tail(inChn))$
- For transition R_5: $g' = g$, $a' = a$, no test needed.
- For transition R_6: $g' = g$, $a' = a$, no test needed.

The set of test cases for object Channel can be used to test object OrdChannel.

6.4 Extensional Inheritance

The goal of extensional inheritance is to provide design refinements that preserve behavior. The TROM object A_E obtained by refining a TROM object A satisfying the following constraints is an extensional inheritance of A.

- Possible Redefinition
  - Any redefinition as permitted in behavioral inheritance
  - The source state s of a transition may be redefined to any substate of s.
- Possible Addition
  - *Event addition*: New events many be added, enriching inherited port-types.
  - *State addition*: A new state may be added only to make an existing simple state as a compound state.
  - *Attribute addition*: New attributes and traits may be added.
  - *Transition addition*: An added transition can have only new events and new states. The superstate of the source state and the destination state should be the same.

– *Time-constraint addition*: An added time-constraint can only constrain a new event.

6.4.1 State Redefinition

The source state s of a transition is redefined. That is $< s, v >$ becomes a complex state, $< s_0, v_0 > \in$ *substates*$(< s, v >)$, $< s_0, v_0 > =$ *initial*$(< s, v >)$. The test suite for A_E is computed as follows.

1. The state cover of $< s, v >$ computed as part of $T_d(A)$ becomes the state cover of $< s_0, v_0 >$ in $T_d(A_E)$:

$$sc(< s, v >) = sc(< s_0, v_0 >)$$

2. If there exists a state $\theta \in \mathcal{G}_d(A)$ and the state cover $sc(\theta) \in T_d(A)$ is of the form $sc(\theta) = sc(< s, v >).p$, where p is a path from state $< s, v >$ to state θ, then replace it by the new state cover of θ, $sc(\theta) = sc(< s_0, v_0 >).p$:

$$SC(A_E)_1 = (SC(A) \setminus \{sc(< s, v >).p\}) \cup \{sc(< s_0, v_0 >).p\}$$

3. If there exists a state θ with an outgoing transition e' such that $tc(e') \in TC(A)$, $tc(e') = sc(\theta).e'$, $sc(\theta) = sc(< s, v >).p$, where p is a path from state $< s, v >$ to state θ, then replace it by the new transition cover of e', $tc(e') = sc(< s_0, v_0 >).p.e'$:

$$TC(A_E)_1 = (TC(A) \setminus \{sc(< s, v >).p.e'\}) \cup \{sc(< s_0, v_0 >).p.e'\}$$

4. For any state s_i, where $< s_i, v_i > \in$ *substates*$(< s, v >) \wedge < s_i, v_i > \neq < s_0, v_0 >$, the state cover of $< s_i, v_i >$ is $sc(< s_i, v_i >) = sc(< s, v >).p$, where p is a path from $< s_0, v_0 >$ to $< s_i, v_i >$:

$$SC(A_E)_2 = SC(A)_1 \cup \{sc(< s_i, v_i >)\}$$

5. For any new added transition, $< s_i, v_i > \xrightarrow{e} < s_j, v_j >$, where $< s_i, v_i >, < s_j, v_j > \in$ *substates*$(< s, v >)$, if $sc(< s_i, v_i >).e \notin SC(A_E)_2$, add it to $TC(A)$:

$$TC(A_E)_2 = TC(A_E)_1 \cup \{sc(< s_i, v_i >).e\}$$

6. Test suite of A_E is the union of state cover and transition cover :

$$T_d(A_E) = SC(A_E)_2 \cup TC(A_E)_2$$

The above method can be applied independently for each state refinement in A. Moreover, if a state of a refined state is refined again, the above method is repeated taking A_E as the base object.

6.4.2 Attribute Addition

There is no change for test suite, however it may involve a recalculation of the enabling condition, port condition and post condition of certain transitions.

6.4.3 Event Addition, State Addition and Transition Addition

An added transition can have only new events and only be between a pair of new states. These additions occur because a state $< s, v >$ is redefined to be a complex state, which has been discussed in section 6.4.1.

6.4.4 Time-constraint Addition

An added time-constraint can only constrain a new event. For $< s_i, v_i > \xrightarrow{e} < s_j, v_j >$, e is the new added event which is constrained to occur within the time shift $[l, u]$, where $< s_i, v_i >, < s_j, v_j > \in \mathit{substates}(< s, v >)$ and $< s, v >$ is the state to be redefined.

New time constraint can be added only to the new transitions in a state refinement. For simplicity of discussion, let us assume that only one state $\theta =< s, v >$ is refined to get A_E. We consider the state hierarchy at θ as a new automaton with encapsulation of time constraints. This is justified because new time constraints should not violate the existing time constraints in A.

We construct $\mathcal{G}_{d_1}(\theta)$, the grid automaton for the θ-machine where d_1 is the quantization determined by the number of new clocks introduced in the refinement. We compute $SC(\theta)$, the set of state covers and $TC(\theta)$, the set of transition covers for the θ-machine. We construct $T_d(A_E)$ as described below:

1. The state cover of $\theta =< s, v >$ computed as part of $T_d(A)$ becomes the state cover of $< s_0, v_0 >$ in $T_d(A_E)$:

$$sc(\theta) = sc(< s_0, v_0 >)$$

2. If there exists a state $\theta_1 \in \mathcal{G}_d(A)$ and the state cover $sc(\theta_1) \in T_d(A)$ is of the form $sc(\theta_1) = sc(\theta).p$, where p is a path from state θ to state θ_1, then replace it by the new state cover of θ_1, $sc(\theta_1) = sc(< s_0, v_0 >).p$:

$$SC(A_E)_1 = (SC(A) \setminus \{sc(\theta).p\}) \cup \{sc(< s_0, v_0 >).p\}$$

3. If there exists a state θ_1 with an outgoing transition e' such that $tc(e') \in TC(A)$, $tc(e') = sc(\theta_1).e'$, $sc(\theta_1) = sc(\theta).p$, where p is a path from state θ to state θ_1, then replace it by the new transition cover of e', $tc(e') = sc(< s_0, v_0 >).p.e'$:

$$TC(A_E)_1 = (TC(A) \setminus \{sc(\theta).p.e'\}) \cup \{sc(< s_0, v_0 >).p.e'\}$$

4. For any state $< s_i, v_i >$, where $< s_i, v_i > \in substates(\theta) \wedge (< s_i, v_i > \neq < s_0, v_0 >)$, the state cover of $< s_i, v_i >$ is $sc(< s_i, v_i >) = sc(\theta).p$, where $p \in SC(\theta)$ and p is the state cover of $< s_i, v_i >$ in θ-machine :

$$SC(A_E)_2 = SC(A)_1 \cup \{sc(< s_i, v_i >)\}$$

5. For any new transition, $< s_i, v_i > \xrightarrow{e} < s_j, v_j >$, where $< s_i, v_i >, < s_j, v_j > \in substates(\theta)$, the transition cover of e, $tc(e) = sc(< s_0, v_0 >).p$, where $p \in TC(\theta)$ and p is the transition cover of e in θ-machine : if $tc(e) \notin SC(A_E)_2$, add it to $TC(A)$:

$$TC(A_E)_2 = TC(A)_1 \cup \{tc(e)\}$$

6. Test suite of A_E is the union of state cover and transition cover :

$$T_d(A_E) = SC(A_E)_2 \cup TC(A_E)_2$$

6.4.5 Example

Figure 27 shows the class specification of a basic telephone. The events *OnHook* and *OffHook* are initiated by the user of the telephone. They occur at a port of type $@P$.

The events *Ring* and *Stop* can occur at a port of type $@Q$, to initiate interactions with a telecommunication switch. The activity related to initiation of a call goes on in the state *initCall*, while the activity related to receiving a call goes on in the state *rcvCall*. Figure 28 shows the specification of the class *Answer-Phone* obtained by decomposing the state *ringing* of the TROM class *Basic-Phone* into two states *wait* and *answer*, and adding the transition *Start* between these two new states. Once in the state *ringing*, an answer-phone waits for 2 time units and then starts giving the message before 4 units of time. The internal event can be disabled if the phone stops ringing (*idel* state is entered) or the call is received (*rcvCall* state is entered). A TROM object of the class *Answer-Phone* is an extensional inheritance of a TROM object of the class *Basic-Phone*. The behavior of *Basic-Phone* objects are preserved by the objects of the class *Answer-Phone*. However, an object of the class *Answer-Phone* cannot be substituted for an object of *Basic-Phone* in the context of its usage.

Class *Basic-Phone* [@P,@Q]
Events: *OnHook?P, OffHook?P, Ring?Q, Stop?Q*
States: **idle, ringing, rcvCall, initCall*
Transition-Specifications:
R_1 : ⟨*idle, initCall*⟩, ⟨*ringing, rcvCall*⟩;
OffHook?(true); true $\Longrightarrow$ *true*;
R_2 : ⟨*initCall, idle*⟩, ⟨*rcvCall, idle*⟩;
OnHook?(true); true $\Longrightarrow$ *true*;
R_3 : ⟨*idle, ringing*⟩; *Ring?(true); true* $\Rightarrow$ *true*;
R_4 : ⟨*ringing, idle*⟩; *Stop?(true); true* $\Rightarrow$ *true*;
end

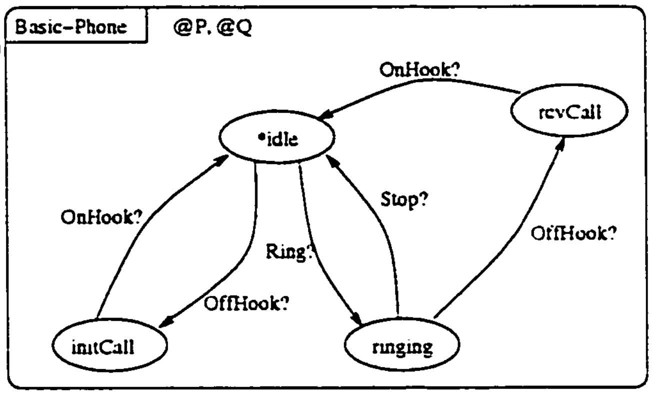


Figure 27: Class Specification of Basic-Phone

There is no time constraint event in the class specification of *Basic-Phone*. Hence, the grid automaton and TROM are identical. The set of test cases for the object from the class *Basic-Phone* T_d(Basic-Phone) includes the state cover and transition cover:

```
state cover SC:        { OffHook?,
                         Ring?,
                         Ring?.OffHook? }


transition cover TC:   { OffHook?.OnHook?,
                         Ring?.Stop?
```

Class *Answer-Phone* [@P,@Q]
Inherits: *Basic-Phone*
Events: *Start*
States: *ringing(*wait,answer)*
Transition-Specifications:
R_5 : ⟨*wait*, *answer*⟩;
Start(true); *true* $\Longrightarrow$ *true*;
Time-constraints:
(R_3, *Start*, [2,4], {*rcvCall,idle*})
end

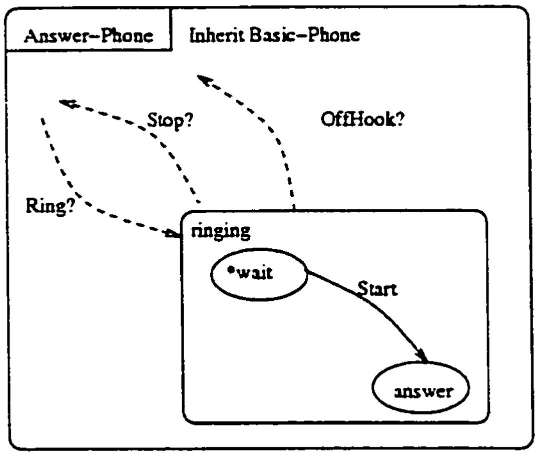


Figure 28: Class Specification of Answer-Phone: Enhancement

```
Ring?.OffHook?.OnHook? }
```

Figure 28 shows the refinement of the derived object Answer-Phone by refining the state *ringing*. The state *ringing* becomes a complex state, including two substates: *waiting* and *answering*; the new transition added between these two states is labeled by the time constrained event *Start*, enforcing that it has to occur within 2 to 4 time unit.

Figure 29 is the grid automaton of *ringing*-machine. State *ringing* becomes a complex state, including substates *wait* and *answer*. All the original transitions whose destination state is *ringing* will be changed to the transitions whose destination state is $< wait, 0 >$, the initial state of *ringing*. All the original transition whose source state is *ringing* will be changed to the transitions whose source states are $< wait, v >$, where $v \in \{0, 1/2, 2/2, 3/2, 4/2, 5/2, 6/2, 7/2, 8/2\}$.

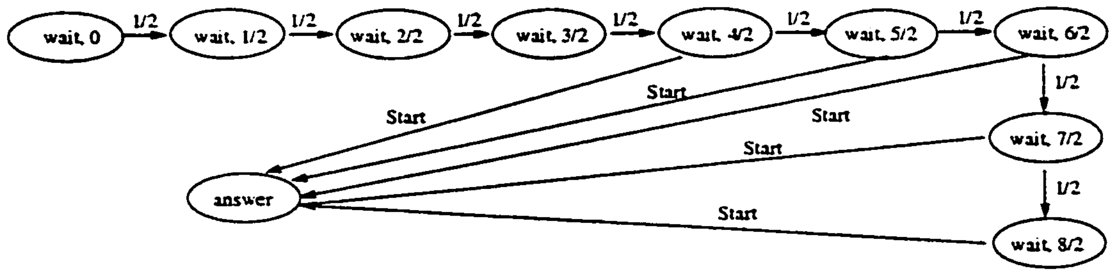


Figure 29: Grid Automaton of *ringing*-machine

The test cases for the *ringing*-machine are the following:

```
state cover SC(ringing):
```

```
{
     1/2,
     1/2.1/2,
     1/2.1/2.1/2,
     1/2.1/2.1/2.1/2,
     1/2.1/2.1/2.1/2.1/2,
     1/2.1/2.1/2.1/2.1/2.1/2,
     1/2.1/2.1/2.1/2.1/2.1/2.1/2,
     1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2,
     1/2.1/2.1/2.1/2.Start    }

transition cover TC(ringing):
{
     1/2.1/2.1/2.1/2.1/2.Start
     1/2.1/2.1/2.1/2.1/2.1/2.Start
     1/2.1/2.1/2.1/2.1/2.1/2.1/2.Start
     1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.Start
}
```

The initial state of the grid automaton *ringing* is $< wait, 0 >$, and its state cover is the state cover of *ringing*. The state cover of any other new added state is the state cover of $< wait, 0 >$, concatenated with the state cover of this state. For instance,

$$sc(< wait, 2/2 >) = sc(< wait, 0 >).sc_{ringing}(< wait, 2/2 >)$$

where $sc(< wait, 0 >) = Ring?$, $sc_{ringing}(< wait, 2/2 >) = 1/2.1/2$, therefore,

$$sc(< wait, 2/2 >) = Ring?.1/2.1/2$$

The transition cover of any new added transition is the state cover of $< wait, 0 >$, concatenated with the transition cover of this transition. For instance,

$tc(< wait, 4/2 > \overset{Start}{\rightarrow} < answer, v' >) =$

$\qquad sc(< wait, 0 >).tc_{ringing}(< wait, 4/2 > \overset{Start}{\rightarrow} < answer, v >)$

$\qquad = Ring?.1/2.1/2.1/2.1/2.Start$

where $v' \in \{4/2, 5/2, 6/2, 7/2, 8/2\}$.

6.5 Polymorphic Inheritance

The TROM object A_P obtained by refining a TROM object A satisfying the following constraints is a polymorphic inheritance of A.

- Possible Redefinition
  - Any redefinition as permitted in Behavioral inheritance.
- Possible Additions
  - *Port Addition*: New port-types may be added. This necessary introduces new events.
  - *State addition*: New states may be added without decomposing any atomic state.
  - *Attribute addition*: New attributes and traits may be added.
  - *Transition addition*:
    - An added transition can be labeled either by a new event or old event. However any new internal event should be strictly between new states.
    - An added internal transition can only have newly added states for both source and destination states.
  - *Time-constraint addition*: An added time-constraint can only constrain a new event and can only be triggered by a new transition.

6.5.1 Attribute Addition

There is no change in the set of test suites, however a recalculation of the enabling condition, port condition and post condition of certain transitions may be necessary.

6.5.2 Other Additions

Let A_P be a polymorphic inherited object of A. We consider the part of the machine A_P consisting of the newly introduced states, the new transitions (may be labeled by either new or old events) and those states in A that can be a source or destination of new transitions, call this machine Rem_P. We construct the grid automaton

$\mathcal{G}_{d_1}(Rem_P)$ and compute the set $SC(Rem_P)$, the set of state covers for Rem_P, and the set $TC(Rem_P)$, the set of transition covers for Rem_P. We justify the construction of new grid automaton $\mathcal{G}_{d_1}(Rem_P)$ based on the facts that new time constraints introduce new clocks, and should not interfere with the existing time constraints in A. We construct the set $T_d(A_P)$ as follows:

1. The initial state of the new grid automaton $\mathcal{G}_{d_1}(Rem_P)$ should be the state in A that can be a source state of the new transitions. If there is more than one such state, we arbitrarily choose one of them as the source state.

2. Construct the grid automaton $\mathcal{G}_{d_1}(Rem_P)$ and compute the set $SC(Rem_P)$, the set of state covers for Rem_P, and the set $TC(Rem_P)$, the set of transition covers for Rem_P.

3. For any new added state θ, the state cover of θ is the state cover of θ_0, concatenated with the state cover of θ in machine Rem_P, where θ_0 is a state in A, and the initial state of grid automaton $\mathcal{G}_{d_1}(Rem_P)$:

$$SC(A_P)_1 = SC(A_P) \cup \{sc(\theta) = sc(\theta_0).sc_{Rem_P}(\theta)\}$$

4. For any new added transition r, if $tc_{Rem_P}(r) \in TC(Rem_P)$, then the transition cover of r is the state cover of θ_0, concatenated with the transition cover of r in machine Rem_P, where θ_0 is a state in A, and the initial state of grid automaton $\mathcal{G}_{d_1}(Rem_P)$:

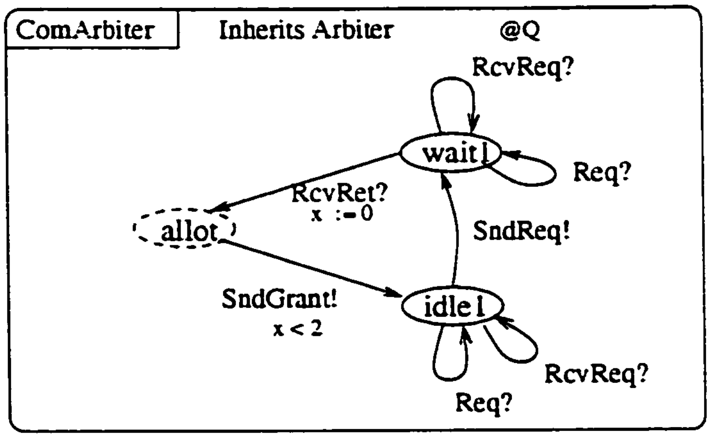
$$TC(A_P)_1 = TC(A_P) \cup \{tc(r) = sc(\theta_0).tc_{Rem_P}(r) \mid tc_{Rem_P}(r) \in TC(Rem_P)\}$$

5. The test suite for A_P is the union of the set of state covers and the set of transition covers :

$$T_d(A_P) = SC(A_P)_1 \cup TC(A_P)_1$$

6.5.3 Example

As an example of this kind of inheritance consider the arbiter class *Arbiter* shown in Figure 7. Objects of *Arbiter* class do not communicate among themselves. The class obtained from *Arbiter* class, respecting polymorphic refinements, is the class *ComArbiter* shown in Figure 30. Arbiter objects of the refined class communicate



Class *ComArbiter* [@Q]

Inherits: *Arbiter*

Events: $SndGrant!Q, RcvRet?Q, SndReq!Q, RcvReq?Q$

States: $idle1, wait1$

Attributes: $rqQueue, rqAQueue : UQueue,\ atPrt : @Q$

Attribute-function:

$idle1, wait1 \mapsto \{rqQueue, rqAQueue, atPrt\};\ allot, wait \mapsto \{atPrt\};$

Transition-Specifications:

$R_3 :< allot, allot >, < wait, wait >, < idle1, idle1 >,$
$< wait1, wait1 >;\ Req?(\bar{}(\text{pid} \in rqQueue));$
$true \Longrightarrow rqQueue' = insert(\text{pid}, rqQueue) \wedge equal(rqAQueue', rqAQueue);$

$R_6 :< allot, idle1 >;\ SndGrant!(\text{pid} \in rqAQueue);$
$true \Longrightarrow atPrt' = \text{pid} \wedge equal(rqQueue', rqQueue) \wedge equal(rqAQueue', tail(rqAQueue));$

$R_7 :< wait1, allot >;\ RcvRet?(\text{pid} = atPrt);$
$true \Longrightarrow equal(rqQueue', rqQueue) \wedge equal(rqAQueue', rqAQueue);$

$R_8 :< idle1, wait1 >;\ SndReq!(\text{pid} = atPrt);$
$true \Longrightarrow equal(rqQueue', rqQueue) \wedge equal(rqAQueue', rqAQueue);$

$R_9 :< idle, allot >, < allot, allot >, < wait, wait >, < idle1, idle1 >, < wait1, wait1 >;$
$RcvReq?(\bar{}(\text{pid} \in rqAQueue));$
$true \Longrightarrow equal(rqQueue', rqQueue) \wedge rqAQueue' = insert(\text{pid}, rqAQueue);$

Time-Constraints:

$TC_3 : (R_7, Grant, [0, 2], \{wait1\})$

$TC_4 : (R_7, SndGrant, [0, 2], \{wait\})$

end

Figure 30: Class Specifications of the Communicating Arbiter

through the events *SndGrant*!, *SndReq*!, *RcvReq*?, and *RcvRet*? at the ports of the new port type @Q to request and return resource owned by them. Whenever a resource requested by a client of arbiter a is not available with a but is available and not in use with some client of arbiter b, arbiter a can request for the resource and get it from arbiter b. In order to specify this requirement, the class specification of *ComArbiter* includes the new transitions R_6, R_7, R_8, R_9, and a modification of the *Arbiter* transition R_3. The new states *idle*1 and *wait*1, and two new timing constraints are associated with the new transitions. The new attribute *rqAQueue* is the list of port identifiers at which requests from other arbiters are received. The attribute *atPrt* denotes the port at which the resource was last granted through the communication channel. In state *allot* an arbiter can allocate a resource to one of it clients or can grant access to the resource requested by another arbiter. In state *idle*1, an arbiter object does not hold the resource nor does any of the users associated with it hold a resource. If it needs the request, it sends the message *SndReq* to an arbiter and waits in state *wait*1. As soon as receiving the message *RcvRet* signaling that the resource has been returned by another arbiter, it changes its state to *allot*, making the enabling condition for the transition *Grant* to true, thereby allocating the resource for one of its clients. An object from *ComArbiter* class can be substituted for an object from *Arbiter* class in the latter's context. In such use, no port of type @Q will be linked, blocking the newly introduced events from occurring.

We construct the grid automaton for the machine $Rem_P(ComArbiter)$ as in Figure 31. The state $< allot, 0 >$ is the initial state of grid automaton $\mathcal{G}_{d_1}(Rem_P)$. The test cases for the TROM $Rem_P(ComArbiter)$ are the following:

```
state cover SC:
{
  1/2,
  1/2.1/2,
  1/2.1/2.1/2,
  1/2.1/2.1/2.1/2,
  SndGrant!,
  SndGrant!.SndReq! }

transition cover TC:
```

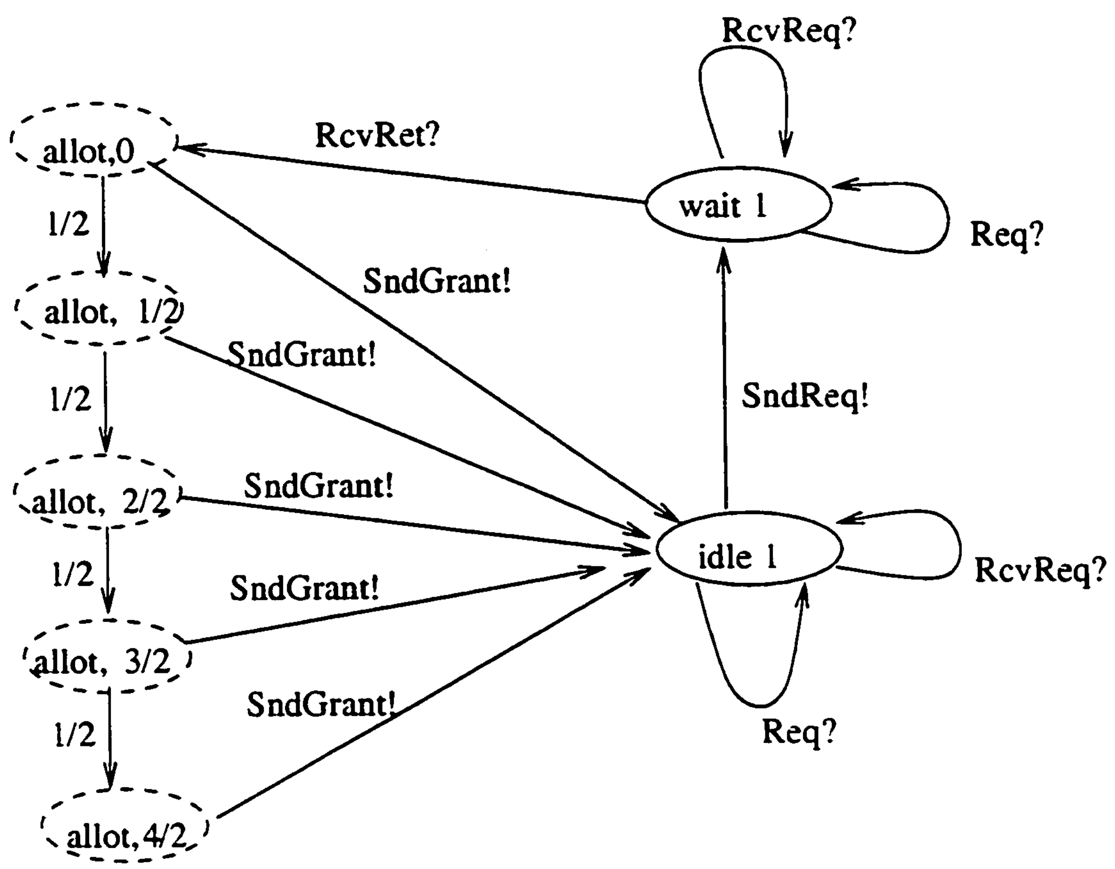


Figure 31: Grid Automaton of Rem_P

```
{
  1/2.SndGrant!,
  1/2.1/2.SndGrant!,
  1/2.1/2.1/2.SndGrant!,
  1/2.1/2.1/2.1/2.SndGrant!
  SndGrant!.Req?,
  SndGrant!.RcvReq?,
  SndGrant!.SndReq!.Req?,
  SndGrant!.SndReq!.RcvReq?
  SndGrant!.SndReq!.RcvRet?   }
```

The state cover of $< allot, 0 >$ is the state cover of state *allot*; the state cover of any new added state θ is the state cover of state $< allot, 0 >$, concatenated with the state cover θ in the grid automaton of Rem_P. For instance:

$$sc(< idle1, v >) = sc(< allot, 0 >).sc_{Remp}(< idle1, v >)$$

where $v \in \{0, 1/2, 2/2, 3/2, 4/2\}$.

If there exists a new added transition r, $tc_{Remp}(r) \in TC(Rem_P)$, then the transition cover of r is the state cover of state $< allot, 0 >$, concatenated with the transition cover of r in the grid automaton of Rem_P. For instance:

$$tc(< idle1, v > \overset{SndReq!}{\rightarrow} < wait1, v >) = sc(< allot, 0 >).tc_{Remp}(< idle1, v > \overset{SndReq!}{\rightarrow} < wait1, v >)$$

where $v \in \{0, 1/2, 2/2, 3/2, 4/2\}$.

6.6 Parameterized Events and General Inheritance

In this section we discuss two extensions: one deals with parameterized events, and the other delas with general inheritance.

6.6.1 Parameterized Events

A recent extension of TROM is PTROM [Hay01], where in events with typed parameters are allowed to occur in a reactive unit. PTROM allows constant as well as variable parameters. The test case generation approach remains the same for such instances. The main difference is that the set $T_d(A)$ will contain test cases as well as *test templates*. A test template will include one or more parameterized events. For instance, if the event *Near(uid)* is part of a test template, and *uid* is a constant parameter, different test cases are obtained by instantiating the parameter *uid* with values from the domain associated with its type. If the type of *uid* is *Nat*, and the domain is $\{100, 200, 300\}$, then different test cases involving *Near(100), Near(200), Near(300)* will be generated from the test template involving *Near(uid)*. However, if *uid* is a variable parameter whose values are determined by the postconditions in transition specifications, the test template can only be injected as is or with an initial value for *uid* as provided by the specification. More general situations will arise when a template includes events with several parameters as well as several parameterized events. The key features of a test template are that it is

- *generic*, i.e., it represents a family of input,

- *abstract*, i.e., it is extracted from the specification and can be tailored to any particular implementation, and
- *instantiable*, i.e., there is some representation for the defined class of input values to the parameters,

Test templates can be arranged in a hierarchy and partitioned to meet the goals of testing. Since all tests must be derived from the valid input space of the parameters, the valid input space can serve as the starting point of a hierarchy. Once the test strategy determines the valid input space, it can then partition the valid input space into *domains* of interest. Choice of domains is not determined by the test case generation algorithms. Rather, testing strategies and some heuristics must be used to determine the domains. To meet the objective of testing, one must derive domains which are equivalence classes of error-detecting ability for the program under test, and which cover the valid input space. So, domains must be chosen so that each element of the domain has the same error-detecting capability, and hence the result of using one element in testing applies equally to all elements in the domain. Our observation on domain partitioning for parameter space, is equally valid for events from the environment that are stimuli to the reactive system. These remarks are given here to motivate an implementor of our methods in a practical setting.

6.6.2 General Inheritance

We can incrementally combine the three methods discussed in the previous sections to generate test cases for an object which is derived by combining the three kinds of inheritance. If A is a base object from which A_E, an extensional inheritance of A is derived first and then $(A_E)_P$, a polymorphic inheritance is derived next, then the test set $T_d((A_E)_P)$ is obtained incrementally, generating $T_d(A_E)$ from $T_d(A)$ and then $T_d((A_E)_P)$ from $T_d(A_E)$. In general, if there is a sequence of refinements of A to get an object A_f, where each step in the sequence is one of three basic inheritance, then there exists a sequence of test sets $T_d(A), \ldots, T_d(A_f)$, such that each member in the sequence is derived from previous member by applying a method appropriate to the inheritance. Below we illustrate this approach to derive the test cases of a generalized Arbiter.

A simple version of Arbiter example has been discussed earlier. Now we discuss a version of the Arbiter model in which events have parameters. An Arbiter allocates a resource to processes requesting them. Upon sending a request, a User is granted the resource which it has to return within 20 time units. A user cannot hold more than one resource, and it can send a request to the Arbiter one at a time. The Arbiter class has one port type *@U* through which it receives the messages *Req?* and *Ret?*, and outputs the message *Grant!*. It has one data type *p* of type integer, two parameters *uid* and *rid* of type integer, and three abstract data types: *Queue*, *Set*, and *Pair* necessary for its functioning.

The statechart diagram of the simple Arbiter system is presented in Figure 32. The formal specifications of the *Arbiter* is shown in Figure 33.

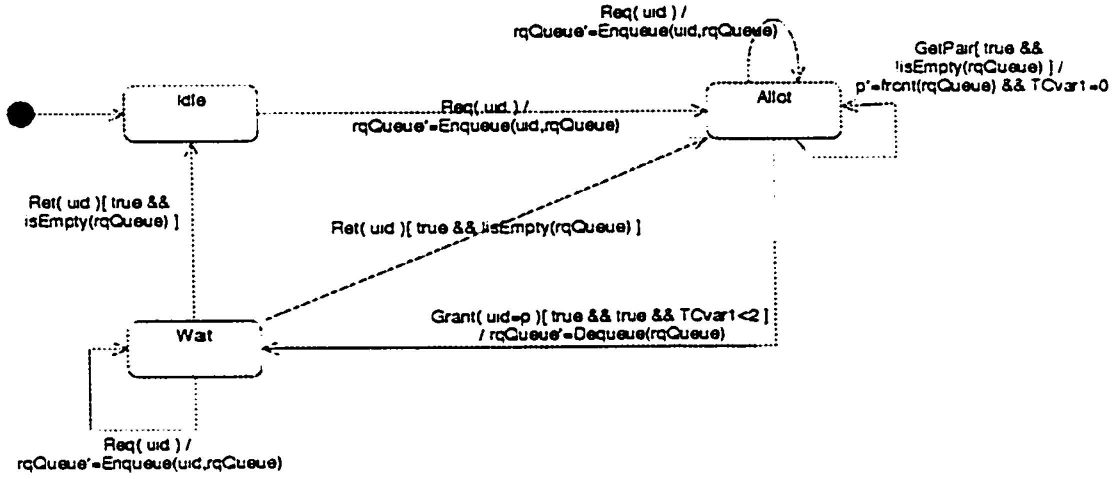

Req(uid) /
rqQueue'=Enqueue(uid,rqQueue)
GetPair[true &&
!isEmpty(rqQueue)] /
p'=front(rqQueue) && TCvar1=0
Idle
Allot
Req(uid) /
rqQueue'=Enqueue(uid,rqQueue)
Ret(uid)[true &&
isEmpty(rqQueue)]
Ret(uid)[true && !isEmpty(rqQueue)]
Grant(uid=p)[true && true && TCvar1<2]
/ rqQueue'=Dequeue(rqQueue)
Wat
Req(uid) /
rqQueue'=Enqueue(uid,rqQueue)

Figure 32: Arbiter State Diagram

The design of a generic Arbiter is based on the following considerations. An Arbiter allocates multi-type resources to processes requesting them. At any instant, a process can simultaneously hold and request more than one resource. To request a resource, the process (User) sends the message *Req* to the Arbiter. The Arbiter can receive requests from different users. In order to remember the User to whom the next resource must be granted, the Arbiter maintains a queue in which the pair of User ID and request ID is stored. The users are serviced on a first–in–first–out basis. Also, the Arbiter has a set where the Ids of available requests are stored. Whenever a request is granted to a User, the request ID is deleted from the Set. If the request ID is not in the Set, the Arbiter sends the message *NotAvailable* to the User requesting it. To keep track of the granted requests and the users holding them, the Arbiter maintains another set where the pairs of granted User IDs and request IDs are stored.

```
Class Arbiter [@U]
Events: Req?@U, Ret?@U, GetPair, Grant!@U
States: *Idle, Wait, Allot
Attributes: rqQueue:UQueue; uid:Integer; p:Integer
Traits: Queue[Integer, UQueue]
Attribute-Function: Idle → {uid};Wait → {rqQueue, uid};Allot → {rqQueue, p, uid};
Parameter-Specifications:
      Grant: uid;
      Req: uid;
      Ret: uid;
Transition-Specifications:
      R1: <Idle,Allot>; Req[uid](true); true ⟹ rqQueue'=Enqueue(uid,rqQueue);
      R2: <Wait,Idle>; Ret[uid](true); isEmpty(rqQueue) ⟹ true;
      R3: <Wait,Wait>; Req[uid](true); true ⟹ rqQueue'=Enqueue(uid,rqQueue);
      R4: <Wait,Allot>; Ret[uid](true); !isEmpty(rqQueue) ⟹ true;
      R5: <Allot,Allot>; Req[uid](true); true ⟹ rqQueue'=Enqueue(uid,rqQueue);
      R6: <Allot,Allot>; GetPair[](true); !isEmpty(rqQueue) ⟹ p'=front(rqQueue);
      R7: <Allot,Wait>; Grant[uid=p](true); true ⟹ rqQueue'=Dequeue(rqQueue);
Time-Constraints:
      TCvar1: R6, Grant, [0, 2), {};
end
```

Figure 33: Formal Specification for GRC Arbiter-Simple

The User returns a resource by sending the message *Ret* to the Arbiter. Then, the resource ID is inserted back into the set of available resource Ids and the pair of granted User ID and resource ID is deleted from the set of granted resources. To guaranty that all the users are served, a time condition constrains the granting event of the Arbiter. The *Grant* message should be sent within 2 time units after retrieving the request from the queue.

The Arbiter has one port type *@U* through which it receives the messages *Req?* and *Ret?*, and outputs the messages *Grant!* and *NotAvailable!*. It includes one data type *p* of type pair, two parameters *uid* and *rid* of type integer, and three abstract data types: *Queue, Set*, and *Pair* necessary for its functioning. The state machine associated with the Arbiter is shown in Figure 34. The state of the Arbiter is determined by the values of 2 invariants. The first is *rqQueue*, a queue where the pairs of requesting user Ids and request Ids are stored. The second is *grSet*, a set where the pair of granted User Ids and request Ids are stored. The possible combinations of the values of these attributes are represented in Table 1. For instance, if the Arbiter is in state *S1*, then *rqQueue* and *grSet* are empty. It means that the Arbiter is in an idle state where it does not have any request to grant, and all the resources are available and returned by users if any were granted. If the Arbiter is in *S2*, then *rqQueue* not empty and *grSet* is empty. This is interpreted as the Arbiter is in a state where it has outstanding requests in the queue to be granted but still none is served. The formal specifications of the Arbiter is shown in Figure 36.

		EMPTY		NON-EMPTY
S1		rqQueue, grSet		
S2		grSet		rqQueue
S3				grSet, rqQueue
S4		rqQueue		grSet

Table 1: Possible States of the Arbiter Class

The generic Arbiter is obtained incrementally from simple Arbiter. First, by polymorphic inheritance, we add the new event *NotAvailable*! between the old states in the simple Arbiter. Second, by extensional inheritance, we refine the state *Wait* to include two substates: S_3, and S_4. Within the refined state *Wait* new transition *NotAvailable*! is added between the new states, and a new time constraint is added. Third, by behavioral inheritance, the data model and some old transitions are refined. Therefore, the generic Arbiter object is obtained by $((A_P)_E)_B$. The test set of generic Arbiter is correspondingly obtained by incrementally combining the methods for the three steps of inheritance.

1. Compute $T_d(A)$: Construct the grid automaton for the state machine A and compute $T_d(A)$, the set of test cases for the simple arbiter A.

2. Polymorphic Inheritance: Construct the grid automaton for the newly added part to the state machine A. Let A_P denote the new state machine. Following the method discussed in Section 6.5 modify the set $T_d(A)$ to recalculate the test set $T_d(A_P)$.

3. Extensional Inheritance: Construct the grid automaton for *Wait*-machine. Let A_{P_E} denote the new machine obtained by the state refinement from A_P. Following the discussion in Section 6.4, incrementally modify the set of test cases $T_d(A_P)$ to get the set $T_d(A_{P_E})$.

4. Behavioral Inheritance: Denote by $A_{P_{E_B}}$ the new machine obtained by behavioral inheritance of the machine A_{P_E}. Following the discussion in Section 6.3 recalculate the set of test cases $T_d(A_{P_{E_B}})$.

The set $T_d(A_{P_{E_B}})$ will include test templates involving the parameterized events in the arbiter specification. The justification for the order in which we carried out the refinement is that new time constraints should not violate the old time constraints.

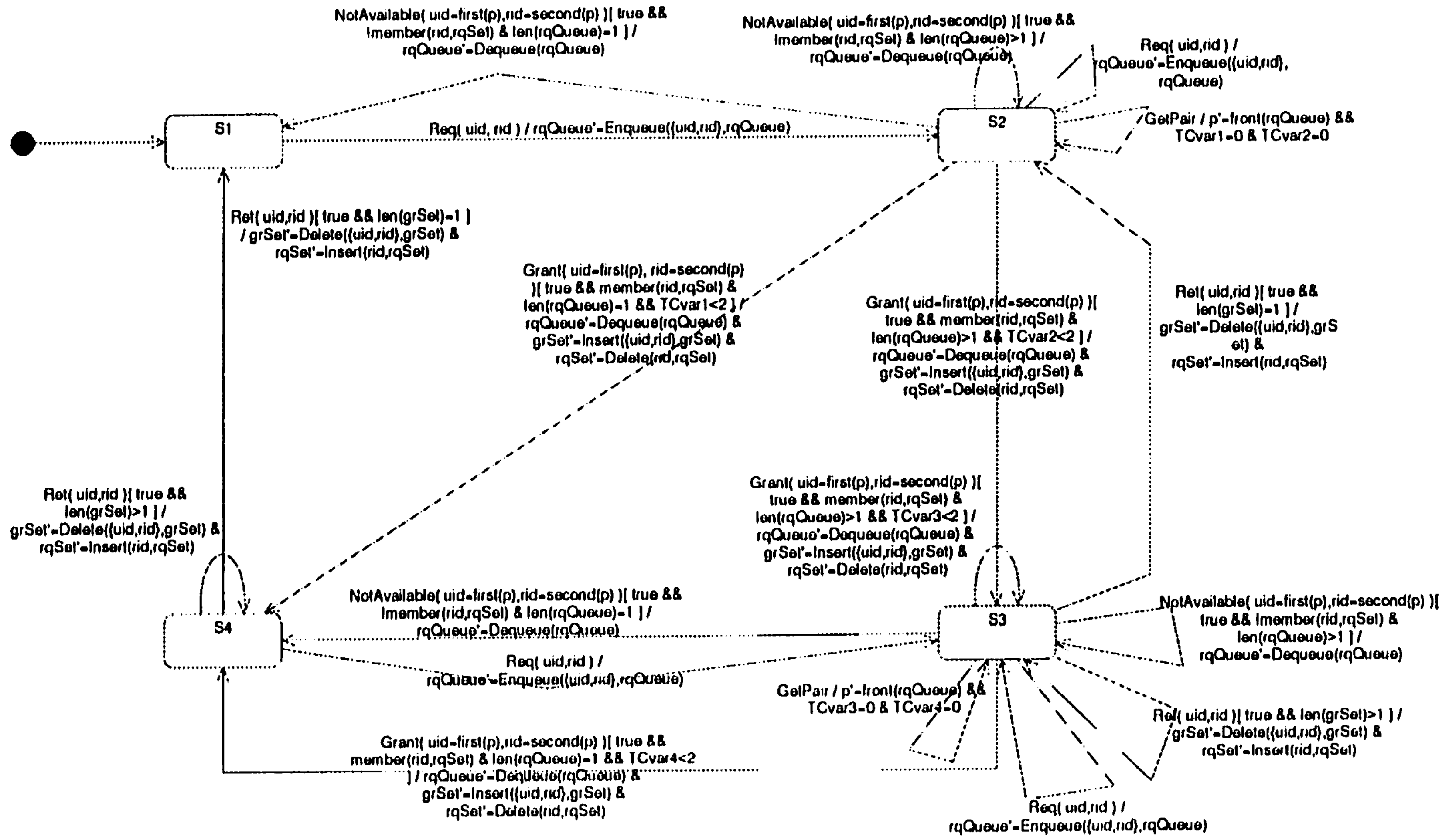

S1
S2
S3
S4
NotAvailable(uid=first(p),rid=second(p))[true && !member(rid,rqSet) & len(rqQueue)=1] / rqQueue'=Dequeue(rqQueue)
Req(uid, rid) / rqQueue'=Enqueue((uid,rid),rqQueue)
NotAvailable(uid=first(p),rid=second(p))[true && !member(rid,rqSet) & len(rqQueue)>1] / rqQueue'=Dequeue(rqQueue)
Req(uid,rid) / rqQueue'=Enqueue((uid,rid), rqQueue)
GetPair / p'=front(rqQueue) && TCvar1=0 & TCvar2=0
Ret(uid,rid)[true && len(grSet)=1] / grSet'=Delete((uid,rid),grSet) & rqSet'=Insert(rid,rqSet)
Grant(uid=first(p), rid=second(p))[true && member(rid,rqSet) & len(rqQueue)=1 && TCvar1<2] / rqQueue'=Dequeue(rqQueue) & grSet'=Insert((uid,rid),grSet) & rqSet'=Delete(rid,rqSet)
Grant(uid=first(p),rid=second(p))[true && member(rid,rqSet) & len(rqQueue)>1 && TCvar2<2] / rqQueue'=Dequeue(rqQueue) & grSet'=Insert((uid,rid),grSet) & rqSet'=Delete(rid,rqSet)
Ret(uid,rid)[true && len(grSet)=1] / grSet'=Delete((uid,rid),grSet) & rqSet'=Insert(rid,rqSet)
Ret(uid,rid)[true && len(grSet)>1] / grSet'=Delete((uid,rid),grSet) & rqSet'=Insert(rid,rqSet)
Grant(uid=first(p),rid=second(p))[true && member(rid,rqSet) & len(rqQueue)>1 && TCvar3<2] / rqQueue'=Dequeue(rqQueue) & grSet'=Insert((uid,rid),grSet) & rqSet'=Delete(rid,rqSet)
NotAvailable(uid=first(p),rid=second(p))[true && !member(rid,rqSet) & len(rqQueue)=1] / rqQueue'=Dequeue(rqQueue)
Req(uid,rid) / rqQueue'=Enqueue((uid,rid),rqQueue)
NotAvailable(uid=first(p),rid=second(p))[true && !member(rid,rqSet) & len(rqQueue)>1] / rqQueue'=Dequeue(rqQueue)
GetPair / p'=front(rqQueue) && TCvar3=0 & TCvar4=0
Ret(uid,rid)[true && len(grSet)>1] / grSet'=Delete((uid,rid),grSet) & rqSet'=Insert(rid,rqSet)
Grant(uid=first(p),rid=second(p))[true && member(rid,rqSet) & len(rqQueue)=1 && TCvar4<2] / rqQueue'=Dequeue(rqQueue) & grSet'=Insert((uid,rid),grSet) & rqSet'=Delete(rid,rqSet)
Req(uid,rid) / rqQueue'=Enqueue((uid,rid),rqQueue)

Figure 34: Arbiter StateChart Diagram

```
Class Arbiter [@U]
Events: Req?@U, Ret?@U, Grant!@U, NotAvailable!@U, GetPair
States: *S1, S2, S3, S4
Attributes: rqQueue: UQueue; rqSet:USet; grSet:GSet; p:pPair;
      uid:Integer; rid:Integer
Traits: Queue[pPair, UQueue],Set[Integer, USet],Set[Integer,GSet],
      Pair[Integer, Integer, pPair]
Attribute-Function: S1 → {grSet, rqSet, uid, rid};S2 → {rqQueue, grSet, rqSet, p, uid, rid};
      S3 → {rqQueue, grSet, rqSet, p, uid, rid};S4 → {rqQueue, uid, rid, grSet, rqSet};
Parameter-Specifications:
      Grant: uid, rid;
      NotAvailable: uid, rid;
      Req: uid, rid;
      Ret: uid, rid;
Transition-Specifications:
      R1: <S1,S2>; Req[uid,rid](true); true ⟹
            rqQueue'=Enqueue(uid,rid,rqQueue);
      R2: <S4,S1>; Ret[uid,rid](true); len(grSet)=1 ⟹
            grSet'=Delete(uid,rid,grSet) & rqSet'=Insert(rid,rqSet);
      R3: <S4,S3>; Req[uid,rid](true); true ⟹
            rqQueue'=Enqueue(uid,rid,rqQueue);
      R4: <S4,S4>; Ret[uid,rid](true); len(grSet)>1 ⟹
            grSet'=Delete(uid,rid,grSet)& rqSet'=Insert(rid,rqSet):
      R5: <S2,S4>; Grant[uid=first(p),rid=second(p)](true);
            memeber(rid,rqSet) & len(rqQueue)=1 ⟹
            rqQueue'=Dequeue(rqQueue) & grSet'=Insert({uid,rid},grSet) & rqSet'=Delete(rid,rqSet):
      R6: <S2,S3>; Grant[uid=first(p),rid=second(p)](true);
            member(rid,rqSet) & len(rqQueue)>1 ⟹
            rqQueue'=Dequeue(rqQueue) & grSet'=Insert({uid,rid},grSet) & rqSet'=Delete(rid,rqSet):
      R7: <S2,S1>; NotAvailable[uid=first(p),rid=second(p)](true);
            !member(rid,rqSet) & len(rqQueue)=1 ⟹ rqQueue'=Dequeue(rqQueue);
      R8: <S2,S2>; NotAvailable[uid=first(p),rid=second(p)](true);
            !member(rid,rqSet) & len(rqQueue)>1 ⟹ rqQueue'=Dequeue(rqQueue);
      R9: <S2,S2>; Req[uid,rid](true); true ⟹
            rqQueue'=Enqueue(uid,rid,rqQueue);
      R10: <S2,S2>; GetPair[](true); true ⟹
            p'=front(rqQueue);
      R11: <S3,S4>; NotAvailable[uid=first(p),rid=second(p)](true);
            !member(rid,rqSet) & len(rqQueue)=1 ⟹ rqQueue'=Dequeue(rqQueue);
```

Figure 35: Formal Specification for GRC Arbiter

```
    R12: <S3,S2>; Ret[uid,rid](true); len(grSet)=1 ⟹
          grSet'=Delete({uid,rid},grSet) & rqSet'=Insert(rid,rqSet);
    R13: <S3,S3>; NotAvailable[uid=front(p),rid=second(p)](true);
          !member(rid,rqSet) & len(rqQueue)>1 ⟹ rqQueue'=Dequeue(rqQueue);
    R14: <S3,S3>; Ret[uid,rid](true); len(grSet)>1 ⟹
          grSet'=Delete({uid,rid},grSet) & rqSet'=Insert(rid,rqSet);
    R15: <S3,S3>; Req[uid,rid](true); true ⟹
          rqQueue'=Enqueue(uid,rid,rqQueue);
    R16: <S3,S3>; GetPair[](true); true ⟹
          p'=Dequeue(rqQueue);
    R17: <S3,S3>; Grant[uid=first(p),rid=second(p)](true);
          member(rid,rqSet) & len(rqSet)¿1 ⟹
          grSet'=Insert({uid,rid},grSet) & rqSet'=Delete(rid,rqSet) & rqQueue'=Dequeue(rqQueue);
    R18: <S3,S4>; Grant[uid=first(p),rid=second(p)](true);
          member(rid,rqSet) & len(rqSet)=1 ⟹
          grSet'=Insert({uid,rid},grSet) & rqSet'=Delete(rid,rqSet) & rqQueue'=Dequeue(rqQueue);
Time-Constraints:
    TCvar1: R10, Grant, [0, 2), {};
    TCvar2: R10, Grant, [0, 2), {};
    TCvar3: R16, Grant, [0, 2), {};
    TCvar4: R16, Grant, [0, 2), {};
end
```

Figure 36: Formal specification for GRC Arbiter-Continued

Chapter 7

System Testing - I

7.1 Introduction

This chapter discusses the testing of a simple system where only a pair of objects interact. We construct the synchronous product machine of the two interacting objects. We use the algorithms described in Chapter 5 to derive a grid automaton and generate test cases from the grid automaton.

7.2 Pair Testing: Testing a pair of Objects

In TROM formalism, there is no communication among the objects from the same class. Therefore, pairs of interacting objects are from different classes. We can compose the synchronous product machine of the two interacting objects, use Algorithm GA to derive the grid automaton for the synchronous product machine, and use Algorithm TC to generate the test cases from the grid automaton.

The set of clocks in the synchronous product machine is the union of the set of clocks of the individual machines. Consequently, if k_1 and k_2 are the number of clocks of the individual machines, the quantization of the grid automata corresponding to the synchronous machine is $\frac{1}{k_1+k_2}$. Notice that the number of clock regions increases for large values of k_1 and k_2, thus forcing the grid automata size (vertices and edges) to increase. Such an increase is acceptable because it only increases the effectiveness of test covers to test the behavior in a finer level of timeliness.

7.2.1 Algorithm for Constructing the Synchronous Product Machine

The algorithm, starting from the initial state of the product machine, will compute only those states that are reachable due to either internal transitions or synchronous transitions. The algorithm terminates when no new state is added to the set of states in the product machine, and timing conflicts are removed.

Timing Analysis

Before we introduce the Algorithm SP for constructing the synchronous product machine, we present the definitions of the temporal predicates for time intervals used in TROM. It is the basis to do timing analysis in the algorithm SP.

We introduce a domain for time intervals in TROM, and define the temporal predicates shown in Figure 37 to express relations on time intervals. The equality predicate *Equal* is symmetric; the predicates *Before*, *Meet*, *Overlaps*, *During*, *Starts*, and *Finishes* are asymmetric. Thus, for a pair of time intervals, there are thirteen relations expressible by these predicates and their inverses. More complex temporal relations on time intervals can be expressed in terms of these predicates. Table 2 gives the definitions of the temporal predicates on time intervals. The variables T_1 and T_2 range over the domain of time intervals; $T_1 = [u_1, v_1]$, $v_1 > u_1$, and $T_2 = [u_2, v_2]$, $v_2 > u_2$, where u_1, u_2, v_1, v_2 are non-negative rationals.

$Before(T_1, T_2)$	$\triangleq$	$v_1 < u_2$
$Meet(T_1, T_2)$	$\triangleq$	$v_1 = u_2$
$Overlaps(T_1, T_2)$	$\triangleq$	$u_1 < u_2 < v_1 < v_2$
$Equal(T_1, T_2)$	$\triangleq$	$u_1 = u_2 \wedge v_1 = v_2$
$During(T_1, T_2)$	$\triangleq$	$u_2 < u_1 \wedge v_1 < v_2$
$Starts(T_1, T_2)$	$\triangleq$	$u_1 = u_2 \wedge v_1 < v_2$
$Finishes(T_1, T_2)$	$\triangleq$	$u_1 < u_2 \wedge v_1 = v_2$

Table 2: Definitions of the Temporal Predicates

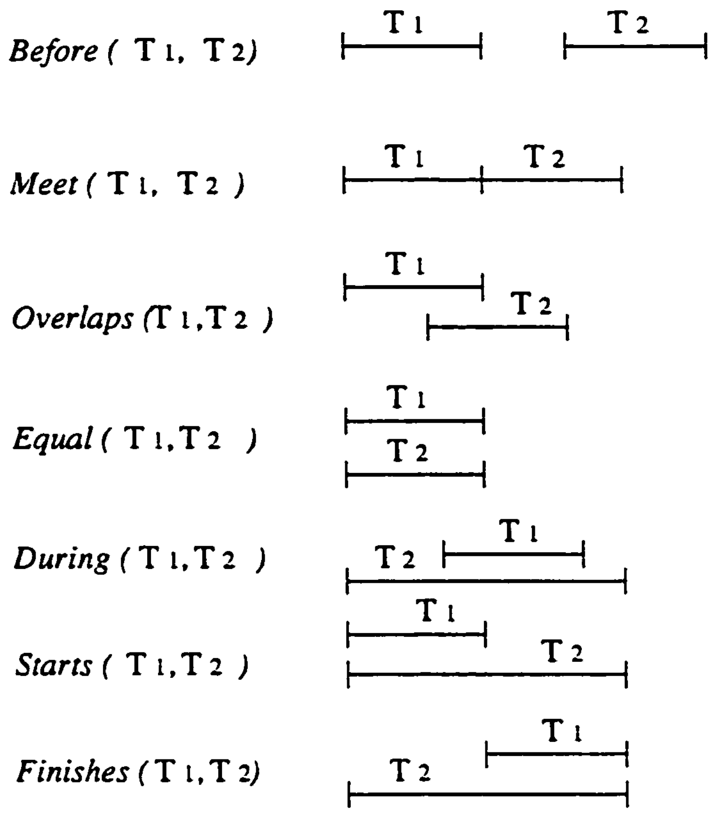


Figure 37: Temporal Predicates

Algorithm SP

Input: Finite state machines $M_1 = (S_1, E_1, T_1)$ and $M_2 = (S_2, E_2, T_2)$, $E_1 \cap E_2 =$ set of shared events.

Output: Synchronous Product Machine $M = M_1 \otimes M_2$, where $M = (S, E, T)$, $S \subseteq \{(s_i, s'_i) \mid s_i \in S_1, s'_i \in S_2\}$, $E = E_1 \cup E_2$. $\forall (s_i, s'_i) \in S$. It has at least one incoming or outgoing transition.

Step 1 Initialization

Step 1.1 NU: set of states which are still to be explored:

$$NU = \{(s_0, s'_0) \mid s_0 \in S_1,\ s'_0 \in S_2\}$$

Step 1.2 S: set of states in the product machine. These are the states that have been reached by transitions.

$$S = \emptyset$$

Step 1.3 T: set of transitions.

$$T = \emptyset$$

Step 2 While $(NU \neq \emptyset)$ do the following steps

Step 2.1 Pick an element (s_i, s'_i) from NU, compute $NU = NU - \{(s_i, s'_i)\}$ and $S = S \cup \{(s_i, s'_i)\}$.

Rename clock variables if necessary so that all the names are distinct.

Step 2.2 For each shared event e that occurs at s_i and s'_i do:

if $(s_i \xrightarrow{e} s_j)$ and $(s'_i \xrightarrow{e} s'_j)$, then $NU = NU \cup \{(s_j, s'_j)\}$, $T = T \cup \{(s_i, s'_i) \xrightarrow{e} (s_j, s'_j)\}$. The guard of the transition $(s_i, s'_i) \xrightarrow{e} (s_j, s'_j)$, is the conjunction of the guard of $s_i \xrightarrow{e} s_j$ and the guard of $s'_i \xrightarrow{e} s'_j$. The action of the transition $(s_i, s'_i) \xrightarrow{e} (s_j, s'_j)$, is the union of the action of $s_i \xrightarrow{e} s_j$ and the action of $s'_i \xrightarrow{e} s'_j$.

Step 2.3 For each internal event e occurring at s_i do:

if $(s_i \xrightarrow{e} s_j)$, then $NU = NU \cup \{(s_j, s'_i)\}$, $T = T \cup \{(s_i, s'_i) \xrightarrow{e} (s_j, s'_i)\}$. The guard and action of transition $(s_i, s'_i) \xrightarrow{e} (s_j, s'_i)$, is the guard and action of $s_i \xrightarrow{e} s_j$.

Step 2.4 For each internal event e occurring at s'_i do:

if $(s'_i \xrightarrow{e} s'_j)$, then $NU = NU \cup \{(s_i, s'_j)\}$, $T = T \cup \{(s_i, s'_i) \xrightarrow{e} (s_i, s'_j)\}$. The guard and action of transition $(s_i, s'_i) \xrightarrow{e} (s_i, s'_j)$, is the guard and action of $s'_i \xrightarrow{e} s'_j$.

End of While loop

Step 3 Conduct a static analysis of time constraints and remove transitions that are in conflict. For each state s in the product machine do the following steps:

Step 3.1 If the outgoing transitions in state s are not time-constrained do nothing.

Step 3.2 If the outgoing transitions in state s are constrained by different clock variables that are not initialized at the same instant, then do nothing.

Step 3.3 If there exists $k \geq 1$ outgoing transitions in state s that are constrained by the same clock variable or by different clock variables that are initialized at the same instant (in the action part of a transition specification), then resolve the conflict as follows:

Let $[a_1, b_1], [a_2, .b_2], \ldots, [a_k, b_k]$ be the time intervals for the time constraints on the k transitions. Determine r for $1 \leq r \leq k$ for which the predicate *before*$([a_r, b_r], [a_j, b_j])$ holds for all $j = 1, \ldots, k$, $j \neq r$. Retain the transition at s which is constrained by the clock variable in the interval $[a_r, b_r]$, and remove the rest of the transitions at s.

Step 4 Remove states(except the initial state) in the product machine that have no incoming transition. These states are unreachable in any execution.

End of Algorithm

7.3 Example

Based on the algorithm, we construct the synchronous product machine of train object and controller object, and the synchronous product of controller object and gate object. These are shown in Figure 38 and Figure 39. We have constructed the grid automaton for these machines. Then we use the Algorithm TC to generate test cases for the interacting behaviors of train and controller objects as well as controller and gate objects.

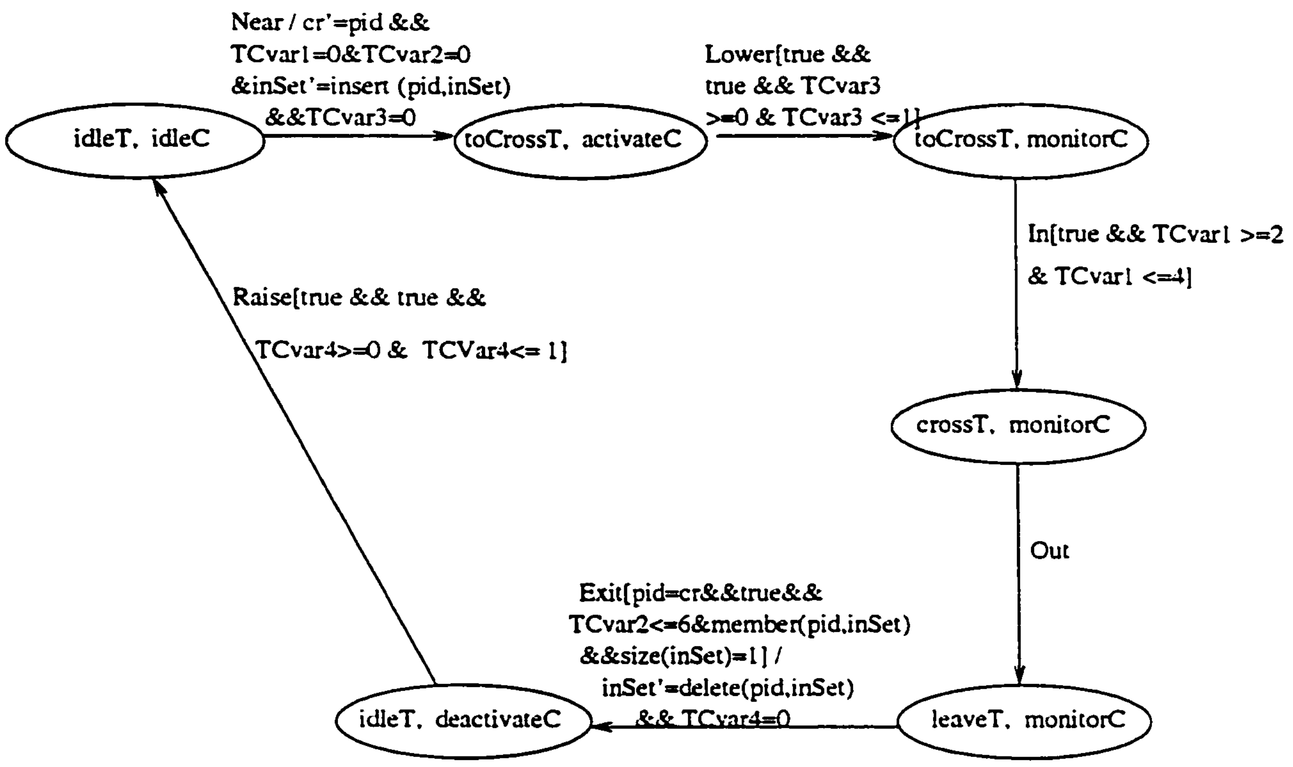


Figure 38: Synchronous Product of Train and Controller

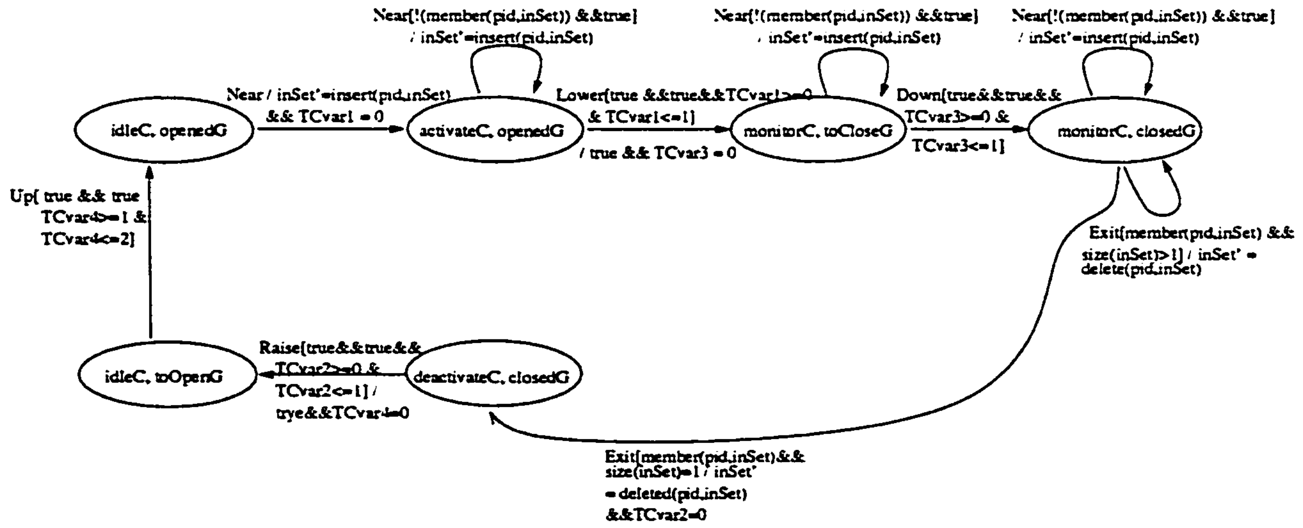


Figure 39: Synchronous Product of Controller and Gate

Chapter 8

System Testing - II

8.1 Introduction

A system is a collection of interacting objects. An object-oriented software system can be considered as a single object. This object, call it *system object*, has a state associated with it. The behavior of a system object is contingent upon its initial state. The state of a system object is the aggregate state of its inherent constituents.

System testing is a broad term. It is a type of testing exercised during system configuration of software development. The objective of system testing is to determine whether the configured software system is ready for its intended users by observing its behavior.

In this chapter we discuss a method to test a system with more than two interacting objects. In general, system testing is inherently complex. It is impossible to construct a synchronous product machine for a large system. Also we would like to reuse the test cases from unit testing and pair testing for system testing. Instead of generating test cases for the system from scratch, we use the test cases generated for unit testing, and test cases used for testing included subsystems to construct the test cases for the whole system.

The reactive objects instantiated from the same class and included in a system configuration do not communicate. Consequently, the graph abstraction of a system configuration is a connected bipartite graph, which can be partitioned into subgraphs where each subgraph is maximal bipartite graph: one vertex is connected to every other vertex in the graph. We call such maximal subgraph a component. For instance, Figure 40 shows a system configured with four objects from GRC_1 and three objects from GRC_2. The components are $\{O_1, O'_2, O_4\}$, $\{O'_1, O_2\}$, and $\{O_3, O'_3\}$. The proposed testing method obtains such components. Two components are either independent or interact through messages exchanged by a common object of the components. We discuss a test method that computes the test cases for a component by using the test cases generated for pairs of interacting objects in the component.

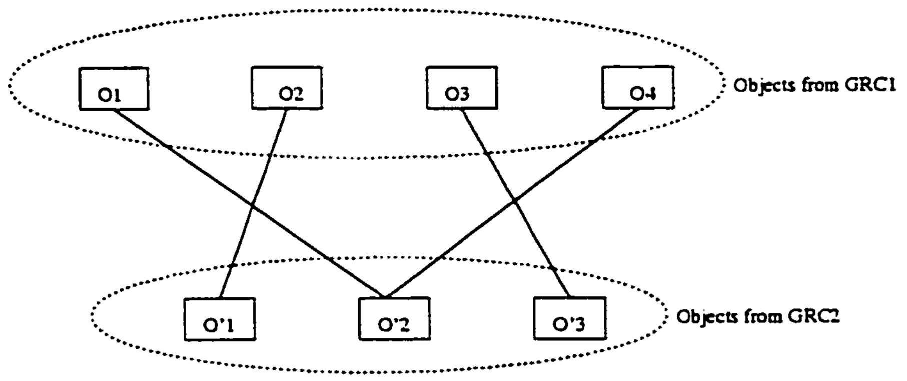


Figure 40: An Example of System Configuration

8.2 Synchronous Product for a System

In Chapter 7 we discussed an algorithm for the construction of a synchronous product machine of two finite state machines communicating with shared events. Since shared events can only happen between two machines, in order to construct the synchronous product machine of more than two finite state machines, we must construct the synchronous product transitively. For instance, if M_1, M_2, M_3 are three finite state machines, M_1 and M_2 communicate with shared events, and M_2 and M_3 communicate with shared events, then we can construct the synchronous product machine of M_1, M_2 and M_3 in three ways:

1. $(M_1 \otimes M_2) \otimes M_3$;

2. $M_1 \otimes (M_2 \otimes M_3)$;

3. $(M_1 \otimes M_2) \otimes (M_2 \otimes M_3)$

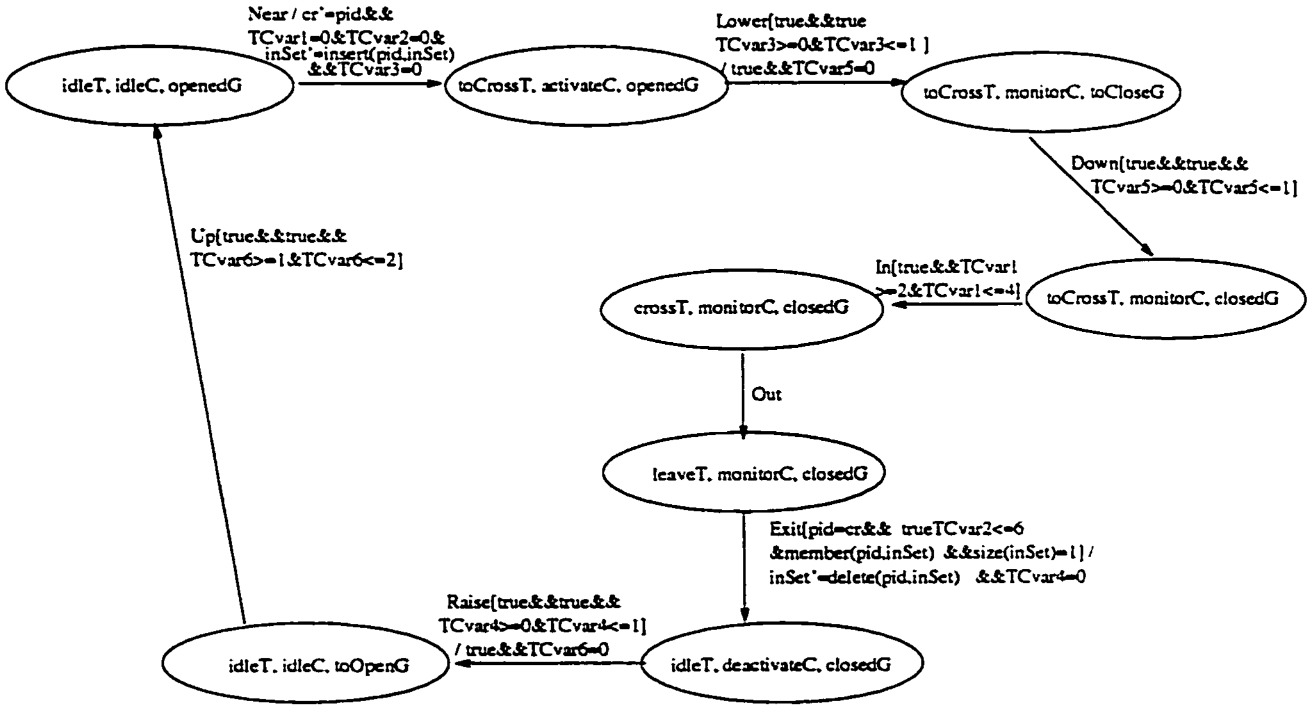
Synchronous Product Theorem:

If M_1, M_2, M_3 are three finite state machines, where M_1 and M_2 communicate with shared events, M_2 and M_3 communicate with shared events, then $(M_1 \otimes M_2) \otimes M_3 = M_1 \otimes (M_2 \otimes M_3) = (M_1 \otimes M_2) \otimes (M_2 \otimes M_3)$.

Proof:

The proof follows from the algorithm for constructing the synchronous product machine.

Figure 41 shows the machine TCG = ((train $\otimes$ controller) $\otimes$ gate). We also generated (train $\otimes$ (controller $\otimes gate$)), as well as ((train $\otimes$ controller) $\otimes$ (controller $\otimes$ gate)) and verified that they are identical to the machine shown in Figure 41.



idleT, idleC, openedG
Near / cr'=pid&& TCvar1=0&TCvar2=0& inSet'=insert(pid,inSet) &&TCvar3=0
toCrossT, activateC, openedG
Lower[true&&true TCvar3>=0&TCvar3<=1] / true&&TCvar5=0
toCrossT, monitorC, toCloseG
Down[true&&true&& TCvar5>=0&TCvar5<=1]
toCrossT, monitorC, closedG
In[true&&TCvar1 >=2&TCvar1<=4]
crossT, monitorC, closedG
Out
leaveT, monitorC, closedG
Exit[pid=cr&& trueTCvar2<=6 &member(pid,inSet) &&size(inSet)=1] / inSet'=delete(pid,inSet) &&TCvar4=0
idleT, deactivateC, closedG
Raise[true&&true&& TCvar4>=0&TCvar4<=1] / true&&TCvar6=0
idleT, idleC, toOpenG
Up[true&&true&& TCvar6>=1&TCvar6<=2]

Figure 41: (Train $\otimes$ Controller) $\otimes$ Gate

The states of the machine TCG are the global states in the system where the controller monitors one train and allows it to cross the gate controlled by it. Every transition in the machine TCG is labeled by an event belonging to one of the machines or shared by two machines. If there are two trains in the system, then we construct the synchronous product machine of T_1CG and T_2CG, where T_iCG is the synchronous product machine for the $train_i$, controller and gate. We can generalize to any number of trains: if there are n trains in the system, then we construct the synchronous product machine $T_1CG \otimes T_2CG \otimes \ldots \otimes T_iCG \ldots \otimes T_nCG$, where $i = 1, 2, \ldots, n$, T_iCG is the synchronous product machine for the $train_i$, controller and gate.

A naive approach to system testing is to construct the synchronous product of the system, derive its grid automaton and then generate the state covers and transition covers for the grid automaton. This approach has three drawbacks:

- the grid automaton of the synchronous product machine (involving 3 or 4 machines) will be huge, especially when the number of clocks in the system modeled by the product machine is high; and
- there is a potential for exponential growth in the generation of the test case sequencing, making it impractical to test the system satisfactorily.
- the test case generated during the unit testing phase can not be used for system testing.

In order to reduce the complexity in testing phase, we want to make use of the test cases generated for each object in the system and incrementally generate test cases for the system under test. Towards this goal, we partition the system architecture, such that the test cases for the whole system can be generated from the test cases of the individual components in the partition.

8.3 Partition Algorithm

The architecture of the system under test consists of objects and communication links between objects that interact in the system. In general, the graph abstracted from a given architecture is a connected graph. A *vertex cover* of a graph is a subset of the vertices of the graph so that every vertex of the graph is adjacent to at least one vertex in the vertex cover. The minimal vertex cover is a vertex cover not contained in any other vertex cover. If we can determine a minimal vertex cover C for the graph $G =< V, E >$ abstracted from the reactive system architecture, then C satisfies the following properties:

- $\forall o_i, o_j \in C \bullet < o_i, o_j > \notin E$
- $\forall o \in C,\ \Pi_o = \{o' \mid o' \in V \land < o, o' > \in E\} \cup \{o\}$

Since every edge in E is a portlink, a communication channel, the collection $\Pi = \{\Pi_o \mid o \in C\}$ gives a *partition* of the system architecture with respect to C: all objects in $\Pi_o - \{o\}$ communicate with o and hence form a connected *interaction component.* Hence, testing each Π_o in isolation, and putting them together to test the full system is justified.

The vertex cover problem is to find a vertex cover of minimal size in G. In general, finding the minimal vertex cover is NP-hard. However, an efficient greedy algorithm can find a vertex cover that is close to optimum. In fact, for many of the architectures that we have come across the following algorithm works very well. The complexity of the greedy algorithm is $\Theta(n)$.

Greedy Algorithm

```
Input: an undirected graph $G = (V, E)$
Output: a vertex cover set $C$
While ($E \neq \emptyset$) {
    1. $\forall v \in V$, Calculate $d(v)$, the degree of vertex $v$.
    2. choose a vertex $w$ of maximum degree.
        • $C \leftarrow C \cup \{w\}$
        • $V \leftarrow V - \{w\}$
        • $E \leftarrow E - \{(w, x) \mid (x \in V) \land ((w, x) \in E)\}$
}
Return $C$
```

For instance, for the graph in Figure 42, the greedy algorithm produces the vertex cover $C = \{1, 4, 5\}$. The partition with respect to C is Π_1, Π_4, Π_5, where $\Pi_1 = \{1, 3\}$, $\Pi_4 = \{2, 3, 4, 6\}$, and $\Pi_5 = \{2, 5, 6\}$.

For example, consider the railroad crossing problem in Figure 17. It is the architecture for a Train-Gate-Controller system with 5 trains, 2 controllers, and 2 gates. Three trains intending to cross one gate communicate with the controller associated

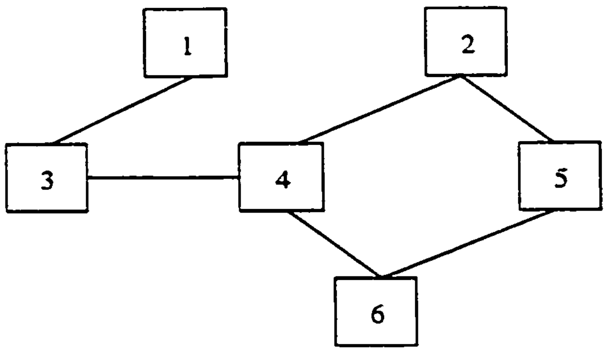


Figure 42: A General Architecture

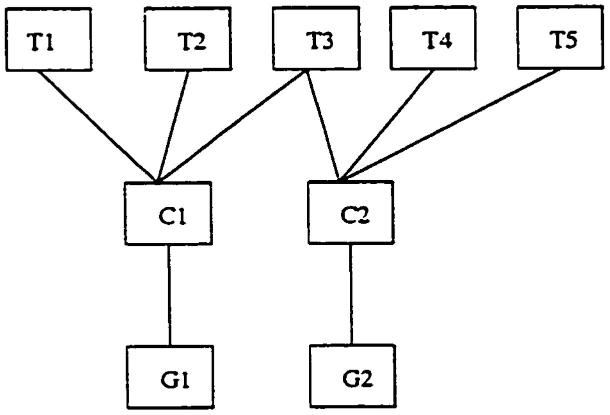


Figure 43: The Architecture of Train-Gate-Controller System

to that gate. We can transform the architecture into a graph whose vertices are objects (ports) and edges correspond to portlinks. The graph corresponding the system in Figure 17 is shown in Figure 43.

From Figure 43, the greedy algorithm produces the vertex cover set $C = \{C1, C2\}$, which is optimal. The derived components are : $\Pi_1 = \{T1, T2, T3, C1, G1\}$ and $\Pi_2 = \{T3, T4, T5, C2, G2\}$. However, $T1$, $T2$, and $T3$ in Π_1 are the objects from the same class, hence, the component Π_1 need to be further partitioned into $\Pi_{11} = \{T1, C1, G1\}$, $\Pi_{12} = \{T2, C1, G1\}$, $\Pi_{13} = \{T3, C1, G1\}$. Similarly, the component Π_2 need to be further partitioned into $\Pi_{21} = \{T3, C2, G2\}$, $\Pi_{22} = \{T4, C2, G2\}$, $\Pi_{23} = \{T5, C2, G2\}$.

8.4 Test Cases for Components in a Partition

All the objects in one component are from different classes. They have a strong interaction, in the sense that every object in the component interacts with the core object in the vertex cover. For instance, if $\Pi = \{o_1, o_2, \ldots, o_k\}$ is a component, and o_1 is in the vertex cover, then every object o_i, $i \neq 1$ interacts with o_1. To test the component Π, we claim it is sufficient to generate the test set for all pairs of objects (o_i, o_1) and put the tests together using a *grid-synchronous product* to get test cases for the component Π.

8.4.1 Grid-synchronous Product

In this section, we define the *grid-synchronous product*, $gs_{\otimes}$, for the test cases generated from two grid automata $\mathcal{G}_d(A)$ and $\mathcal{G}_d(B)$, where A and B are the synchronous product machines. Since $T_d(A) \subset \mathcal{G}_d(A)$, the grid-synchronous product notion applies to grid automata as well.

Let us assume that the test sets $T_d(A)$ and $T_d(B)$ have already been generated from the grid automata $\mathcal{G}_d(A)$ and $\mathcal{G}_d(B)$. Each test case in $T_d(A)$ or in $T_d(B)$ is either a state cover or a transition cover in their respective synchronous product machines' grid automata and their homomorphic liftings are state or transition covers for the corresponding synchronous product machines. It is our goal to construct the test set $T_d(A \otimes B)$, the set of test cases for testing the components composed by interacting pair of synchronous product machines A and B, without explicitly constructing $\mathcal{G}_d(A \otimes B)$. The construction is motivated by the following considerations.

In Figure 44, H_A denotes the homomorphism between $\mathcal{G}_d(A)$ and A, H_B denotes the homomorphism between $\mathcal{G}_d(B)$ and B, and H_{AB} denotes the homomorphism between $\mathcal{G}_d(A \otimes B)$ and $A \otimes B$. If $\tau_A \in T_d(A)$ is a state cover for the state $\theta_A \in \mathcal{G}_d(A)$, then $H_A(\tau_A) = \sigma_A$, where $\sigma_A \in Comp(A)$ covers the state s_A of A, and $H_A(\theta_A) = s_A$. Similarly, if $\tau_B \in T_d(B)$ is a state cover for the state $\theta_B \in \mathcal{G}_d(B)$, then $H_B(\tau_B) = \sigma_B$, where $\sigma_B \in Comp(B)$ covers the state s_B of B, and $H_B(\theta_B) = s_B$. The grid-synchronous product $\tau_{AB} = \tau_A \, gs_{\otimes} \, \tau_B$ of τ_A and τ_B defines a state cover for the state $\theta_{AB} \in \mathcal{G}_d(A \otimes B)$ only if

- $H_{AB}(\theta_{AB}) = \langle s_A, s_B \rangle$ is a state of the synchronous product machine $A \otimes B$, and
- $H_{AB}(\tau_{AB}) \in H_A(\tau_A) \otimes H_B(\tau_B)$

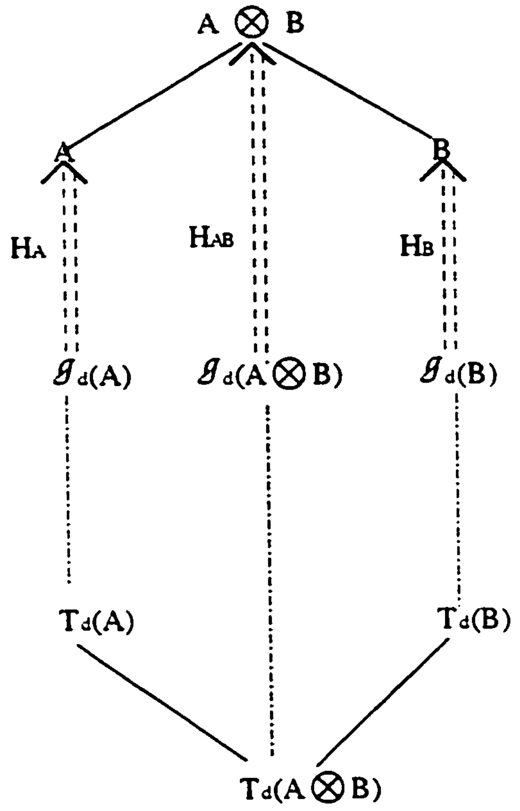


Figure 44: Principle of Grid-Synchronous Product

Note that the grid-synchronous product definition is given implicitly. A similar definition for transition cover can be given.

Theorem GS

If $\tau_A \in T_d(A)$ covers the state $\theta_A \in \mathcal{G}_d(A)$, and $\tau_B \in T_d(B)$ covers the state $\theta_B \in \mathcal{G}_d(B)$, then $\tau_A\, gs_{\otimes}\, \tau_B$ is a state cover for some state in $\mathcal{G}_d(A \otimes B)$.

Proof:

Let $\tau = \tau_A\, gs_{\otimes}\, \tau_B$. By the construction of $\mathcal{G}_d(A \otimes B)$ and the Homomorphism Theorem, for every state $\theta = (s_A, s_B) \in A \otimes B$, where $H_A(\theta_A) = s_A$, and $H_B(\theta_B) = s_B$, there exists a state $\theta' \in \mathcal{G}_d(A \otimes B)$ such that $H_{AB}(\theta') = \theta$. Moreover, $H_A(\tau_A) = \sigma_A$ covers the state s_A of A and $H_B(\tau_B) = \sigma_B$ covers the state s_B of B. Hence the state θ_{AB} defined by the first part of the grid-synchronous product definition is θ'. In the synchronous product machine $\sigma_A \otimes \sigma_B$ there exists at least one path from its initial state to the state θ. The second part of the grid-synchronous product definition maps τ to one of them. That is, $H_{AB}(\tau)$ is a state cover for the state θ, the homomorphic map of the state θ'. By Homomorphism Theorem it follows that τ is a state cover for the state θ'.

8.4.2 Constructing Grid-synchronous Product

In this section we give an algorithm to construct a $gs_{\otimes}$ product of test cases. Let $\tau_A \in T_d(A)$ and $\tau_B \in T_d(B)$ be test cases for which we want to compute $\tau_{AB} = \tau_A \, gs_{\otimes} \, \tau_B$. We assume that both A and B are synchronous product machines, and they can synchronize on some shared events.

Let τ_A cover the state $<(s_{i_1}, s_{i_2}), v_1>$ of $\mathcal{G}_d(A)$ and τ_B cover the state $<(s'_{j_1}, s'_{j_2}), v'_2>$ of $\mathcal{G}_d(B)$ respectively. Let the homomorphic image of $<(s_{i_1}, s_{i_2}), v_1>$ be the state $<\theta_1, \theta_2>$ of the machine A and the homomorphic image of $<(s'_{j_1}, s'_{j_2}), v'_2>$ be the state $<\theta'_1, \theta'_2>$ of the machine B. If $\theta_2 = \theta'_1$, then there exists the state $<\theta_1, \theta_2, \theta'_2>$ in the synchronous product machine $A \otimes B$. Under this condition, there exists a path in $A \otimes B$ that covers the state $<\theta_1, \theta_2, \theta'_2>$. Consequently, by Homomorphism Theorem there must exist a state cover for the state $<(s_{i_1}, s_{i_2}, s'_{j_2}), v>$ in $\mathcal{G}_d(A \otimes B)$, where $H_{AB}(<(s_{i_1}, s_{i_2}, s'_{j_2}), v>) = <\theta_1, \theta_2, \theta'_2>$. One such state cover is computed by the following algorithm.

Algorithm GSP

Input: τ_A and τ_B, where $\tau_A \in T_d(A)$ and $\tau_B \in T_d(B)$ satisfying the above conditions. Both A and B are synchronous product machines. τ_A is a test sequence starting from the initial state θ_0 of $\mathcal{G}_d(A)$, and ending at state θ_k: $\theta_0 = <(s_{0_1}, s_{0_2}), v_0>$, $\theta_k = <(s_{k_1}, s_{k_2}), v_k>$. τ_B is a test sequence starting from the initial state of θ'_0 of $\mathcal{G}_d(B)$, and ending at state θ'_p; $\theta'_0 = <(s'_{0_1}, s'_{0_2}), v'_0>$, $\theta'_p = <(s'_{p_1}, s'_{p_2}), v'_p>$.

Output: Grid-synchronous product $\tau_A \, gs_{\otimes} \, \tau_B$ that is a test cover for the state $<(s_{k_1}, s_{k_2}, s'_{p_2}), v'_k>$ in $\mathcal{G}(A \otimes B)$, for some v'_k to be determined.

Assumption: $s_{0_2} = s'_{0_1}$; $s_{k_2} = s'_{p_1}$. The quantization of A is equal to the quantization of B. $v_0 = v'_0$.

Step 1: Initialization

- NU: set of states which are still to be explored:

$$NU = \{<(s_{0_1}, s_{0_2}, s'_{0_2}), v_0> \mid <(s_{0_1}, s_{0_2}), v_0> \in \Theta_A, \ <(s_{0_2}, s'_{0_2}), v_0> \in \Theta_B\}$$

- Θ: set of states in the grid-synchronous product machine.

  $\Theta = \emptyset$

- T: set of transitions.

  $T = \emptyset$

- *found*: a boolean flag which becomes true only if the test cover for the state $< (s_{k_1}, s_{k_2}, s'_{p_2}), v_k >$ is found.

  $$found = false$$

Step 2: while $(NU \neq \emptyset \wedge !found)$ do the following steps

- Pick an element $< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i >$ from NU
- $NU = NU - \{< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i >\}$
- $\Theta = \Theta \cup \{< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i >\}$
- For each shared event e that occurs at $< (s_{i_1}, s_{i_2}), v_i >$ and $< (s_{i_2}, s'_{i_2}), v_i >$ do:

  if $(< (s_{i_1}, s_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}), v_i >)$ and $(< (s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_2}, s'_{j_2}), v_i >)$, then $NU = NU \cup \{< (s_{j_1}, s_{j_2}, s'_{j_2}), v_i >\}$, $T = T \cup \{< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}, s'_{j_2}), v_i >\}$. The guard of the transition $< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}, s'_{j_2}), v_i >$, is the conjunction of the guard of $< (s_{i_1}, s_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}), v_i >$ and the guard of $< (s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_2}, s'_{j_2}), v_i >$. The action of the transition $< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}, s'_{j_2}), v_i >$, is the union of the action of $< (s_{i_1}, s_{i_2}), v_i > \xrightarrow{e} < (s_{j_1}, s_{j_2}), v_i >$ and the action of $< (s_{i_2}, s'_{i_2}), v_i > \xrightarrow{e} < (s_{j_2}, s'_{j_2}), v_i >$. If $(s_{j_1} = s_{k_1}) \wedge (s_{j_2} = s_{k_2}) \wedge (s'_{j_2} = s'_{p_2}) \wedge (v_i = v_k)$, then $found = true$, return the test sequence starting from the initial state $< (s_{0_1}, s_{0_2}, s'_{0_2}), v_0 >$, and ending at the state $< (s_{j_1}, s_{j_2}, s'_{j_2}), v_i >$.

- For d that occurs at both $< (s_{i_1}, s_{i_2}), v_i >$ and $< (s_{i_2}, s'_{i_2}), v_i >$ do:

  $NU = NU \cup \{< (s_{i_1}, s_{i_2}, s'_{i_2}), v_j >\}$, where $v_j = v_i + d$, $T = T \cup \{< (s_{i_1}, s_{i_2}, s'_{i_2}), v_i > \xrightarrow{d} < (s_{i_1}, s_{i_2}, s'_{i_2}), v_j >\}$. If $(s_{i_1} = s_{k_1}) \wedge (s_{i_2} = s_{k_2}) \wedge (s'_{i_2} = s'_{p_2}) \wedge (v_j = v_k)$, then $found = true$, return the test sequence starting from the initial state $< (s_{0_1}, s_{0_2}, s'_{0_2}), v_0 >$, and ending at the state $< (s_{i_1}, s_{i_2}, s'_{i_2}), v_j >$.

- For each internal event e occurring at $<(s_{i_1}, s_{i_2}), v_i>$ do:
  if ($<(s_{i_1}, s_{i_2}), v_i> \xrightarrow{e} <(s_{j_1}, s_{j_2}), v_i>$, then $NU = NU \cup \{<(s_{j_1}, s_{j_2}, s'_{i_2}), v_i>\}$, $T = T \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{j_1}, s_{j_2}, s'_{i_2}), v_i>\}$. The guard and action of transition $<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{j_1}, s_{j_2}, s'_{i_2}), v_i>$, is the guard and action of $<(s_{i_1}, s_{i_2}), v_i> \xrightarrow{e} <(s_{j_1}, s_{j_2}), v_i>$. If $(s_{j_1} = s_{k_1}) \wedge (s_{j_2} = s_{k_2}) \wedge (s'_{i_2} = s'_{p_2}) \wedge (v_i = v_k)$, then *found* $=$ *true*, return the test sequence starting from the initial state $<(s_{0_1}, s_{0_2}, s'_{0_2}), v_0>$, and ending at the state $<(s_{j_1}, s_{j_2}, s'_{i_2}), v_i>$.
- For d occurring at $<(s_{i_1}, s_{i_2}), v_i>$ do:
  $NU = NU \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>\}$, where $v_j = v_i + d$, $T = T \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{d} <(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>\}$. If $(s_{i_1} = s_{k_1}) \wedge (s_{i_2} = s_{k_2}) \wedge (s'_{i_2} = s'_{p_2}) \wedge (v_j = v_k)$, then *found* $=$ *true*, return the test sequence starting from the initial state $<(s_{0_1}, s_{0_2}, s'_{0_2}), v_0>$, and ending at the state $<(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>$.
- For each internal event e occurring at $<(s_{i_2}, s_{i'_2}), v_i>$ do:
  if ($<(s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{j_2}, s'_{j_2}), v_i>$), then $NU = NU \cup \{<(s_{i_1}, s_{j_2}, s'_{j_2}), v_i>\}$, $T = T \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{i_1}, s_{j_2}, s'_{j_2}), v_i>\}$. The guard and action of transition $<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{i_1}, s_{j_2}, s'_{j_2}), v_i>$, is the guard and action of $<(s_{i_2}, s'_{i_2}), v_i> \xrightarrow{e} <(s_{j_2}, s'_{j_2}), v_i>$. If $(s_{i_1} = s_{k_1}) \wedge (s_{j_2} = s_{k_2}) \wedge (s'_{j_2} = s'_{p_2}) \wedge (v_i = v_k)$, then *found* $=$ *true*, return the test sequence starting from the initial state $<(s_{0_1}, s_{0_2}, s'_{0_2}), v_0>$, and ending at the state $<(s_{i_1}, s_{j_2}, s'_{j_2}), v_i>$.
- For d occurring at $<(s_{i_2}, s_{i'_2}), v_i>$ do:
  $NU = NU \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>\}$, where $v_j = v_i + d$, $T = T \cup \{<(s_{i_1}, s_{i_2}, s'_{i_2}), v_i> \xrightarrow{d} <(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>\}$. If $(s_{i_1} = s_{k_1}) \wedge (s_{i_2} = s_{k_2}) \wedge (s'_{i_2} = s'_{p_2}) \wedge (v_j = v_k)$, then *found* $=$ *true*, return the test sequence starting from the initial state $<(s_{0_1}, s_{0_2}, s'_{0_2}), v_0>$, and ending at the state $<(s_{i_1}, s_{i_2}, s'_{i_2}), v_j>$.

End of While loop
End of Algorithm

The algorithm terminates as soon as a cover to the state $<(s_{k_1}, s_{k_2}, s'_{p_2}), v'_k>$ is found.

If the quantization of A is not equal to the quantization of B, we can always stretch them to be equal. If $d_1 = \frac{1}{k_1}$ and $d_2 = \frac{1}{k_2}$ are different quantizations for automata A and B, then we choose $d = \frac{1}{k}$, $k = lcm(k_1, k_2)$, as the quantization for grid-synchronous product of $\mathcal{G}_d(A)$ and $\mathcal{G}_d(B)$. This is justified because $lcm(k_1, k_2)$ is the maximal number of clocks in grid-synchronous product machine. Hence, the generated test case based on such a value of k will ensure accuracy of testing for all possible clock values in the product machine.

We use railroad crossing example to show the construction of the grid-synchronous product of two linear automata shown in Figure 45. One test case is taken from $\mathcal{G}_d(T \otimes C)$: *Near.Lower*.1/4.1/4.1/4.1/4.1/4.1/4.1/4.1/4.*In* with the granularity of $\mathcal{G}_d(T \otimes C)$ is $\frac{1}{4}$. The other test case is taken from $\mathcal{G}_d(C \otimes G)$: *Near.Lower.Down* with the granularity of $\mathcal{G}_d(C \otimes G)$ is $\frac{1}{3}$.

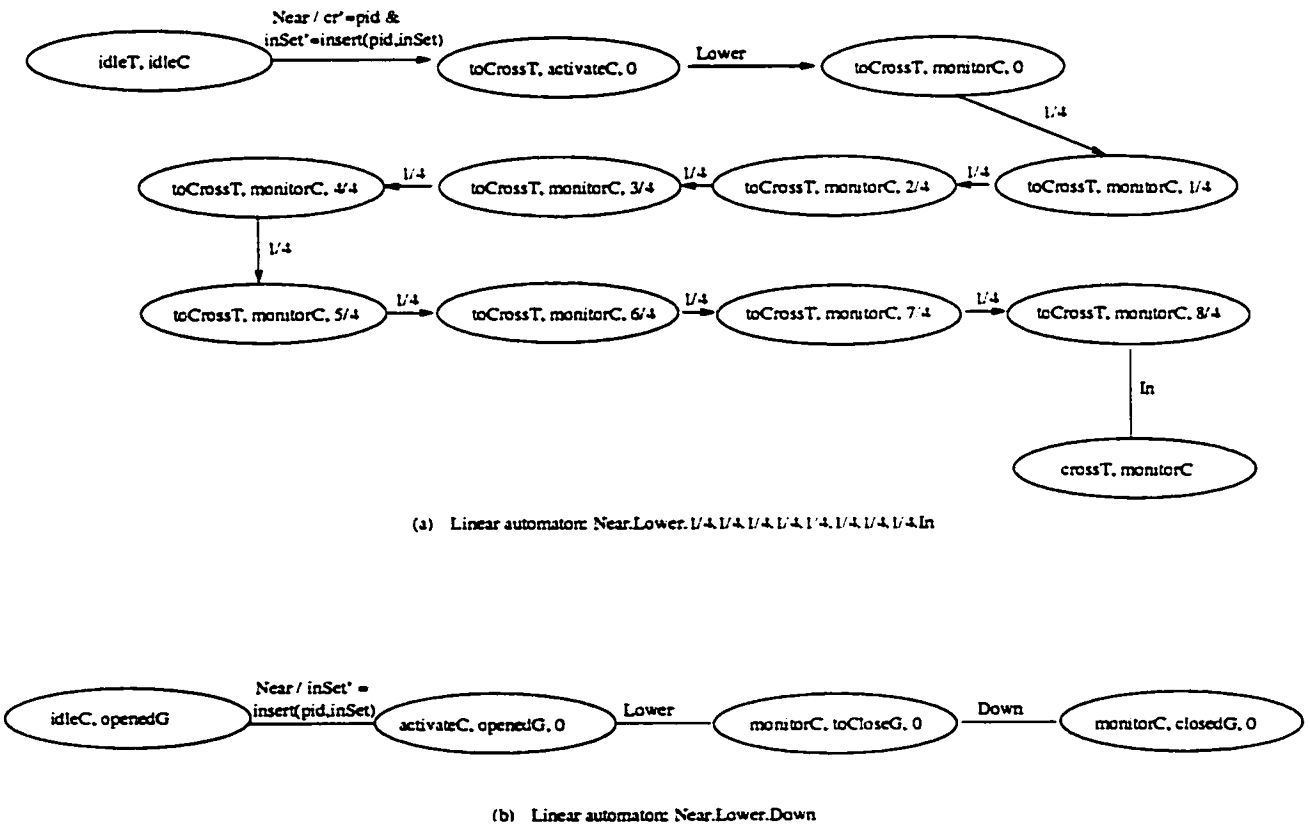


Figure 45: Two Linear Automata

We stretch the two quantizations to be equal to $\frac{1}{12}$. Based on the algorithm, the grid synchronous product of these two linear automata is:

$Near.Lower.Down.$1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.

1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.1/12.In

8.4.3 Grid-synchronous Product Theorems

We define the set $T_d(A)\, gs_{\otimes}\, T_d(B)$ as

$$\{\tau_A\, gs_{\otimes}\, \tau_B \mid \tau_A \in T_d(A), \tau_B \in T_d(B)\}\}$$

Since the test set can be viewed as a subautomaton of the grid automaton $\mathcal{G}_d(A)$, we can define the product machine $(\mathcal{G}_d(A)\, gs_{\otimes}\, \mathcal{G}_d(B))$ by extending the above definition to all paths in the grid automata. It is easy to see that every state of the grid automaton $\mathcal{G}_d(A \otimes B)$ is a state in $\mathcal{G}_d(A)\, gs_{\otimes}\, \mathcal{G}_d(B)$, and every transition in $\mathcal{G}_d(A \otimes B)$ is a transition in $\mathcal{G}_d(A)\, gs_{\otimes}\, \mathcal{G}_d(B)$. We summarize this result below as theorem.

Theorem GSP1

$$T_d(A)\, gs_{\otimes}\, T_d(B) \subseteq \mathcal{G}_d(A \otimes B) \subseteq \mathcal{G}_d(A)\, gs_{\otimes}\, \mathcal{G}_d(B)$$

Algorithm GSP computes the test cases for $\mathcal{G}_d(A \otimes B)$ without computing the grid automaton. It is necessary to show that this set consists of all state covers and transition covers that are both necessary and sufficient to test the grid automaton.

Theorem GSP2

The test cases computed by Algorithm GSP form a necessary and sufficient set of test cases for $\mathcal{G}_d(A \otimes B)$.

Proof:

From Theorem GSP1 the necessary condition follows. We prove sufficiency by assuming the contrary and arriving at a contradiction. Suppose there is a state $(\theta_A, \theta_B) \in \mathcal{G}_d(A \otimes B)$ for which no cover exists in the set $T_d(A)\, gs_{\otimes}\, T_d(B)$. Because Algorithm GSP examines all events in a states to construct a path, this implies that either the state θ_A of the grid automaton $\mathcal{G}_d(A)$ had no state cover in the set $T_d(A)$ or the state θ_B of the grid automaton $\mathcal{G}_d(B)$ had no state cover in the set $T_d(B)$. In either case we have a contradiction, since both $T_d(A)$ as well as $T_d(B)$ have sufficient number of test cases covering all the states in the respective grid automata $\mathcal{G}_d(A)$ and $\mathcal{G}_d(B)$.

The algorithm TC that generates state and transition covers is non-deterministic. It may not choose the same set of covers for a given input on two different runs. However, all the test sets output by the algorithm for different run are *equivalent*. Two test sets $T_d(A)$ and $T'_d(A)$ are equivalent if the following properties hold for the test sets:

- they have the same number of test cases,
- corresponding to each state $\theta \in \mathcal{G}_d(A)$ there exists exactly one cover in $T_d(A)$ and exactly one cover in $T'_d(A)$, and
- corresponding to each transition $t \in \mathcal{G}_d(A)$ there exists exactly one cover in $T_d(A)$ and exactly one cover in $T'_d(A)$.

By replacing a state (transition) cover in one test set by a different state (transition) cover of the same state (transition), we get an equivalent test set.

Lemma E

Let $T_d(A)$ be a test set generated by Algorithm TC. If $X \subseteq T_d(A)$, and X is equivalent to the set Y of test cases, then the set $T'_d(A) = (T_d(A) \setminus X) \cup Y$ is equivalent to $T_d(A)$.

Theorem GSP3

Let $T'_d(A \otimes B)$ denote the set of test cases computed by applying $gs_\otimes$ to the test sets $T_d(A)$ and $T_d(B)$. Let $T_d(A \otimes B)$ be the set of test cases generated by applying Algorithm TC on the synchronous product machine $\mathcal{G}_d(A \otimes B)$. Then the sets $T'_d(A \otimes B)$ and $T_d(A \otimes B)$ are equivalent.

Proof: From Theorem GSP2, every $\tau_{AB} \in T'_d(A \otimes B)$ is necessary as well as sufficient to test a state or a transition of $\mathcal{G}_d(A \otimes B)$. Algorithm GSP computes only one cover for each state in $\mathcal{G}_d(A \otimes B)$, and only one transition cover for each transition in $\mathcal{G}_d(A \otimes B)$. Hence, either $T'_d(A \otimes B) = T_d(A \otimes B)$ or $T'_d(A \otimes B)$ is equivalent to $T_d(A \otimes B)$.

8.4.4 Test Case Generation for a Component

For simplicity, we assume the quantization is the same for all grid automata.

Algorithm TCC

```
Input: A component $\Pi = \{A_1 \to A_o, A_2 \to A_o, \ldots, A_k \to A_o\}$, has $k+1$ objects,
    where $A_o$ is the core object in the vertex cover that has direct interaction with
    all the other objects in the component. $A_i \to A_o$ means that $A_i$ and $A_o$ interact,
    $i = 1, 2, \ldots, k$. Test sets $T_d(A_1 \otimes A_o)$, $T_d(A_2 \otimes A_o)$, ..., $T_d(A_k \otimes A_o)$.

Output: $T_d(\Pi)$

Step 1: Initialization
    $T_d(\Pi) \leftarrow T_d(A_1 \otimes A_o)$

Step 2: for $j = 2$ to $k$ do
    $T_d(\Pi) \leftarrow T_d(\Pi)\ gs_{\otimes}\ T_d(A_j \otimes A_o)$.
    where $(A_j \otimes A_o)$ must interact with current $\Pi$

End of Algorithm
```

8.5 Test Case Generation for System

Two components Π_1 and Π_2 of a partition Π are called independent if there is no communication between them. That is, no object in Π_1 communicate with any object in Π_2. However, Π_1 and Π_2 may have a common object which communicates at any time exclusively either with objects in Π_1 or objects in Π_2. If the components Π_i of a partition Π are independent, the test cases for the system Π, will be the union of the test cases for Π_is. In more general situations, the components of a partition may not be independent.

For instance, for the architecture shown in Figure 42, the components of Πs are not independent in the sense that a message communicated from 1 to 3 in component Π_1 may trigger a communication from 3 to 6 in component Π_4. A typical interaction scenario for the system, such as the one below,

$$1 \to 3 \to 4 \to 6 \to 5 \to 2 \to 4 \to 3 \to 1$$

can be broken up into the subsequences

$$1 \to 3\,(\Pi_1),\ 3 \to 4 \to 6\,(\Pi_4),\ 6 \to 5 \to 2\,(\Pi_5),\ 2 \to 4 \to 3\,(\Pi_4),\ 3 \to 1\,(\Pi_1)$$

where each subsequence is an interaction of objects in a component of the partition. To test whether a scenario is valid in a system, it is now sufficient to test the validity of each subsequence within a component, which is a smaller system. Moreover, each subsequence can be split pairwise, and the validity test can be conducted on every pair of objects.

We generate the test cases for the system $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$ depending upon whether or not the components are independent.

Algorithm TCS

Case 1 All the components are independent: $T_d(\Pi) = T_d(\Pi_1) \cup T_d(\Pi_2) \ldots, T_d(\Pi_m)$, where each $T_d(\Pi_i)$ is calculated by using the Algorithm TCC.

Case 2 The components are not independent as explained earlier. An interaction scenario for the system can be broken up into subsequences where each subsequences is an interaction of objects in the component. Consequently, $T_d(\Pi)$ is calculated by using the algorithm below.

```
Input: A system $\Pi = \{\Pi_1 \to \Pi_2 \to \ldots \to \Pi_k\}$, has $k$ component(s), where $\Pi_i$
    is a component, $\Pi_i \to \Pi_j$ means that $\Pi_i$ and $\Pi_j$ interact, $i, j = 1, 2, \ldots, k$,
    $i < j$. Test sets $T_d(\Pi_1)$, $T_d(\Pi_2)$, ..., $T_d(\Pi_{k-1})$, $T_d(\Pi_k)$.
Output: $T_d(\Pi)$
Step 1: Initialization
    $T_d(\Pi) \leftarrow T_d(\Pi_1)$
Step 2: for $j = 2$ to $k$ do
    $T_d(\Pi) \leftarrow T_d(\Pi)\ gs_{\otimes}\ T_d(\Pi_j)$
End of Algorithm
```

We apply Algorithm TCS to the railroad crossing problem architecture shown in Figure 43. The components Π_1 and Π_2 are independent. Hence, the test set for the whole system is the union of the test sets of the components. That is, $T_d(\Pi) = T_d(\Pi_1) \cup T_d(\Pi_2)$. We show below how $T_d(\Pi)$ is calculated:

The component Π_1 can be split into smaller components $\Pi_{11} = \{T_1, C_1, G_1\}$, $\Pi_{12} = \{T_2, C_1, G_1\}$, $\Pi_{13} = \{T_3, C_1, G_1\}$, because we would like to keep the objects in a component are from different classes. In TROM, there is no communication between the objects from same class. The interaction of the objects in Π_{11} is of the type $T_1 \rightarrow C_1 \rightarrow G_1$. So, consider pairwise interacting objects: T_1, C_1 and C_1, G_1. From the test templates generated from the grid automaton of the synchronous product machine Train $\otimes$ Controller, we generate the set of test cases $T_d(T1 \otimes C1)$ for the pair T_1, C_1. Similarly, from the test templates generated from the grid automaton of the synchronous product machine Controller $\otimes$ Gate, we generate the set of test cases $T_d(C1 \otimes G1)$ for the pair C_1, G_1. The test case for Π_{11} will be composed from $T_d(T1 \otimes C1)$ and $T_d(C1 \otimes G1)$: $T_d(\Pi_{11}) = T_d(T1 \otimes C1)\, gs_{\otimes}\, T_d(C1 \otimes G1)$.

Similarly, we compute the following test sets:

$$\begin{aligned}
T_d(\Pi_{12}) &= T_d(T2 \otimes C1)\, gs_{\otimes}\, T_d(C1 \otimes G1)\\
T_d(\Pi_{13}) &= T_d(T3 \otimes C1)\, gs_{\otimes}\, T_d(C1 \otimes G1)\\
T_d(\Pi_{21}) &= T_d(T3 \otimes C2)\, gs_{\otimes}\, T_d(C2 \otimes G2)\\
T_d(\Pi_{22}) &= T_d(T4 \otimes C2)\, gs_{\otimes}\, T_d(C2 \otimes G2)\\
T_d(\Pi_{23}) &= T_d(T5 \otimes C2)\, gs_{\otimes}\, T_d(C2 \otimes G2)
\end{aligned}$$

$$T_d(\Pi_1) = T_d(\Pi_{11})\, gs_{\otimes}\, T_d(\Pi_{12})\, gs_{\otimes}\, T_d(\Pi_{13})$$

$$T_d(\Pi_2) = T_d(\Pi_{21})\, gs_{\otimes}\, T_d(\Pi_{22})\, gs_{\otimes}\, T_d(\Pi_{23})$$

Finally,

$$T_d(\Pi) = T_d(\Pi_1) \cup T_d(\Pi_2)$$

Chapter 9

Conclusion

The goal of testing is to uncover errors. It can be viewed as a search problem. We are looking for those few input and state combinations that will reach, trigger, and propagate bugs out of several millions of behaviors. Hence, testing must be systematic, focused, and automated. It must be *systematic* if we are to ensure that every targeted combination is tried. It must be *focused* if we are to take advantage of available information about where bugs are likely to be found. It must be *automated* if we are to produce and run the greatest number of consistent and repeatable tests. In this thesis we have restricted to a study of test methods and test case generation that can be automated and systematically applied in a black-box fashion.

9.1 Test Case Generation and Systematic Testing

We have implemented the following algorithms:

- grid automaton construction,
- test case generation from a grid automaton, and
- test case generation for derived classes.

Whereas the construction of grid automaton and test case generation from it are fully automatic, test case generation from derived classes requires user interaction, in guiding the test generator the type of inheritance and the modifications made in the base class. In general, given two reactive objects, it is computationally hard to determine whether or not one object is inherited by the other object. Consequently, a complete mechanization of the algorithm for testing derived classes is not possible, unless its test cases are generated by constructing its grid automaton and then generating test cases from it. We have not implemented test case generation algorithms for testing a reactive system. Their implementation and test executions are non-trivial issues and are left for future research. Once their implementations are available, they work in **TROMLAB** environment in the following manner:

From a RTUML model, a conformance relation between the model and its implementation can be defined, and used by the black-box testing to link an implementation with its model. A translator implemented by Popistas [Pop99] is a tool to translate the graphically designed models such as statecharts, collaboration and sequence diagrams into TROM formal specifications. The input for the test case generator is the formal specification produced by the translator. The output of the test case generator are test suits for class testing, and system testing.

It is very important to organize the test information from the algorithms in a useful way. Test cases include test templates. As we remarked in Chapter 6, test templates must be organized after a careful examination of the goals of testing, and domain partitioning. It is very useful to define methods that manipulate and exercise test suites. Such methods are necessary for any realistic application of the methods provided in this thesis. To guide the testing process, in particular to provide a stopping rule for testing, we have developed a sufficiency criteria [Orm02]. This criteria is embedded in the Test Selection Algorithm that selects a sufficient number of tests from a given collection of test cases.

Let V denote the set of binary strings representing the original set of test cases TC, ϵ denote the initial target distance, and $\epsilon_{\min}$ denote some comprehensive minimum value of distance such that any approximation on distance smaller than $\epsilon_{\min}$ would not give more meaningful approximations. Let $\mathbf{C}$ denote some given threshold cost, and *Cost* denote the function representing the resources required to execute the (set of) test case(s). The Test Selection Algorithm selects the optimal set of test cases A from the set V. The algorithm stops when the cost limit is reached, the distance $\epsilon_{\min}$ is reached, or there are no more test cases left. We define the distance of a point $t \in V$ from the set A, $A \subseteq V$ by the formula $td(t, A) = inf\{td(t, y) \mid y \in A\}$.

Test Selection Algorithm

Precondition: $\{V = \mathbf{V} \neq \emptyset \ \wedge \ \epsilon_{\min} > 0 \ \wedge \ C = \mathbf{C} \ \wedge \ A = \emptyset\}$

```
    Step 1. $Initialization(A, V, \epsilon)$
    Step 2. $Create - Optimal - Test - Set(A, V, \epsilon)$
Postcondition: $\{A \neq \emptyset \wedge (Cost(A) \geq C \vee \epsilon < \epsilon_{\min} \vee V = \emptyset)\}$
```

Algorithm for *Initialization*(A, V, ϵ)

```
Precondition: $\{V = \mathbf{V} \wedge A = \emptyset\}$
    Step 1. $t = Longest - test - case(V)$
    Step 2. $Add(A, t);$
    Step 3. $Remove(V, t);$
    Step 4. $\epsilon = Length(t) - 1;$
    Step 5. IF $\epsilon <= 0$ THEN
                $\epsilon = \epsilon_{min};$
            ENDIF;
Postcondition: $\{A \neq \emptyset \wedge \epsilon > 0\}$
```

Algorithm for
Create – Optimal – Test – Set$(A, V, \epsilon, \epsilon_{\min})$

```
Precondition: $\{A \neq \emptyset \wedge \epsilon > 0\}$
  WHILE
    $!(Cost(A) \geq C \vee \epsilon < \epsilon_{\min} \vee V = \emptyset)$
      IF $(\exists$ test case $t$ : $td(t, A) >= \epsilon)$
      THEN $Add(A, t);$ $Remove(V, t);$
      ENDIF;
      $\epsilon = \epsilon - 1;$
  ENDWHILE;
Postcondition:
$\{A \neq \emptyset \wedge (Cost(A) \geq C \vee \epsilon < \epsilon_{\min} \vee V = \emptyset)$
```

The test selection algorithm has to be applied in order to select an optimal set of test cases. This optimization would reduce the cost of the testing process while maintaining the same level of efficiency.

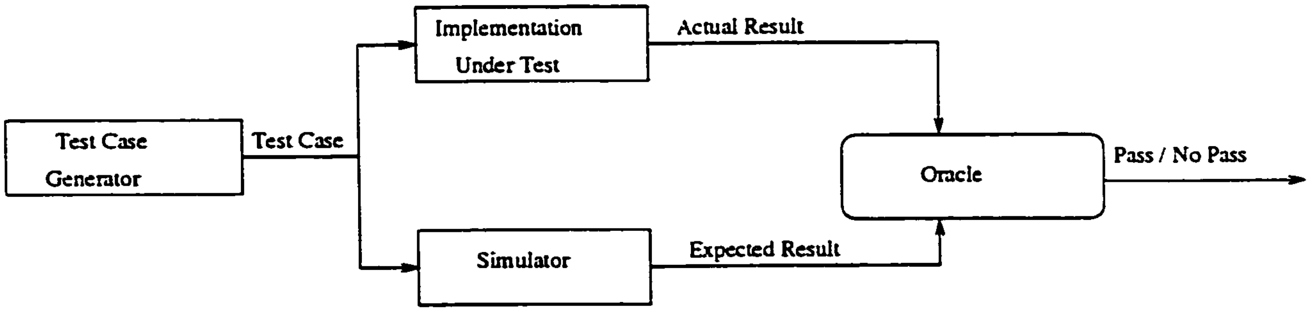


Figure 46: The Role of an Oracle

Figure 46 shows test harness template – the whole system used in the process of test execution and communicating with the implementation under test (IUT). The post-conditions in transition specifications can serve as the oracle. In TROMLAB we have a simulator [Hai99], which takes the system specification and simulates its behavior according to the operational semantics of the system. The expected behavior of the system for a given input (test case) is the system status reached by the simulator when started from the initial status of the system. The actual behavior for a given input (test case) is the output from IUT. The oracle's role is to examine whether the expected result matches the actual result. That is,

- the states match;
- for each active attribute in the states, the values in the states are equal;
- the time at which the states are reached are equal;
- the outstanding reactions are equal.

9.2 Summary of Contributions and Future Work

The major contributions of this thesis are:

- *State cover* and *transition cover* criteria are developed and applied in test case generation method. The generated test cases form a minimal set and can exhaustively test an implementation. Minimum implies that all test cases in the set are necessary. Exhaustivity implies that the test cases in the generated set are sufficient.
- Real-time models are compact representations of continuous varying in behaviors of reactive objects. In principle, the state machine of reactive objects has infinite number of states; however, due to the duration of the time constants, the continuous behaviors is modeled as part of the states and the transitions models discrete behaviors. The proposed testing method uses grid automaton to discretize the continuous time into discrete time model to avoid state exploration. The grid automaton has the same behaviors as original state machine.
- An algorithm of constructing a grid automaton from a TROM is given, and homomorphism theorem is introduced to justify the construction algorithm.
- Algorithms to generate test cases from the grid automaton are given. Since each class is associated with a grid automaton, test cases generated from the grid

automaton are used for testing the class. The test case generation algorithm produces a minimal set that can exhaustively test the implementation of a class for all specified properties.

- Test templates for inheritance testing are given. Test cases for a derived class are generated from the test cases for its base class.
- Algorithms are given to generate the synchronous product machine corresponding to a pair of interacting reactive objects and generate test case for the synchronous product machine. The test case generation algorithm produces a minimal set that can exhaustively test the implementation for all specified interacting properties.
- To reduce the complexity in generating test cases for a system, which is a collection of interacting reactive objects, an algorithm for partitioning the system architecture is developed. For each component in the partition, test cases are developed. To minimize the complexity in generating test cases for each component, it is proved that the test cases for unit testing and test cases for every pair of objects are sufficient.
- A method is given to compose the test cases of the whole systems from the test cases of components in the partition.

The research work on testing reactive systems developed in TROMLAB was recently studied. Issues that are currently being investigated include:

- The test case generation methods are valid for UML models that conform to the RTUML [Mut00] semantics. However, for general real-time UML models discussed in [Dou98], only Algorithm TC for unit testing is valid. Testing system configurations modeled in UML [Dou98] is an important research direction.
- Test Template Organization.
- Developing test drivers, test script driver for Java programs, test execution, and test result analysis.
- Management of the complexity of the test case generation for system testing.

- Integrating the testing module with other modules of TROMLAB framework: the Java code generation module [Zha00], and the measurement tool that collects and validates the testing quality measurement [Orm02], and the simulator [Hai99]. With the completion of this work, we can make the tools in TROMLAB framework available for real-time software engineering practice.

Bibliography

[AAM98] V.S. Alagar, R. Achuthan, D. Muthiayen. *TROMLAB: A Software Development Environment for Real-Time Reactive Systems.* (first version 1996, revised 2001), Submitted for Publication.

[AC95] V.S. Alagar and A. Celer. *Automating the Generation and Sequencing of Test Cases from Larch Specification.* Technical Report, Concordia University, Montreal, Canada, June 1995.

[Ach95] R. Achuthan. *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems.* Ph.D. thesis, Concordia University, Montreal, Canada, October 1995.

[AD94] R. Alur and D. Dill. *A Theory of Timed Automata.* Theoretical Comput. Sci., 126:183-235, 1994.

[AHOZ02] V.S. Alagar, May Haydar, O. Ormandjieva, M. Zheng. *A Rigorous Approach for Constructing Self-Evolving Real-Time Reactive Systems.* Submitted for Journal Publication, January 2002.

[AHOZ01] V.S. Alagar, May Haydar, O. Ormandjieva, M. Zheng. *A Rigorous Approach for Constructing Reusable, Self-Evolving Real-Time Reactive Systems.* In Proceedings of Concordia Prestigious Workshop Communication Software Engineering, CPWCSE2001, pp:139 - 154, Montreal, Canada, September, 2001.

[AOZ00a] V.S. Alagar, O. Ormandjieva, M. Zheng. *Specification-Based Testing for Real-Time Reactive Systems.* In Proceedings of 34th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS34, pp:25 - 36, IEEE Computer Society, Santa Barbara, California, USA July-August, 2000.

[AOZ00b] V.S. Alagar, O. Ormandjieva, M. Zheng. *Managing Complexity in Real-Time Reactive Systems.* In Proceedings of Sixth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS2000, pp:12 - 24, IEEE Computer Society, Tokyo, Japan, September, 2000.

[AZ01] V.S. Alagar, M. Zheng. *A Rigorous Method for Testing Real-Time Reactive Systems.* In Proceedings of Eighth Asia-Pacific Software Engineering Conference, APSEC2001, pp: 213 - 220, IEEE Computer Society, Macau, China, December 2001.

[BG93] I. Bashir and A.L. Goel. *Object-Oriented Metrics and Testing.* In Proceedings of Fifteenth Minnowbrook Workshop on Software Engineering, pages 1-9, Syracuse, New York, Jul. 1993. The Center for Advanced Technology in Computer Applications and Software Engineering(CASE), Syracuse University.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[CGP01] A. Cavalli, C. Gervy and S. Prokopenko. *New Approaches for Passive Testing using an Extended Finite State Machine Specification.* In Proceedings of Concordia Prestigious Workshop Communication Software Engineering, CPWCSE2001, pp:225 - 250, Montreal, Canada, September, 2001.

[CLRZ99] A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaidi. *Hit- or- Jump an algorithm for Embedded Testing with applications to In services.* In Proceeding of IFIP International conference FORTE/PSTV'99, Beijing, China, 5-8, October 1999.

[CTCC98] NY Chen, TH Tse, FT Chan, TY Chen. *In Black and White: an Integrated Approach to Class-level Testing of Object-Oriented Programs.* ACM Transactions on Software Engineering and Methodology, 1998; 7(3):250-295.

[CRS96] J Chang, DJ Richardson, S Sankar. *Automated Test Generation from ADL Specifications.* Proceedings of the Third International Symposium on Software Testing and Analysis. San Diego, ACM Press, 1996;62-70.

[DF93] J. Dick and A. Faivre. *Automating the Generation and Sequencing of Test Case Generation from Model-Based Specifications.* FME'93-Industrial Strength Formal Methods: First International Symposium of Formal Methods Europe, Odense, April 1993 (Lecture Notes in Computer Science, vol. 670), Springer-Verlag, 1993.

[Don97] MR Donat. *Automating formal specification-based testing.* TAPSOFT'97: Theory and Practice of Software Development, Lecture Notes in Computer Science Series vol.1214, Spring-Verlag, 1997:833-847.

[Dou98] B. P. Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems.* Addison-Wesley, Reading, MA, 1998.

[EDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, A.Elqortobi. *Timed Test Case generation Based on State Characterization Technique.* QA 76.54 R43, IEEE Real-Time System Symposium, New York, N.Y. 1998.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specifications.* Springer Verlag, 1993.

[Hai99] G. Haidar. *Reasoning System for Real-Time Reactive Systems.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.

[Hay01] M. Haydar. *Parameterized Events for Designing Real-Time Reactive Systems.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, February 2001.

[Heg89] W.A. Hegazy. *The Requirements of Testing a Class of Reusable Software Modules.* Ph.D thesis, Ohio State University, 1989.

[HM96] C.Heitmeyer, D.Mandrioli *Formal Methods for Real-Time Computing.* John Wiley & Sons, 1996.

[HP85] D. Harel, A. Pnueli. *On the Development of Reactive Systems.* In Logic and Models of Concurrent Systems, NATO, Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985.

[IEE83] ANSI IEEE. *An American National Standard IEEE Glossary of Software Engineering Terminology.* IEEE, 1983.

[Kir94] S. Kirani. *Specification and Verification of Object-Oriented Programs.* Ph.D thesis, Department of Computer Science, University of Minnesota, Minneapolis, MN, November 1994.

[MA99] D. Muthiayen, V.S. Alagar. *Mechanized Verification of Real-time Reactive Systems in an Object-Oriented Framework.* Technical Report, Concordia University, Montreal, Canada, 1999, (revised version 2001 submitted for publication).

[MD98] B.P. Mahony and J.S. Dong. *Blending Object-Z and Timed CSP: An Introduction to TCOZ.* Proceedings of the International Conference on Software Engineering (ICSE'98), Kyoto, Japan, April 1998, pp.95-104.

[MH92] B.P. Mahony and I.J. Hayes. *A Case Study in Timed Refinement: A Mine Pump.* IEEE Transactions on Software Engineering, vol.18, no.9, 1992, pp. 817-826.

[Mut96] D. Muthiayen *Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1996.

[Mut00] D. Muthiayen *Real-Time Reactive System Development - A Formal Approach based on UML and PVS.* Ph.D thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2000.

[Nag99] R. Nagarajan. *Vista - a Visual Interface for Software Reuse in TROMLAB Environment.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, April 1999.

[Orm02] O. Ormandjieva. *Quality Measurement for Real-Time Reactive Systems* Ph.D Thesis (in preparation), Department of Computer Science, Concordia University, Montreal, Canada, 2002.

[PA00] K. Periyasamy, V.S. Alagar. *A Rigorous Method for Test Templates Generation from Object-Oriented Specifications.* Software Testing, Verification and Reliability, vol.10, 2000.

[PA01] K. Periyasamy and V.S. Alagar. *RTOZ: An Object-oriented Language for the Specification of Real-Time Systems.* Submitted for Publication, March 2001.

[Pai75] Paige, M.R. *Program Graphs, an Algebra, and their Implications for Programming.* IEEE Trans. Softw. Eng. SE-1, 3(Sept. 1975).

[Pom99] F. Pompeo. *A Formal Verification Assistant for TROMLAB Environment.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, November 1999.

[Pop99] O. Popistas. *Rose-GRC Translator: Mapping UML Visual Models onto Formal Specifications.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, April 1999.

[Pro96] A. Protopsaltou. *Constructing Black-box Test Suits for Systems Specified in Larch/C++.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, June 1996.

[RP95] J. Rushby, M. Park. *Formal Methods and their Role in the Certification of Critical Systems.* SRI technical report, CSL-95-1, March 1995.

[SC96] P. Stocks, D. Carrington. *A Framework for Specification-Based Testing.* IEEE Transactions on Software Engineering, vol. 22. pp:777-793 No.11 November 1996.

[Sel99] B. Selic. *Turning Clockwise: Using UML in the Real-Time Domain.* Commun. ACM 42,10 October 1999, pp. 38-54.

[Sri99] V.Srinivasan. *Graphical User Interface for TROMLAB Environment.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.

[SVD97] J. Springintveld, F. Vaandrager, P. Dargenio. *Testing Timed Automata.* Technical Report CTIT97-17, University of Twente, Amsterdam, The Netherlands, 1997.

[Tao96] H. Tao. *Static Analyzer: A Design Tool for TROM.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, August 1996.

[WGS94] E Weyuker, T Goradia, A Singh. *Automatically generating test data from a boolean specification.* IEEE Transactions on Software Engineering 1994; 20(5):353-363

[Zha00] L. Zhang. *Implementing Real-Time Reactive Systems from Object-Oriented Design Specifications* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2000.