Scenario Based Requirement Analysis Modeling and Design Evaluation for Real-Time Reactive Systems

Shen Jian

A THESIS

IN

THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEMPTEMBER 2004
©SHEN JIAN, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94751-3
Our file Notre référence
ISBN: 0-612-94751-3

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canadä

ABSTRACT

Scenario Based Requirement Analysis Modeling and Design Evaluation for Real-Time Reactive Systems

Jian Shen

An effective requirements engineering process can greatly improve the quality of software development, direct an effective design solution, reduce early mistakes, and provide a foundation for evaluating design and implementation. In this thesis work, we explored how to integrate scenario-based requirements modeling approach into real time reactive system development. We developed an object-event-scenario requirement analysis modeling method, which describes system context, behavioural characteristics and system usages. We also developed a scenario-based design evaluation method, which applies a black-box approach to simulate design execution, test design solutions, and evaluate design performance. To promote and facilitate our scenario-based design evaluation, some supporting tools are designed and developed.

Acknowledgements

I would like to thank Dr. Olga Ormanjieva and Dr. V. S. Alagar who guided this work. I am grateful for their valuable advice and ideas in my research work. It is a great experience for me to have opportunity to work in TROMLAB team. I am grateful for invaluable friendship from Dr.Olga.

I would like to thank Dr. Olga for providing me financial support during my research study. I am grateful to Concordia computer science department for providing such good research environment that I myself benefit a lot from research resources.

Finally, I would like to express my deepest gratitude for the constant support and love that I received from my husband Lin.

To my family

Table of contents

List of Figures viii
Chapter 1 Introduction 1
 1.1 Motivation 1
 1.2 Research approach 4
  1.2.1 Research goal 4
  1.2.2 Research overview 4
 1.3 Thesis Outline 5
Chapter 2 Background 7
 2.1 Real-time reactive system 7
 2.2 TROMLAB 9
  2.2.1 Overview 9
  2.2.2 TROM Architecture 10
  2.2.3 Operational semantics 14
  2.2.4 TROMLAB Components 15
 2.3 Case study – Generalized Railroad Crossing 22
Chapter 3 Requirement analysis model 28
 3.1 Overview of our analysis modeling approach 28
  3.1.1 Requirement analysis characteristics 28
  3.1.2 Tasks of requirement analysis 29
  3.1.3 Features of our approach 29
 3.2 System static model 31
  3.2.1 Object model 31
  3.2.2 Identify objects 31
  3.2.3 System boundary 32
  3.2.4 Design boundary 33
  3.2.5 Summary of this section 34
  3.2.6 Production Cell case study 35
 3.3 System behaviour model 36
  3.3.1 Event-based modeling 36
  3.3.2 Object-events 37
  3.3.3. Object event attributes 37
  3.3.4 Object events classification 38
  3.3.5 Behavioural characteristics and constraints on objects 39
 3.4 System Usage model 41
  3.4.1 Usage scenario 41
  3.4.2 Usage model for real-time reactive system 42
Chapter 4 Scenario based design evaluation 44
 4.1 Introduction 44
 4.2 Design evaluation process 46
 4.3 Performance evaluation model 49
  4.3.1 Define performance metrics 50
  4.3.2 System Output Correctness 52
  4.3.3 Inability 57
  4.3.4 Average response time 59

4.3.5 Maximum throughput 60
4.4 Environment Data Generation 61
4.4.1 Canonical set 61
4.4.2 Guideline on systematic selection 62
4.4.3 Test data generation model 63
4.5 Case Study --- Evaluating the design of Generalized Railway Crossing 67
4.5.1. Correctness Validation 68
4.5.2. Evaluating response time 75
4.5.3. Inability Evaluation 76
Chapter 5 Tool support for automatic evaluation process 77
5.1 Design evaluation system 77
5.2 Event generation 78
5.2.1 Generation Algorithm 79
5.2.2 Generator and filter 82
5.2.3 Component design 84
5.3 Simulation 88
5.4 Evaluation 89
5.4.1 Evaluation Algorithm 89
5.4.2 Expression library 90
5.4.3 Scenario Legality evaluation 95
Chapter 6 Conclusion and future work 99
6.1 Thesis Review 99
6.2 Future work 99
Bibliography 101
Appendix 106
A. Generator snapshots 106
B. Expression library (part) 108

List of Figures

Figure 2-1 Three Tiers 11
Figure 2-2 Set trait 12
Figure 2-3 Template for TROM Class configuration specification 13
Figure 2-4 Template for Sub-system configuration specification 14
Figure 2-5 GRC-Translator Architecture 17
Figure 2-6 Interpreter Architecture 20
Figure 2-7 Architecture of simulation tool 22
Figure 2-8 Class diagram for railroad crossing 23
Figure 2-9 Statechart diagram for controller 24
Figure 2-10 Statechart diagram for gate 25
Figure 2-11 Collaboration Diagram for Railroad Crossing gate 25
Figure 2-12 Formal specification for Controller 26
Figure 2-13 Formal specification for Gate 26
Figure 2-14 Formal specification for Controller 27
Figure 3-1 Static model of real-time reactive system 34
Figure 4-1 Scenario-based design evaluation process 47
Figure 4-2 Simulated system and black-box design evaluation 49
Figure 4-3 scenario segments set of a simulated scenario 54
Figure 4-4 Interaction pattern I 64
Figure 4-5 Partition and selection 65
Figure 4-6 Interaction pattern II 66
Figure 4-7 Interaction pattern III 67
Figure 4-8 Modified design version 74
Figure 5-1 Design evaluation system 77
Figure 5-2 event sequence Generator and Filter 83
Figure 5-3 class diagram of Environmental stimuli simulator (ESS) 84
Figure 5-4 class diagram of Generator and filter 85
Figure 5-5 Activity diagram of Environmental stimuli simulator (ESS) 86
Figure 5-6 Filter and TES 87
Figure 5-7 Program model design 91
Figure 5-8 Scenario legality Evaluation algorithm 95
Figure 5-9 The Evaluation Result of Scenario A 97

"The whole of science is nothing more than a refinement of everyday thinking."

-- Albert Einstein

Chapter 1 Introduction

1.1 Motivation

1. Software engineering in real time reactive system development

More and more real-time reactive systems involve software design and development. The safety critical characteristics of real-time system require its software development to be incorporated into its engineering process, which is a promising way to improve quality, and to handle the increasing complexity of software.

2. Common drawbacks in a system development

Comparing to other phases of software engineering, requirement engineering (RE) is relatively weak in tools support and in practice [1] [30]. In industry, many delivered systems do not meet requirements due to ineffective requirement engineering. One of the reasons might be attributed to the development that is often conducted in tight schedule and cost budget. For saving time or cost, people may choose shortcut and rush to implementation. Before designers fully understand requirement, they quickly jump to a design solution by taking some implicit assumptions. And most this leads to design errors because of wrong assumptions and misunderstanding of the requirements. Another reason may be attributed to designer's lack of requirement engineering practice. This results in the fact that many designers contribute much more effort on specific design technique rather than on understanding the requirements. This also leads to limitation of solution space, in which some better solutions might be overlooked.

Correcting errors in implementation phases consume much more time and cost than doing this in requirement and design stages, many delayed and failed projects have proved this fact.

3. Roles of requirements engineering in a system development

As Zave described, requirement engineering is concerned with the real-world goals for, functions of, and constraints on software systems [2]. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families. Precise specifications provide the basis for analysing requirement, validating requirement, defining what designers have to build. In real-time reactive system, software has to function in the system in which it is embedded. Hence requirement engineering has to encompass a systems level view.

4. Desirable benefits of requirements engineering

With effective requirement engineering practice, we can pursue the following desirable benefits:

1. Bridge the gap between user's view and designer's view.

User and designer have different perspectives. User's viewpoint is at the usage of the system services. It is about "what" problem of system. In contrast, designer's view is about "how" system services are implemented. Requirements Engineering practice should be able to bridge these two different views and produce a clear contract that both user and designer can understand and agree on.

2. Make design more effective.

Requirement driven development will guide design solution to pinpoint the real-world problem, hence we can get effective design solution comparing with the one coming from design driven development.

3. Avoid error at early design phase

By clarifying requirements in all aspects in requirement engineering, we can avoid errors due to misunderstanding on requirement assumptions and system constraints at early design phase. It promises correctness of the design solution.

4. Avoid missing functionality via a systematic view of requirements

With a systematic view of domain problem, it is possible to give design solution with a completed functionality. It promises completeness of design solution.

5. Maximize solution space.

RE is independent of any design solution. RE is a comprehensive study of domain problem and does not hinge upon any design solution. Hence it can maximize design spaces.

6. Detect design error.

With a concise and unambiguous requirement, we can walk through design with scenarios to detect errors.

7. Automatic validation.

By incorporating formal approach into requirement engineering, we can realize automatic validation for design artefact.

8. Generate test cases.

Requirement is the fundamental source in the development. It can help generate test cases to test system artefact at any development stage.

9. Evaluate performance of design artifacts

Different design solutions may have different performance that can be evaluated through executable scenarios.

People realize from successful projects that effective RE efforts save development time and cost at the end, and brings many benefits to the project [12]. Moreover, RE is a key factor in determining the quality of systems that are delivered.

The term *engineering* in RE reminds people that RE is an important part of the engineering process. RE plays key role in the success of a project because it anchors development activities to a real-world problem. It analyzes appropriateness and evaluates cost-effectiveness of the interim and final solutions. RE represents a series of engineering decisions that lead from recognition and specification of a problem to be solved to guiding, validating, and testing a solution to that problem.

With these benefits in mind, in this thesis work we attempt to explore scenario-based requirement engineering in the development of real time reactive system.

1.2 Research approach

1.2.1 Research goals

This research is to explore an effective requirement engineering approach in real-time reactive system development.

The research work is conducted in the context of TROMLAB — a development framework for real-time reactive systems. We intend to enrich TROMLAB development environment with a comprehensive requirement analysis and design evaluation methodology.

1.2.2 Research overview

Bearing these goals in mind, the following works have been done in this thesis:

1. Requirement analysis modeling

We developed a scenario-based modeling method for analyzing real time reactive systems. This method can help us describe a system and its environment, and identify our requirement and environment assumption. We applied an object-event-scenario model in our modeling method, with which we can easily and clearly describe a system in different views.

2. Design performance evaluation

By extending the scenario-based modeling method, we developed a design evaluation method for evaluating different design solutions. In this method, we apply an executable model to simulate design execution and evaluate design performance. This method can help us develop an effective strategy to evaluate design artefacts and realize automatic design evaluation.

3. Tool support

To promote and facilitate our approach, we designed and developed software tools to support automatic design evaluation.

1.3 Thesis Outline

In chapter 2, we introduce background knowledge of this thesis work, which includes an introduction of real-time reactive systems, TROM formalism, and TROMLAB framework.

In chapter 3, we introduce our scenario-based requirement analysis model for real-time reactive systems. We propose our object-oriented event-based modeling to describe system dynamic and static view at requirements level.

In chapter 4, we introduce our scenario-based performance evaluation approach, which includes evaluation process introduction, performance metrics design, test data generation, some common metrics, and a case study.

In Chapter 5, we present our design and implementation for some supporting tools.

In Chapter 6, we conclude with some possible future direction to further enrich our modeling and evaluation approach.

Chapter 2 Background

2.1 Real-time reactive system

1. Definition of real-time reactive systems

Real-time reactive systems are computer systems that monitor and control their ***environment*** via continuous interaction. "Reactive" means system react permanently to changes of the environment. "Real-time" means reaction must be guaranteed within a certain interval of time. Any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period, is a real-time reactive system. A real-time reactive system may be a component of a larger system in which it is embedded. Such a component is called an embedded system. [7]

2. Standard software / hardware Architecture

Normally, a real-time reactive system is connected to its ***environment*** via standard real-time input-output devices such as sensors, actuators, etc.

3. Common general characteristics

In order to avoid possible confusion with other systems, true real-time reactive systems can be distinguished by the fact that failure to respond to an input in some pre-specified time is usually as bad as a wrong response. This is summarized by saying: "The right answer late is wrong". These pre-specified times are usually called deadlines, and they are an inherent characteristic in requirements specifications for real-time systems. When measuring the time at which an output is produced, we normally relate an output event to the time at which a particular input event was received by the system. For real-time

reactive systems, this input-to-output response is a clearly identified requirement of the system.

Real-time reactive system must often operate for days or even years without stopping in the most hostile environments. Their performance requirements are as important as functional requirements.

A real-time reactive system is fully responsible to synchronize with the dynamic environment. Beside this, reactive system works in non-termination mode, that the system is monitoring its environmental objects in all the time through sensors (or other input-output interfaces) and ready to response at any time through actuators (or other input-output interfaces).

4. Example applications of real-time reactive systems

Applications of real-time reactive systems, especially embedded systems are pervasive in our life. Some examples of real-time reactive systems are:

Railway crossing control, road traffic control, air traffic control;

Medical systems such as patient monitoring, radiation therapy;

Automobile cruise control system;

Manufacturing systems with robots;

Telephone, radio, and satellite communications systems;

Military uses such as firing weapons command and control;

5. Two Important properties

Real-time reactive system must function in ***time bounded*** mode because of real-time behaviour of environmental objects. It should not only provide correct service functions but should also satisfy timing constraints imposed on the functions. Timing constraints regulate real-time system behaviours.

The distinguishable characteristics of a real-time reactive system are its *stimulus-response* behaviour.

- Stimulus sychronization: always reacts to a stimulus from its environment.
- Response sychronization: the time elapsed between a stimulus and its response is acceptable for a relative dynamic environment.

Missing any one of these two features, real-time reactive system cannot function correctly and will definitely lead to system failure.

6. Safety critical characteristics demand different software engineering activity

In general, real-time reactive systems operate in safety-critical context that system failure may lead to major losses and cost. Because of this safety critical nature, it is usually not feasible to test and debug systems with their actual complete environments. Therefore, developing real-time reactive system mostly relies upon simulation techniques, comprehensive requirement analysis and design methodology. [7]

2.2 TROMLAB

This thesis work is conducted in the context of TROMLAB. In this section we introduce TROMLAB framework and its components that are related to this thesis work.

2.2.1 Overview

TROMLAB is a development framework based on TROM formalism for object-oriented design and development of real-time reactive system. TROM formalism provides conciseness and rigor description of system designs.

The design approach has all the merits attributed to object-oriented design: *modularity*, *compositionality*, and *hierarchy*. The design ensures that the stimulus and

response synchronization abilities of a reactive object are preserved in compositions and hierarchical refinements.

The basic building block of a reactive system model is **TROM**, a Timed Reactive Object Model. A TROM is a generic class from which several reactive objects can be instantiated and cooperated to create a reactive subsystem. [21] [13]

2.2.2 TROM Architecture

The three- tier structure of the object oriented methodology introduced by Achuthan in his PhD thesis [21], as shown in Figure2-1, is the basis of **TROMLAB** environment for developing reactive systems. The benefits derived from the object-oriented techniques include modularity and reuse, encapsulation, and hierarchical decomposition using inheritance.

The system is modelled using a three-tier design language. The three tiers independently specify the system configuration, reactive classes, and the abstract data types included in reactive class definitions. Lower-tier specifications are imported into upper tiers. Abstract data types are specified as LSL (Larch Shared Language) traits in the lowest tier, and can be used by objects modelled by TROM. The middle-tier formalism specifies TROM classes. TROM is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and time constraints. The upper-most tier specifies object collaboration where each object is an instance of a TROM.

The three tiers shown in Figure 2-1 are briefly described in the following sections.

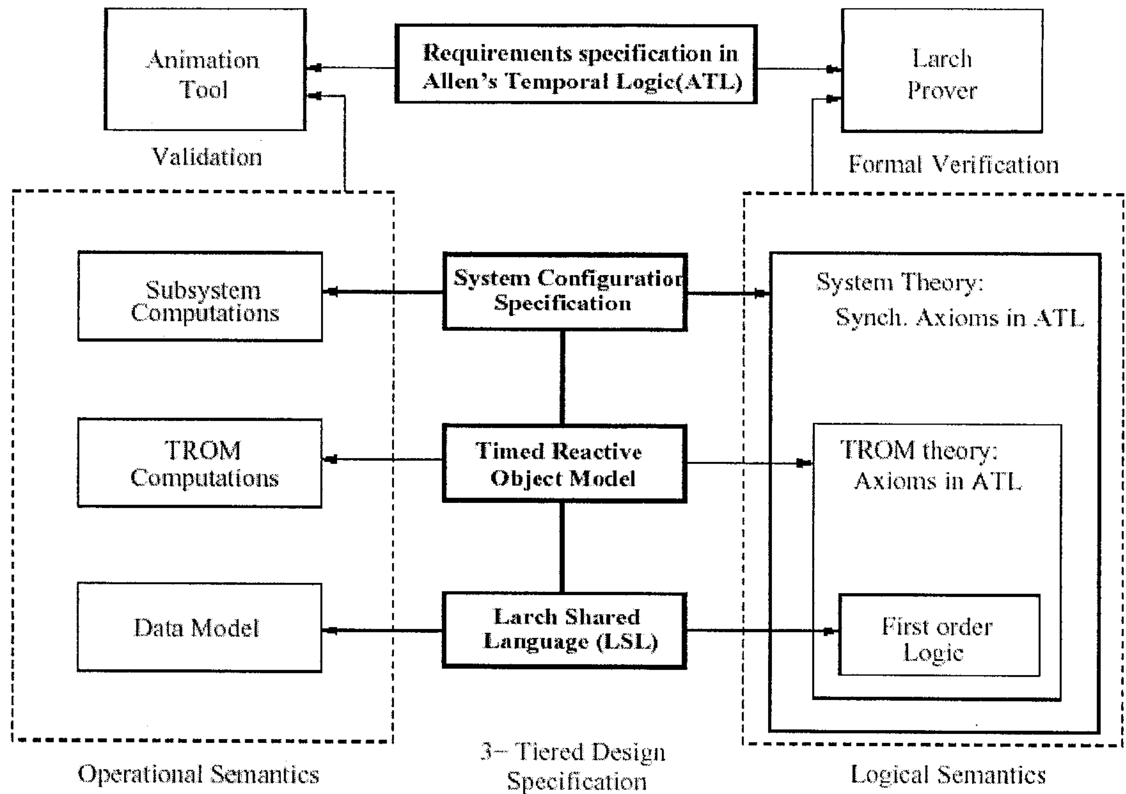

Figure 2-1 Three Tiers

2.2.2.1 Data Abstraction Tier

This level specifies the abstract data types included in the class definition of the middle tier. An abstract data type is defined as Larch Shared Language (LSL) trait. Larch provides a two-tier approach to specification:

- First tier, called Larch Interface Language (LIL), is used to describe the semantics of a program module.
- Second tier, called Larch Shared Language (LSL), is used to specify mathematical abstractions which can be referred to in any LIL specification.

Presently, the implementation of TROMLAB includes only LSL traits. Figure 2-2 shows the LSL trait for set data type.

```
Trait: Set(e, S)
      Includes: Integer, Boolean
      Introduces:
            creat  :       -> S;
            insert : e, S -> S;
            delete : e, S -> S;
            size   : S    -> Int;
            member : e, S -> Bool;
            isEmpty : S   -> Bool;
      Asserts:
end
```

Figure 2-2 Set trait

2.2.2.2 TROM Tier

A TROM models a *Generic Reactive Class* (**GRC**). A GRC is an augmented finite state machine with port types, attributes, hierarchical states, events triggering transitions and future events constrained by strict time bounds. A state is an abstraction denoting a system information during a certain interval of time.

An event denotes an instantaneous signal. The events are classified into three types: *Input*, *Output*, and *Internal*. Input (Output) events occur at the ports of a TROM, synchronising with the Output (Input) events of another TROM. The ports are abstraction of synchronous communication between TROMs. TROM objects can only interact through the port linking them as defined in SCS. Only compatible ports can be linked, such that event sent at one port is acceptable as an input event at the other port at the same time. The specification of a transition states the conditions under which an event may occur, and the consequences of such an occurrence. The time constraints enumerate the events triggered by a transition and the time bounds within which such events should occur. Thus, a GRC is a class parameterised with port types, and encapsulates the

behaviour of all TROM objects that can be instantiated from it. A formal definition of TROM is given in Achuthan's thesis [21].

The occurrence of an event ***e*** at a port ***p*** at time ***t*** triggers an activity which may take a finite amount of time to complete. These events may lead the TROM(s) affected by the event to undergo a state change and may further lead to the occurrence of new events as specified by the timing constraints. Figure 2-3 shows the syntax for specifying a TROM.

```
Class < identifier > [< porttypes >]
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 2-3 Template for TROM Class configuration specification

2.2.2.3 Subsystem Specification Tier

This level is the top most tier which constitutes subsystem configuration specifications (SCS). This specification uses objects instantiated from classes specified in the second tier. An object is instantiated from a class by creating a finite number of ports for each port type in the class specification, and by initializing the attributes included in the class. Each instantiated object will carry its own set of attributes. A port link is an abstraction of a communication medium between two objects. A port link is established between a port of one object and a compatible port in another object. Objects communicate by exchanging messages (external events) through the port links. These objects may have

different number of ports for each port type, and consequently have the ability to communicate and interact differently with their environment. We can also include other subsystem configurations in defining a subsystem. The syntax for subsystem specification is shown in Figure 2-4. The **Include** section lists imported subsystems. A reactive object is created in the Instantiate section, with parametric substitutions to cardinality of ports for each port type. The Configure section defines a configuration obtained by composing objects specified in the **Instantiate** section and in the subsystem specification imported through the **Include** section.

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

Figure 2-4 Template for Sub-system configuration specification

2.2.3 Operational semantics

The structure and behaviour of TROM can be described either textually or visually. The templates for textual descriptions of TROMs and subsystems are shown in Figures 2-3 and 2-4. The visual representation of a reactive system includes the class diagrams, state machine diagrams, and the collaboration diagrams.

Reactive objects in a system communicate through messages. A message from an object to another object in the system is called a *signal* and is represented by a tuple $<e, p, t>$, denoting that the event e occurs at time t at a port p_i. The status of a TROM at any

time t, is the tuple $< s; a; R >$, where the current state s is a simple state of the TROM, a is the assignment vector, and R is the vector of outstanding reactions. A computational step of a TROM occurs when the object with status $< s, a, R >$, receives a signal $<e, p, t>$ and there exists a transition specification that can change the status of the TROM. A computation c of a TROM object A is a sequence, possibly infinite, of alternating statuses and signals, such that successive statuses in the sequence result due to the signal in between them. A reactive system may not terminate; consequently, a computation is in general an infinite sequence. The set of all computations of a TROM object A is denoted by *Comp(A)*. The computation of a system is an infinite sequence of system statuses and signals that effect global status changes of the system.

2.2.4 TROMLAB Components

TROMLAB is integrated with Rational Rose that enables the construction of visual models of a reactive system. The formal language RTUML [23] is discussed in D. Muthiayen's PhD thesis, a real-time extension of UML, which, together with TROM semantics, provides the semantic grounding for modeling real-time reactive systems in UML visual models.

An application developer can interact with TROMLAB functionalities through the interface provided by the Rose-GRC tool and a graphical user interface GUI. The visual models are mechanically translated by the Rose-GRC translator into the formal notation described in the previous section. GUI in TROMLAB is used by developers to interact with the rest of the TROMLAB components that include interpreter and simulator.

In the following sections we briefly review the functionality of the Rose-GRC Translator, Interpreter, and Simulator.

2.2.4.1 The Rose-GRC Translator

The translator is implemented by Popistas [14], and uses Rose script, a scripting language supported by Rational Rose. The translator is a tool to automatically translate the graphically designed models: class diagrams, state chart diagrams, collaboration diagrams into TROM formal specifications. The diagrams are modeled using Rational Rose that supports UML based system modeling. The translator takes the Rose model as input and produces text files: TROM class formal specifications, subsystem configuration.

The Rose-GRC translator consists of the following parts:

- *Interface to Rose:* It is the user interface that provides access to create visual representations and invoke the translator.
- *Translator:* Translator takes the specified Rose diagrams and performs the following tasks:
    a. Checks the correctness of the Rose diagrams by performing syntactic and semantic check to ensure that the models conform to TROM formalism.
    b. Handles any error occurring during the execution by producing clear and specific error messages.
    c. Translates the Rose diagrams into an internal structure, using record data types.
    d. Produces the textual specifications according to the syntax presented in the previous section.

  Figure 2-5 Shows the architecture of the Rose-GRC translator, as proposed by [14].

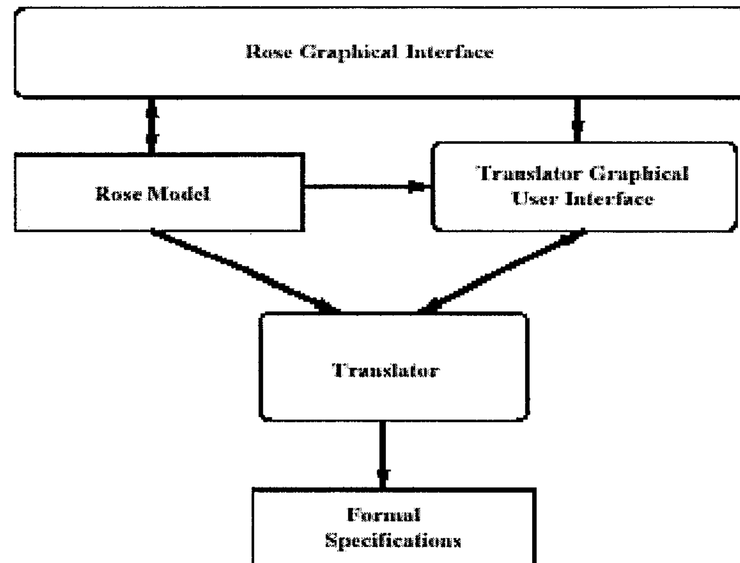

Figure 2-5 GRC-Translator Architecture

2.2.4.2 The Interpreter

The interpreter was the first tool to be implemented in TROMLAB by Tao [15], checks the textual specification for syntactic correctness and builds an internal representation of the formal specification of a reactive system. The implementation, in C++, required the textual descriptions of all the three levels to be provided as a single source file. The advantage of the three-tiered design was not realized in this implementation. Haidar [16] reengineered the interpreter implementation in Java. This version included incremental and independent compilation of specifications, and enhanced error reporting. The interpreter checks the syntactic correctness of specifications and builds an internal representation of the well-formed formal specification of a reactive system. Figure 2-6 shows the architecture of the interpreter.

In order to build the internal representation it performs the following tasks:

- *Syntactic analyzer:* It makes sure that the files are syntactically correct; that is, consistent with TROM grammar.

- *Semantic analysis:* It does simple semantic analysis such as:
  a. States of a TROM have different names.
  b. An LSL trait is used after being declared.
  c. Every transition has an outgoing and incoming state
  d. Transition specifications are well-formed logical formulas
- *Internal structure:* Based on a syntactically and semantically correct text file, it generates an Abstract Syntax Tree, an internal representation for the models, that would be used by the simulator in TROMLAB.

The components of interpreter are as follows:

- *Scanner*

A single text file containing LSL traits, TROM class specifications, subsystem specification, and an initial event list is taken as input to the scanner. The scanner performs lexical analysis and identifies the tokens to be used by the parser.

- *Parsers*

This certifies the syntactic correctness of the tokens received from the scanner. The parsers are implemented in JavaCC and JJTree. Java Compiler Compiler is a parser generator for use with Java applications that produces Java code. JJTree is a preprocessor for JavaCC that inerts in JavaCC source actions for parse tree building. There exists separate parsers for LSL trait, TROM class specification, SCS, and initial simulation event list.

- *Syntax analyzer*

Using predefined grammars for TROM and subsystem, this module evaluates the syntactic correctness of token. Any error at this stage will be communicated to the user and will terminate the execution of the interpreter.

- *Abstract syntax tree generator*

An abstract syntax tree is generated for each TROM and subsystem input to the interpreter.

- *Semantic analyzer*

Semantic analysis is done in two phases: in phase 1, semantic analysis internal to a class specification is done; and in phase 2, semantic analysis of the subsystem configuration is done.

- *Error message handler*

This is part of semantic analyser functionality. Every semantic error detected will be saved in a file until the end of semantic analysis.

2.2.4.3 Simulator

The Simulator tool was designed and implemented by Muthiayen in 1996, and redesigned and by Haidar [16]. It works with the new interpreter and has reasoning capabilities. Figure 2-7 shows the simulator architecture. The Simulator interfaces with the abstract syntax tree built by the Interpreter to extract the information for simulation. It builds a simulation event list to keep track of all outstanding events in the system. The Simulator can work in one of two modes:

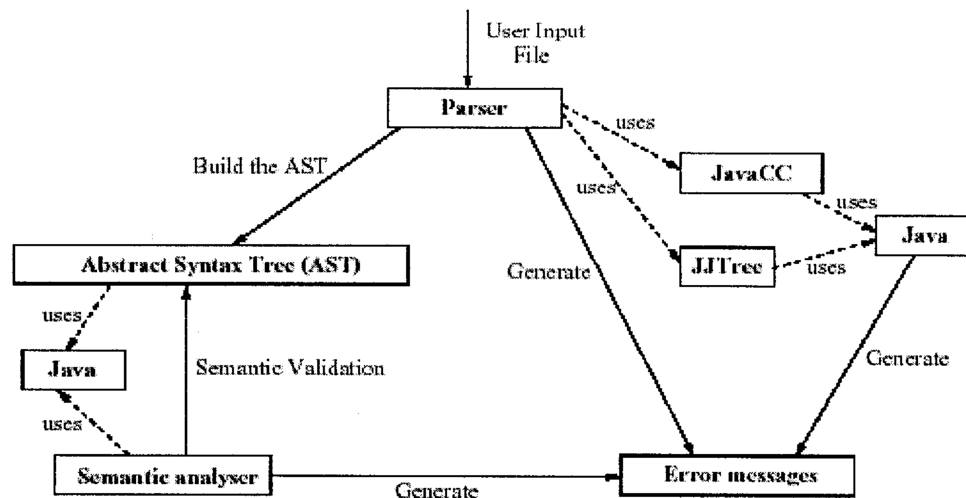

Figure 2-6 Interpreter Architecture

- Debugger mode: In this mode the developer can, at the end of every handled event, invoke the debugger and use it to query the system. The system can be rolled back and new events can be injected.
- Normal mode: In this mode the simulation will go on uninterrupted until the system goes into a stable state. The result of the simulation is one scenario of what could happen, given the initial set of events.

The Simulation tool consists of the following components:

- *Simulator* consists of an Event handler, a Reaction window manager and an Event scheduler.

a. Event handler is responsible for handling the events which are due to occur and detects the transition which the event will trigger.

b. The Reaction window manager is responsible for activating the computational step to handle the transition causing events to be fired, disabled or enabled.

c. The Event scheduler causes an enabled event to occur at a random time within the corresponding reaction window. It schedules output events through the least recently used port using a round robin algorithm.

- *Consistency checker* ensures continuous flow of interactions by detecting deadlock configurations.

- *Validation tool* consists of a Debugger, a Trace analyser, and a Query handler.

  -The Debugger supports system experimentation by allowing the user to examine the evolution of the status of the system throughout the simulation process. It also supports interactive injection of simulation event, and simulation rollback to a specific point in time.

  -The Trace analyzer includes facilities for the analysis of the simulation scenario. It gives feedback on the evolution of the status of the objects in the system, and the outcome of the simulation event.

  -Query handler allows examining the data in the AST for the TROM class to which the object belongs, and supporting analysis of the static components during simulation.

- *Object model support* supports the specification of the TROM classes and the evaluation of the logical assertions included in the transition specifications.

- *Subsystem model support* creates subsystems by instantiating included subsystems from object and port links.

- *Time manager* maintains the simulation clock updating it regularly. It allows setting the pace of the clock to suit the needs of analysis of simulation scenarios.

It also allows freezing the clock while analyzing the consequences of a computation.

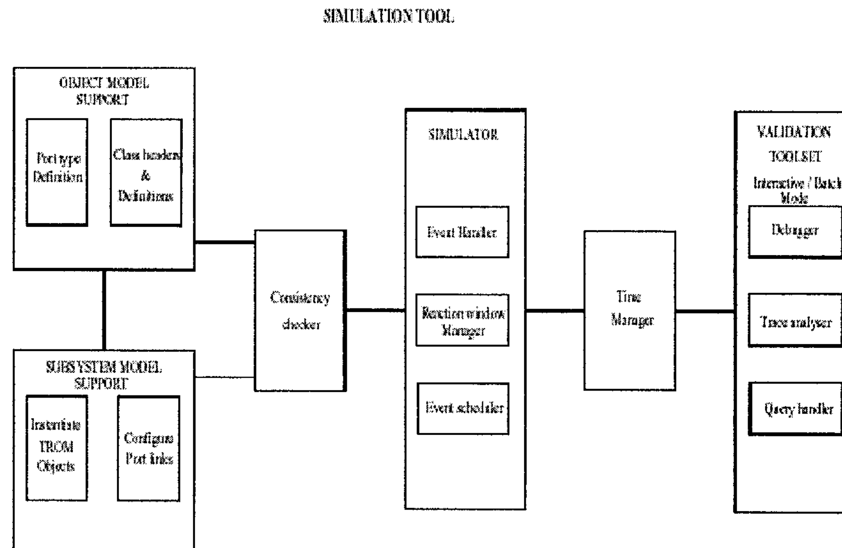

Figure 2-7 Architecture of simulation tool

2.3 Case study – Generalized Railroad Crossing

We illustrate TROMLAB methodology with this case study, in which we show a design version that model system behaviour using generic reactive classes introduced in this Chapter. The completed design specification of this case study can also be found in [18], [23]. This design version will be evaluated later on using evaluation methodology introduced in Chapter 4.

Problem Description:

Generalized Railroad Crossing problem is a benchmark case study in real-time reactive research community. We use a version of this problem, which is to develop an automatic gate-controlling system. This system can detect trains approaching and leaving, then make decisions to open or close gate. The system context is that, several trains, traveling

in different directions, traverse a crossing independently and simultaneously using multiple non-overlapping tracks. Gate-controlling system detects the first train entering the crossing and makes sure the gate is closed before any train is inside the crossing. Gate-controlling system detects the last train leaving the crossing and make sure the gate is opened. The initial position of a gate is open and the gate remains in initial position when no train comes. When there is a train in the crossing, the gate should remain closed. To maintain generality, we assume an arbitrary number of trains in the system, and consider that all trains interact with the system.

Timing assumption about a gate is as follows:

The gate will take 1 time unit to change the position from open to close.

The gate will take 1 to 2 time units to change the position from close to open.

The controlling system takes 1 time unit to issue order to gate for opening or closing action.

Timing assumption about a train is as follows:

A train is inside the crossing 3 to 5 time units after the controlling system detect its approaching, then the train takes another 1 to 2 time units to leave the crossing and the controlling system detects its leaving.

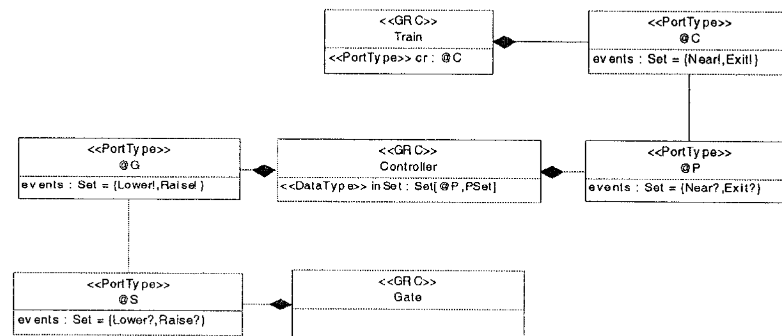

Figure 2-8 Class diagram for railroad crossing

Design solution with TROM formalism:

With TROM formalism, we model the behaviour of system entities using generic reactive classes. For GRC classes, we visually model their behaviour using UML statechart diagrams, and derive the corresponding formal specifications. Figure 2-8 shows the GRC classes Controller, Train, and Gate, with their corresponding PortType classes. The binary associations between the PortType classes indicate communication channels between instances of the GRC classes. Figures 2-9,2-10 show the statechart diagrams depicting the behaviour of instances of the GRC classes *Controller, Gate*. The UML collaboration diagram in Figure 2-12 shows the configuration of a railroad crossing system with four trains, one controller, and one gate. In this configuration, all the trains interact with the controller. Figures 2-13,2-14,2-15 show the formal specifications generated from the UML model of the GRCs. The formal specifications in figure 2-16 describe the configuration of the above mentioned instance of the railroad crossing system.

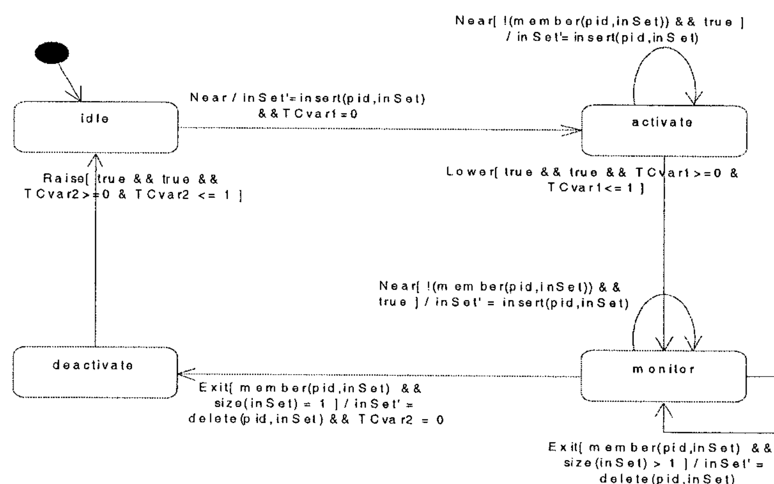

Figure 2-9 Statechart diagram for controller

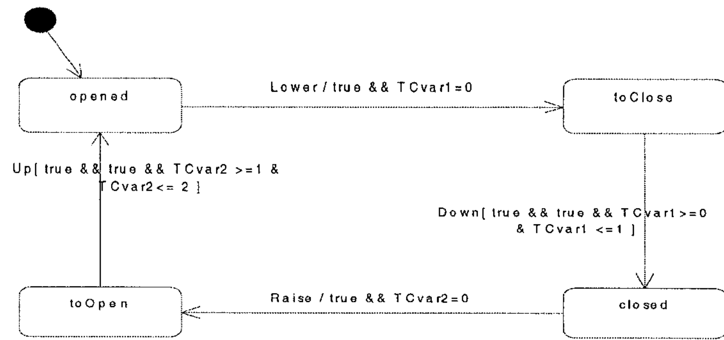

Figure 2-10 Statechart diagram for gate

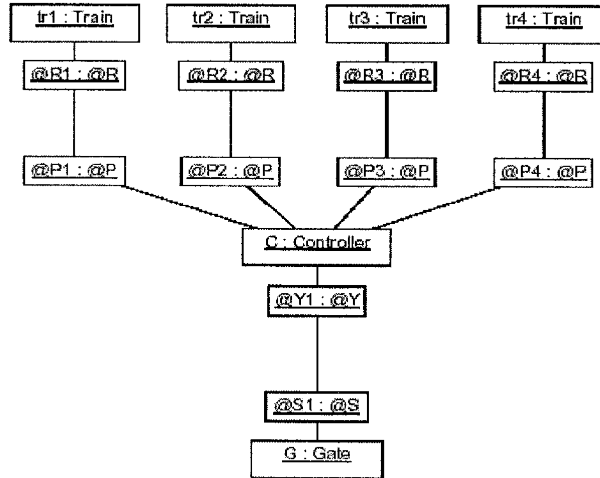

Figure 2-11 Collaboration Diagram for Railroad Crossing gate

```
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 2-12 Formal specification for Controller

```
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 2-13 Formal specification for Gate

```
SCS TrainGateController
    Includes:
    Instantiate:
        G::Gate[@S:1];
        tr1::Train[@R:1];
        tr2::Train[@R:1];
        tr3::Train[@R:1];
        tr4::Train[@R:1];
        C::Controller[@P:4, @Y:1];
    Configure:
        C.@Y1:@Y <-> G.@S1:@S;
        C.@P1:@P <-> tr1.@R1:@R;
        C.@P2:@P <-> tr2.@R2:@R;
        C.@P3:@P <-> tr3.@R3:@R;
        C.@P4:@P <-> tr4.@R4:@R;
end
```

Figure 2-14 Formal specification for Train-Gate-Controller subsystem

Chapter 3 Requirement analysis modeling

Requirement analysis is the start of the system development and will steer the whole system development from beginning to the end. The output of requirement analysis is the foundation for the subsequent development activities, guiding system design, and implementation, and providing criteria to validate design artefacts and final products. The quality of the analysis output is the key to achieve a successful development.

Therefore, it is important to have a sound analysis modeling approach that can help us produce consistent and completed requirement analysis result. In this chapter, we propose a scenario-based approach to model and analyze real time reactive system requirements.

3.1 Overview of our analysis modeling approach

3.1.1 Requirement analysis characteristics

As an important initiative process in software development, requirement analysis mainly has the following aspects:

1) Problem oriented

Requirement analysis emphasizes an investigation of the domain problem and requirements, rather than the possible solution [9]. The models that are developed during analysis are fully problem-oriented. No consideration is paid to the real implementation detail by which the system is to be realized.

By requirement analysis, we produce clarified requirement documents that can eliminate misunderstanding in the design phase. In addition, focusing on problems instead of a specific solution can also widen our view on possible solution space.

2) User-centered analysis

Requirement analysis is an investigation from user perspective rather than designer perspective. Thus, it focuses on how the system will be used and in what context. From user perspective, usually we see more requirement details than from designer's perspective. This can help us reduce chances of missing requirements.

3.1.2 Tasks of requirement analysis

A requirement analysis process should deliver outputs that cover the following:

1) A static view of the system to be designed as well as its environment;

2) A behavioural view of the system and its environment;

3) A system usage view.

In this chapter we propose an object-event-scenario modeling method aiming at describing these three views of a system under development.

3.1.3 Features of our approach

Based on the above understanding of requirement analysis, we propose our requirement analysis modeling approach by which we aim at the following objectives:

1) With this approach, we may produce high quality requirement analysis output that can guide the design process to be effective;

2) The output of the requirement analysis can be used to evaluate design artefacts and implementation.

In general, the modeling approach we proposed has the following characteristics:

1) Object oriented

Object-oriented analysis has many advantages in modeling system context. We use object model to describe the static structure of a system and its environment. The object model we applied can be easily mapped to the counterpart in design phase.

2) Event-based

Instead of using state-based model such as finite state machine that is commonly used as design modeling tool, we adopt event model in describing behavioural view of the system. This is very intuitive for all people involved in the development. In addition, event-based model is the building block for constructing usage scenarios.

3) Scenario-based

Scenarios are used to describe the usage of the system. Scenario-based approach is a very intuitive way to describe system requirement from user's perspective. Scenarios describe how system components, the environment objects interact in order to provide system level functionality. Each scenario is a story which, when combined with all other scenarios, should conform to provide a complete system description.

At analysis stage, scenarios models are good means to involve stakeholders contributing their experiences and knowledge. Normally, each stakeholder has his own *viewpoint* on how the system should work and how the interactions should flow among objects. During requirement analysis, different viewpoints can be discovered and conflicts across the different viewpoint have to be solved.

4) Formalism

As real-time reactive systems often operate in safety-critical situation, they require relatively rigorous method in the system development. In our approach, we apply a formal method on requirement specification as well as performance property specification. This formalism is used not only for the purpose of specification but also for automatic evaluation of the performance of design artefacts. We are going to explain evaluation approach in details in next chapter.

In the following sections, we are going to elaborate in details on our requirement analysis model.

3.2 System static model

3.2.1 Object model

Requrements analysis is based on information of different kinds. We may assume that they are informally described documents or information from all sources. The first step of requirement analysis is to elicit a static view of the system context, including the system to be developed and its enviroment.

3.2.2 Identify objects

Object-oriented requirement analysis emphasizes on identifying and describing the *objects* in the problem domain. In real-time reactive problem domain, a system to be developed and its environment form a closed system context. In this section, our task is to identify *objects* in this closed system context.

When we study a system, *objects* are the things that can be named by nouns and concern us in terms of the targeting problem to be solved. An *object* can be a physical component, a software component, an entity, or a data value.

Identified objects play certain roles in the system interactions. In early requirements analysis phase, identified objects that concern us are usually observable from users' perspective. The term "*observable*" here means the object has observable behaviour from the user point of view, or the object participates in some observable interaction behaviour. For example, in railroad crossing case study introduced in section 2.3, trains and the gate have observable behaviours while the controller is not visible to users.

3.2.3 System boundary

Identified objects can be classified into two categories: *system objects* and *environmental objects*. By this classification, we aim at clarifying the *system boundary*. The system boundary defines what objects compose the system as a whole and what objects do not belong to the system under development, but interact with it. Clarifying the system boundary is necessary for identifying the system and environmental events. In the analysis of system functional requirements, system boundary clarifies the scopes of system functions.

The reactive system to be designed is usually composed of many objects and is considered as a *composite system object* in a black-box view. Any objects that belong to the composite system object are system objects. The objects outside of system boundary are the environmental objects.

Among all system objects, those that do not have observable behaviour from the users' point of view are located inside the system. We name them as *internal system objects*. In requirement analysis phase, internal system objects are not of concern, unless

the requirements to be analyzed present some internal constraint to the development such as adoption of available system components.

Those system objects that have observable behaviours usually have interaction with environmental objects. We name them as *system interface objects*.

Besides system objects and environmental objects, there are some objects that concern us, however, you can not say they are system objects or environment objects. Their appearances are always combined with the occurrence of some object behaviours. Therefore, we name them as *parameter objects*.

Object quantity needs to be specified in the requirement, such as number of systems to be developed in the usage context, number of system interface objects of each type for each system, and number of environmental objects that can concurrently interact with the system.

3.2.4 Design boundary

It is very often the case that in real-time reactive system, some system objects are pre-selected commercial off-the-shelf (COTS) software or hardware products that we only know their input-output interfaces without knowledge of COTS' internal structure. Hence, *design space* for COTS will not include COTS itself but only COTS' interface, which is the communication mechanism of COTS interfaces that is about how to integrate them with other system objects. COTS' interface is inside *Design boundary*. Design boundary segments the system into designable and non-designable areas within system boundary.

3.2.5 Summary of this section

In this section, we have identified different kinds of objects after drawing system boundary and we illustrated in Figure 3-1:

- Environmental objects are located outside of system boundary, which trigger system behaviours; System objects are located within system boundary;
- System Interface objects are objects within system boundary having interaction with environmental objects. System internal objects are objects within system boundary that have no direct interaction with environmental objects.

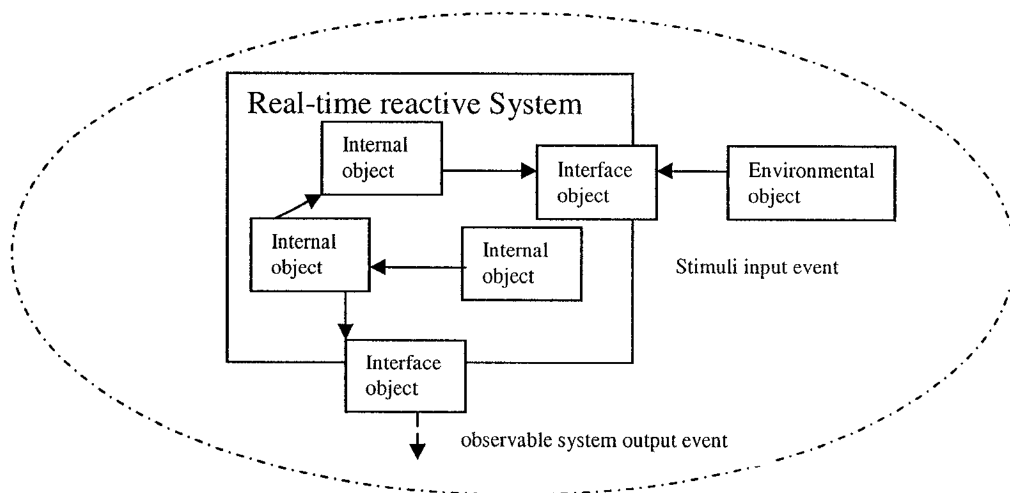

Figure 3-1 Static model of real-time reactive system

By identifying these objects and clarifying relationship among them, we have a static architecture topology of a system, which provide a platform for system dynamic behaviours.

3.2.6 Production Cell case study

In the railroad crossing example, we can identify trains as environmental objects, and a gate-controlling system as system object. Gate is a system interface object in gate-controlling system. The gate can not send events to other objects, but it has observable behaviour. (See observable event illustrated in Figure 3-1). Stimuli from trains are sent to controlling system to trigger gate's behaviour.

We introduce another example here. It is a version of an industrial production cell [13]. This example will be used throughout the thesis.

> The assembly floor consists of several assembly units. Each assembly unit is a production cell that consists of users, a robot, a conveyor belt, a tray, and a vision system. Users place two kinds of parts, cup and dish on the belt. A robot has two arms that each arm can pick up a part from the belt, hold a part, and place a part on a designated assembly pad. The vision system senses a part and recognizes its type. The belt stops whenever a part is sensed, so that the robot can pick up the part from the belt. The belt moves again after the part on it is picked up by the robot. An assembly is performed when the robot places a cup held in one of its arms and the dish held in its other arm onto a tray. In the generalized version of the problem, several assembly units cooperate to complete the assembly of a complex component.

In this example, we can identify environmental objects as users who introduce parts into production cell system constantly. Production cell system is the composite system object under designed that its main function is to assemble the parts provided by the users. Belt is the interface object located within the system boundary that receives

parts from user. The tray is the interface object that has observable behaviour which is to trash assembled parts. Vision system, belt and robotics are COTS (commercial off the shelf products) components. Parts and assembled products are parameter objects. Parts are the parameter for object user's behaviour. Assembled products are the parameter for object tray's observable behaviour.

3.3 System behaviour model

3.3.1 Event-based modeling

After having a static view of a system context, the next immediate step is to have a behavioural view of the system and its environment, namely, to clarify the behavioural characteristics of each concerned object.

There are generally two types of modeling approach in describing a course of system actions: state based and event based. These two modeling approaches are equivalent since an event may indicate the change of an object's state depending on how states are defined. In requirements analysis, we choose event-based model to describe external observable behaviours of a system and its environment. Unlike state-based modeling that usually demands certain level of design effort, event-based modeling is an intuitive way in describing behaviours, especially from user's perspective. With event-based model, we focus on interactions between objects. In addition, event model can be easily used to construct system usage scenarios.

3.3.2 Object-events

In general, we use *Event* to describe something happening at a given place and time. In our case, in real-time reactive problem domain, we use object events to describe object behaviours.

An object may be capable of performing a set of behaviours of different types. For example, the gate has two different types of behaviours, *Open* and *Close*. Behaviour is an occurrence of a behaviour type, and we name it as object event. For instance, the gate can have a sequence of behaviour occurrences: *Open at 5 time unit, Close at 9 time unit, Open at 13 time unit, Close at 17 time unit.* In this sequence, there are two occurrences of behaviour type *Open* that these two occurrences are with different time attributes, or we say events *Open(5)* and *Open(13).* There are also two occurrences of behaviour *Close* with different time attributes, they are events *Close(9)* and *Close(17).*

Each object event is owned by a single object. An object event may indicate:

- Communication between two objects.
- A change of an object's state.

3.3.3 Object event attributes

An object event is always time stamped. In other words, it has a time value that is an attribute of the event and indicates the instance when the event occurs. In real-time system development, time is an important factor that concerns the behaviour. Time concepts are introduced to support the notion of quantified time for the description of real-time systems with a precise meaning of the sequence of events in time. Our events are instantaneous and do not consume time. Quantitative time values represent the time

distance between pairs of events. The time progress (a time unit) is equal for all instances in the model.

Besides the time attribute, some object events may come with some other attributes, namely, *parameter objects*. We name this type of object event as *parameterized object event*. Apparently, those events that do not have parameter object attributes are non-parameterized object events.

3.3.4 Object events classification

If an object event belongs to a system object, we name it as a *system event*. If an object event belongs to an environment object, we name it as *environmental event*. An environmental event is usually a stimulus to the system in a real-time reactive system and is assumed to happen. An observable system event belonging to a system interface object is named as system output event (refer figure 3-1) that is usually expected to happen as specified in the requirements. A system event belonging to a system internal object is named as system internal event. A system event owned by a system interface object can also be an internal event when it is not observable and not specified in the requirement.

Having the object event concept in mind, we can see that some objects that may have a set of behaviour types and are capable of having object events while other objects have an empty set of behaviour type and are not capable of having object events. We name those objects that are capable of having object events as *active objects,* and those objects that are not capable of having object events as *passive objects. Parameter objects* are usually passive objects. System interface objects and environment objects are usually active objects.

For example, a train in Railroad crossing system, an environmental object, is an active object because it can generate stimuli events to the system. In Production Cell case, parts introduced by the user to the production system, are passive objects since they can't generate any events, that they are the parameter objects conveying data information.

3.3.5 Behavioural characteristics and constraints on objects

With the understanding of above object-event modeling method, we should be able to model the characteristics of environmental objects and system objects and the relations between them.

Environmental characteristics include environmental objects types, quantities, and behaviour characteristics (mainly stimuli arrival patterns).

There may be some constraints on environmental object behaviours, which lead to the constraint on the occurrences of object events. Those behavioural constraints may be the following types:

-Ordering constraint: some object events must occur in an ordered relation.

-Quantitative constraint: some objects may have repeatable event while some object don't.

-Concurrency constraint: whether some events can occur concurrently as well as maximum concurrency event volumes arriving at a specific time range.

- Timed constraint: the time interval between two events may be limited to a range.

- Parameter constraint: event parameter objects may be defined.

- Interaction constraint: some object event may define interaction between two objects, especially environmental object and system object. Object interaction style may be specified also, such as synchronous vs. asynchronous.

We give some examples about the above constraints.

-An example about ordering constraints. In Railroad crossing case, object events of a train always have order constraints as: near → in → exit.

-An example about concurrency constraint. As there are multiple tracks in the crossing, multiple trains can traverse the crossing at the same time. Hence object events of trains' objects can occur concurrently.

-An example about timed constraint: In Railroad crossing case, for a train object, time interval between event *Near* and event *In*, between event *In* and event *Exit* are within the time ranges.

-An example about parameter constraint: In production cell case, event *Put* owned by a user object has a parameter attribute *part*.

System events are expected to occur under certain condition. The description of those expectations is a functional requirement. Each individual atomic functional requirement usually can be described by a pair of predicates: pre condition and post condition. These two conditions usually have parameters such as objects, events, event attributes including time stamp and event parameter objects. Post conditions usually contain system events as parameter. The Example can be found in section 4.5.

The time stamp, namely the occurrence time value of an event is usually important to real time system development. It can be used to measure the response time of the system. Some critical event can be even defined by certain timed condition.

By specifying each identified pair of pre/post condition, we can construct the whole functional requirement of the system to be developed.

3.4 System Usage model

Up to now, with modeling methods explained above, we are able to describe a system context, environment assumptions, and system functional requirements. The environment assumption and functional requirements acquired by the above approach can be clear and detailed. However they might be too fragmental since the semantics of those interactions and functions and links among them are lacking. To be more specific we need to understand how the system as a whole interacts with environment objects and what are the usages of the system from user perspective. System usage scenarios model are the way to achieve this aim.

3.4.1 Usage scenario

We apply a scenario-based method to model system usage from users' point of view. Scenarios are a very common and intuitive way for users to describe system usage and requirements. We can elicit system properties and implied assumptions by scenarios. Enumerating and analyzing scenarios can easily identify requirements. In addition, checking different scenarios usually can help detect missing requirements and requirement conflicts.

A usage scenario is a temporal sequence of interaction events between the system and its environment for archiving a usage goal. It is an instance of use for the system under development from users' perspective in requirement analysis phase. In object-oriented event modeling, objects in domain problem are identified and object behaviours

are described in terms of events. Objects and events outline system behaviour traces -- scenarios. Hence, a *scenario* can be modeled as a temporal sequence of events and each event in a scenario belongs to an object that owns it.

3.4.2 Usage model for real-time reactive system

For real-time reactive system, it is essential to understand its environmental behaviour before modeling the system behaviour, because timing requirements of a real-time system are determined and constrained by its environmental objects. Environmental objects of real-time reactive system behave in a timed fashion, where the system is required to give a response to the environmental events in a timed fashion. It is important to understand environmental objects behaviour and more important is their arrival pattern of stimuli events to the system, because it is crucial for analysis of system timing behaviour.

In real-time system, the first event in a scenario is always the trigger event from an environmental object. For example in Railroad crossing case study, in a scenario of a train traversing the crossing, the train's event *Near* is a trigger event in this scenario. Naturally, environmental stimuli must have their timing characterized. When we specify scenarios for real-time reactive system, we inject system events into behaviour pattern of environmental objects.

We can observe a train's three different events that constitute a train's traversing-the-crossing scenario. A train's observable events in the scenario are a train enters the crossing (event: Near); a train crosses the gate (event: In); a train leaves the crossing (event: Exit). These three events are of interest to gate-controlling system.

The timing features of three events are:

- 3 to 5 time units between a train's event *Near* and event *In*.
- 1 to 3 time units between a train's event *In* and event *Exit*.

The gate is an interface object in the gate-controlling system. It has two observable behaviours that are *open* and *close*. For the safety concerns, when trains are traversing the crossing, the gate should be *Close* before the first train's event *In* and can only *Open* after last train's event *Exit*. Gate behaviour of *Open* and *close* must be in time bound to satisfy trains' time constrains.

As we can see from the above Railroad crossing case, at requirement phase, we only concern external observable behaviours of system, (Gate's *open* and *close)*. We do not care how the system produces those observable results.

Chapter 4 Scenario based design evaluation

4.1 Introduction

In previous chapter, we have illustrated a scenario-based approach in requirement modeling and analysis. The output of the requirement analysis is the foundation that will guide the following development process. In this chapter, we extend the scenario-based approach so that we can further utilize this output to evaluate design artefacts.

Certainly, requirement analysis output can also be used to validate implementation or even to generate test cases for testing purpose. However, there are some apparent benefits that drive this work to be focusing on design evaluation. These benefits include:

- Cost effective

  Design evaluation can expose errors of design artefact before the implementation phase thus can greatly reduce the development cost. Correcting errors in design phase cost much less than in implementation phase.

- Performance prediction

  The Performance of the final product usually affects users' satisfaction. Although some factor related to the implementation affects the performance, design is usually the main variable that determines the performance. In general, by evaluating design the performance is usually predictable. It can predict system behaviours for different predefined performance metrics. By predicting performance, we can identify unsatisfactory performance of design artefacts in the early stages of the development process, not wait until final product is implemented.

- Choice and comparison among different design choices

  Under the same requirement, design may have different solutions. Different design solutions may have strength on different performance aspects. Design evaluation provides a way of evaluating them according to a set of predefined criteria.

- A measurement for design optimization

  Knowing a predicted performance of a specific design allow the designer to optimize the design and measure the optimization result.

Aiming at these advantages, we proposed and developed a design evaluation approach that has the following characteristics:

- Black box strategy

  Black box strategy creates an execution model to simulate the execution of the intended design solution, and the evaluation work is conducted on simulation output. We apply the black box strategy in our evaluation approach for its simplicity and effectiveness.

- Scenario based evaluation

  By applying simulation method, we can generate simulated execution scenarios. These simulated scenarios serve as the input to the evaluation of the performance.

- Automatic evaluation

  Manual evaluation is certainly an intuitive and effective way to inspect the simulated output. However, when large amount of testing is involved, automatic

evaluation can help improve the productivity dramatically. Formalism is applied in the design evaluation that makes automatic evaluation to be realized.

4.2 Design performance evaluation process

In general, our design evaluation method simulates the execution of a design and analyzes the simulation output in order to evaluate the performance of a design. The diagram in figure 4-1 illustrates the data flow of the evaluation process as well as its relationship with the requirement modeling and analysis process. As we already know, by requirement modeling and analysis we can produce specification document including system requirements and environment assumption and constraints. This output is the input of the design process, in which the designer must target at it. The output is also the input of our design evaluation process, in which we can define evaluation metrics from it. The output of the design process, namely the design specification, is the input to our evaluation process in which we evaluate it.

Our design evaluation process includes two steps: *simulation* and *evaluation*. In the simulation step, we simulate the execution of the system based on the design to be evaluated. In the evaluation step, we evaluate the output of the simulation step according to a set of evaluation criteria.

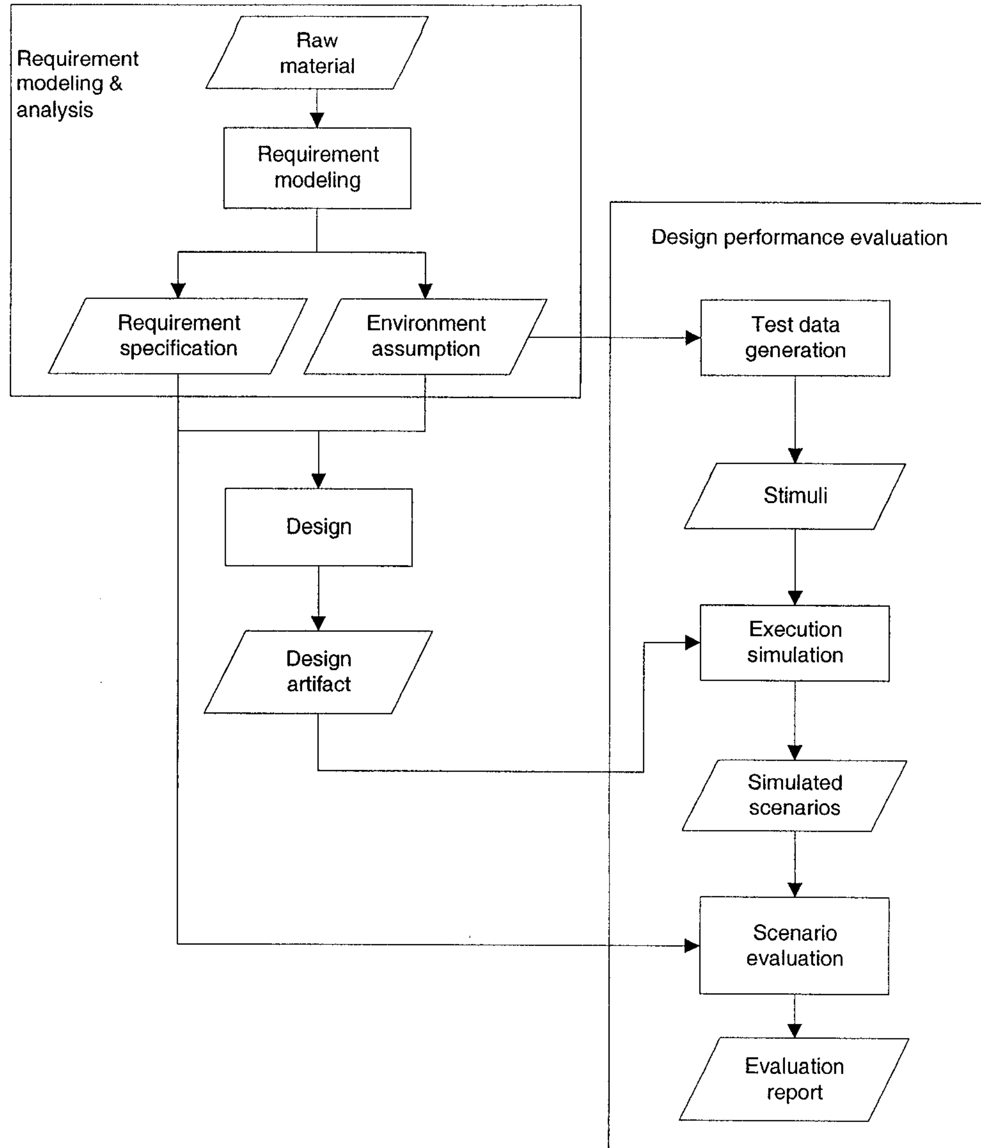

Figure 4-1 Scenario-based design evaluation process

The execution of a system is triggered by its environment. Therefore, the simulation step involves two functional parts: design *execution simulation* and *environment data generation.*

A design execution simulation process receives environmental stimuli generated from environment data generation process, computes and generates system behaviours, which are represented as sequences of system events. The generation of system events should strictly conform to the design specification. Simulation process is conducted in TROMLAB simulator [15,16,17]. The execution simulation process produces system events including *system internal events* and *system output events.* System output events are generated by system interface objects. They should be recognizable from users' point of view since they are specified in the requirement. System internal events are unrecognizable since they are not specified in the requirement.

These events come with environment stimulus events during simulation. We combine system output events and environment events in one run of simulation to generate an event sequence in totally timed order. We name such combined event sequence as a *simulated scenario.* The environment event sequence in a simulated scenario is an *environment stimuli scenario* that usually represents one scenario of system usage. Simulated scenarios are the output data to be evaluated in the evaluation process.

Environment data generation process generates stimulus events as input to the design execution simulation process. Simulated scenarios produced in execution simulation process are the input data of the evaluation process by which we can evaluate the performance of a design.

Up to now we have introduced the process of the design evaluation. However, different applications can be very different in design evaluation depending on application characteristics. Different performance attributes and different environment data generation schemes are applied in different applications. In the following sections, we will elaborate these issues in detail.

4.3 Performance evaluation model

As we already introduced, our design evaluation approach adopts a black box strategy. The execution simulator integrated with the design specification forms a simulated system, by which we can 'run' the system, test the system with environment input, and evaluate the test result. Therefore, to evaluate a design for a specific application, we need to know the following:

<1> What are the output data that need to be evaluated (section 4.2);

<2> What are the metrics in the evaluation; (section 4.3.1)

<3> What are the test data we should produce in the evaluation. (section 4.4)

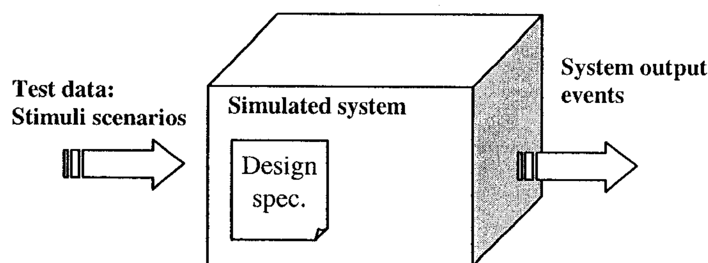

Figure 4-2 Simulated system and black-box design evaluation

4.3.1 Define performance metrics

Performance evaluation targets at externally observable system performance characteristics [24]. Performance attributes are identified from the user's perspective.

Performance function

Defining a design performance metric usually can be regarded as specifying a performance function *m(SS)*, which takes in a simulated scenario set *SS* as an argument.

The *performance function m(SS)* can be further refined as an aggregation function *g* and a single scenario metric function *f* such that

$m(SS) = g(\ f(\text{ss}), \text{SS}\ )$, where $ss \in SS$ ------------------------------------(4-1)

f is a *single scenario performance function* based on each individual simulated scenario and its argument *ss* is a simulated scenario that belongs to the set *SS*. The aggregation function *g* aggregates outputs of *f* for all simulated scenarios *ss* in the set *SS*.

As we know, each simulated scenario *ss* can be regarded as the output of a simulation function *exSim()* with the input of a *design artefact* and an *environment stimuli scenario es* (es belongs to an *environment stimuli scenario set ES*). We can describe it as the following:

$exSim\ (design_artifact,\ es\ ) =\ ss$, where $ss \in SS$ and $es \in ES$ ------(4-2)

$exSim\ (design_artifact,\ ES\ ) = SS$ --(4-3)

We combine (4-1),(4-2) and (4-3) , we get m function as follows:

$m(SS) = g(\ f(\ exSim(design_artifact,\ es\ )\ ),\ exSim\ (design_artifact,\ ES\ )\ ),$

where $es \in ES$ --(4-4)

Defining metrics

From the formula (4-3), we can conclude that designing a scenario-based performance evaluation metric can be done by defining the following three elements:

1) ***f*** –

This is a single scenario performance function that is computed based on a single simulated scenario. This is the basic function that contributes to the performance metric.

For example: computes response time value from each single simulated scenario so that average response time can be calculated based on each single scenario response time result.

2) ***g*** –

This is the aggregation function that is computed based on each output of single scenario metric function f in terms of a set of simulated scenario SS. We need to specify what f is used in the aggregation and how the output of each f is aggregated.

The aggregation function may be or may not be a single computed value. For example the aggregation function for average response time is to compute an *average* value of response time for all simulated scenarios and this function outputs a single value.

When it comes to system inability, the evaluation output of system inability is a collection of environment event sequences with which the design cannot produce

satisfactory output. In this case, the output of aggregation function is not a single value.

3) ***ES*** –

This is the simulated environment scenario set, in which each simulated environment scenario *es* represents a scenario of possible usage of the system and can be regarded as test case input. An *es* together with the system output events construct the simulated scenario *ss*.

Defining a design performance metric usually requires us decide what system usages should be tested. This is an important factor that contributes to the quality of the metric to be evaluated.

Different types of applications might be suitable for different set of metrics that construct meaningful evaluation result. For example: Maximum throughput metric is necessary to conduct in production cell case.

In the following sections, we enumerate some commonly applied metrics in design evaluation and analyze their patterns on defining *f*, *g*, and *ES* elements for each metric.

4.3.2 System Output Correctness

Correctness is a vital and integral performance metric for all cases since it is the first attribute we want to evaluate before any others. Evaluating the correctness of a design is to validate and verify whether a design satisfies our requirements. Since our evaluation approach is a black box approach, our correctness metric focuses on requirement

validation, in other words, validating whether system outputs are correct according to our requirements based on a canonical set of environment stimuli scenarios as test input.

We follow the performance formulas we introduced in the last section to define the generic correctness metric, which includes an aggregation function *g*, a single scenario metric function *f*, and an environment stimuli scenario set *ES*.

1) Single scenario performance function *f*

The single scenario performance function *f(s)* in correctness metric should reflect whether a design satisfies system requirements in terms of a single simulated scenario s. Therefore, *f(s)* is a Boolean function that tells us whether the system outputs are correct in the simulated scenario. If *f(s)* returns true, we say *s* is a *legal simulated scenario* or *legal scenario,* or *s* is *legal.*

To design such *f*, our system requirements, especially functional requirement, need to be formalized so that a computable *f* can be constructed. We established a scenario legality model by which our requirement can be specified into *f* and the legality of a scenario can be evaluated.

A simulated scenario is a timely ordered event sequence. We name a consecutive sub-sequence of a simulated scenario and that doesn't miss any event before itself as its *scenario-segment* (See Figure 4-7). Since there could be some concurrent events, a scenario-segment defined here should include all the events in its time span. For example, e_{i+1} and e_{i+2} are two events with the same timestamp, and therefore a scenario segment s_{i+1} of s should contain both of them. A simulated scenario s has limited scenario segments including the initial scenario-segment that has no event and itself.

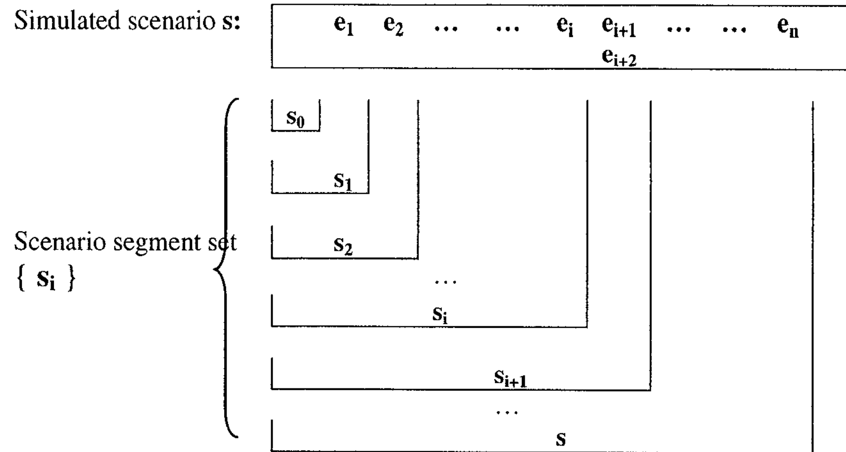

Figure 4-3 Scenario segments set of a simulated scenario

We can conclude that a simulated scenario s is a legal scenario if and only if all its scenario segments { s_i } are legal.

Here s_i represents one scenario segment of s, s_0 represents the initial scenario segment, and { s_i } represents the complete scenario segment set of s. In addition we also use s_{i+1} represents s_i's *next scenario segment* in s. If s_{i+1} is s_i's *next scenario segment* in s, that means s_{i+1} has more events than s_i, and there is no such *scenario segment* in s that has more events than s_i and less events than s_{i+1}.

From the above conclusion, we can see that the legality of S can be evaluated by evaluating the legality of each scenario segment in { s_i }. We can utilize *next scenario* segment relation to create an inductive method, by which we can evaluate the legality of each scenario segment in { s_i } and define the function *f*.

As we know, s_0 as an initial empty scenario segment is always a legal scenario. If we find a way to specify the legality function of s_{i+1} by assuming s_i is legal, then we can compute the legality of s_1, and then s_2, s_3,…, until the whole set is computed.

The functional requirement of a system usually describes system observable behaviours, which are about "what a system must do and what a system should not do. Each do/don't can be expressed as a pair of pre/post conditions. We apply this form in our specification and adapt it into the scenario legality model. Therefore, our requirement can be expressed as a set of *legality rules {Rj}* and each rule can be defined as:

$$Rj(s_i, s_{i+1}) = P(s_i) \rightarrow Q(s_{i+1}) \text{ ---------------------------------(4-5)}$$

An Rj represents a legality rule. $P(s_i)$ is a predicate function of scenario segment representing the pre-condition and $Q(s_{i+1})$ is a predicate function of scenario segment representing the post-condition. $P(s_i)$ takes in a scenario segment s_i as input to evaluate if the pre-condition is satisfied and $Q(s_{i+1})$ takes in s_i's next scenario segment s_{i+1} as input to evaluate.

If s_i is a legal scenario and Rj is computed to be true, then we can say s_{i+1} satisfies the legality rule Rj. If all the rules in {Rj} are satisfied, namely $R_1(s_i, s_{i+1}) \wedge R_2(s_i, s_{i+1}) \ldots \wedge R_N(s_i, s_{i+1})$ = true, we can conclude that s_{i+1} is legal.

The semantics of the rules set {Rj} in this context can be explained as: given a legal scenario segment, {Rj} defines what next event(s) is allowed to happen and not allowed to happen. By this way, we can compute the legality of s_1, s_2, … , and finally s.

Hence, the single scenario metric function *f* can be defined as:

$$f(s) = \forall s_i \bullet \ s_i \in \{ s_i \} \ \bullet (\ \forall r \bullet \ r \in \{ Rj \} \bullet \ r(s_i, s_{i+1})) \text{ -----------------(4-6)}$$

where { s_i } is the complete scenario segments set of s, s_{i+1} is s_i's next scenario segment, and {Rj} is the legality rule set.

This formula establishes a generic computable evaluation model for the legality of simulated scenarios. We can follow and extend it in different applications by transforming functional requirements into a legality rule set. With such model we can easily realize automatic requirement validation.

2) Aggregation function *g*

Given a set of environment stimuli scenarios, we need to evaluate all scenarios in the set in order to compute the correctness metric. Apparently, the output is a Boolean value and can be expressed as:

$$g(\{f\}) = \forall s \bullet s \in \{SS\} \bullet f(s) \text{ -----------------(4-7)}$$

Besides this output, however, if g is evaluated as false, the simulated scenarios that are not legal should be included in the output set, since we usually need to know those illegal scenarios by which design faults may be identified.

3) Environment stimuli scenario set *ES*

Certainly if possible and not too costly we should use full set of possible environment stimuli scenario to test design artefacts. However, there are quite a few of cases where we have to reduce the set to a size so that we can afford. And reducing evaluation effort on repeating similar test cases is a wise way to minimize cost. Therefore, in this case, ES should be a representative subset of the full set.

Selecting a representative subset could have different ways depending on characteristics of applications. In correctness metric, the coverage of ES is very important since the selection of ES directly affects the adequacy of the evaluation of the design

artefact and hence affects the evaluation quality. There are some ways on checking the coverage of ES.

1) Usage coverage

   The canonical set should have enough coverage on the possible usage of the system.

2) Important factor coverage

   An application usually has some important factors in requirements that affect the system behaviour and system usage a lot. For example, time is an important factor in real time systems. By checking coverage on each factor, we can identify some missing usage.

Besides correctness metric, there are some others we want to evaluate. In the following we analyze other three common criteria.

4.3.3 Inability

By the name, it seems inability metric is just correctness metric with different name. The difference between the two metrics is that even though a system design is proved to be correct, we still want to know to what extent a design can function well (i.e. in which environmental assumptions make the design functioning well) and which interaction patterns can make a design fail. Knowing this performance will give users more knowledge and confidence on environment context (environmental assumption) for the design. Inability metric is designed for this purpose.

Single scenario performance function *f*

In inability metric, the evaluation of a single simulated scenario is the same as the counterpart in correctness metric. The function f is the same as formula (4-6).

Aggregation function *g*

Instead of giving a collection of identified illegal simulated scenarios, an inability evaluation should produce more meaningful result. We can enumerate a set of factors and the evaluation should produce the capability range based on each factor. Therefore, g is a collection of capability ranges: $\{ g_i (\{(f(s), s)\}) \}$, where $s \in SS$ and SS is simulated scenario set. Each element $g_i (\{(f(s), s)\})$ is a function that computes the capability range of a concerned factor with the input of $\{(f(s), s)\}$.

Example 1: Production cell case study

The number of parts of the same type that user can continuously put on the belt depends on the system's volume to tolerate the same type of parts. When user put the same type of parts that exceed the system's tolerated volume, the production cell has to stop functioning. We should have a $g_i (\{(f(s), s)\})$ to reflect that range.

Example 2, Railroad Crossing case study

A train's speed is a concerned factor and we want to know what range of train speed is suitable for a system design. In this case, we need to have a $g_1 (\{(f(s), s)\})$ that gives the train speed range.

Environment stimuli scenario set *ES*

ES in inability criteria is different from that in correctness criteria. Some scenarios that exceed the constraints specified in environment assumption usually need to be included. In addition, the selection of canonical scenarios should be based on each factor range function accordingly.

4.3.4 Average response time

System response time is an important metric to evaluate a design performance, especially in real time reactive systems.

Single scenario performance function *f*

Response time of a design in a single simulated scenario can be measured by the time interval of two events: the stimuli event and the system output event.

Therefore, *f* can be defined as:

f (s) = output(s).time-stimuli(s).time

In this formula, stimuli(s) and output(s) are two functions that need to be defined in order to identify two important events. stimuli(s) identifies and returns the stimuli event in the input scenario s and output(s) identifies and returns the system response event in s.

Aggregation function *g*

Average response time computes the average value of response time computed from each simulated scenario.

g = average({(*f*(s), s)}), where $s \in SS$ and SS is simulated scenario set.

Environment stimuli scenario set *ES*

The selection of ES may be similar to that in correctness metric. We can also specify selection criteria based our concerned factors.

4.3.5 Maximum throughput

Maximum throughput is a useful metric to reflect the throughput capability of a design. This is especially suitable for applications like production cell case. However it is not suitable for Railroad crossing case.

Single scenario performance function *f*

f(s) is to compute the throughput of a single scenario. We can define it as the following:

f(s) = output(s) / time_span(s)

where output(s) and time_span(s) are two functions that need to be defined.

output(s) computes the total output in the scenario s. This function is usually computed based on the parameters of system output events.

time_span(s) computes the time span in s. It can be further defined as:

time_span(s) = last_event(s).time – first_event(s).time

Aggregation function *g*

g is to identify the maximum value of throughputs computed by *f* for each scenario in SS. We can define g as:

g = maximum({(*f*(s), s)}), where $s \in SS$ and SS is simulated scenario set.

Environment stimuli scenario set *ES*

Normally, system throughput has a linear relation with system input stimuli in some range and the throughput stop increasing when it reaches its capacity. The selection of ES should follow this rule so that maximum throughput can present in the evaluation.

4.4 Environment Data Generation

4.4.1 Canonical set

When we look at the three elements in a performance function, we notice that the aggregation function g and the single scenario performance function f in a performance function m are easy to define. However, deciding on the environment stimuli scenario set ES is quite difficult comparing to the formal two elements, since usually the size of the full set of environment stimuli scenarios in some applications might be quite large and some may be unbounded.

The purpose of the environment data generation process is to produce the environment stimuli scenario set ES for each design evaluation metric. The outputs are collections of test cases that are used to test a specific design.

It's costly to evaluate a design with an unbounded sized possible environment scenario set. Therefore, we choose a subset as ES in the evaluation. There are generally two ways to define ES.

1) Scenario enumeration

In this method we enumerate all concerned scenarios based on our understanding of the usage requirement and some experience. This is very natural and effective, especially when the user wants to do a partial evaluation or trace detailed system behaviour. However, when the size of possible scenarios become large, it is not easy to enumerate manually and guarantee enough coverage of the enumeration set.

2) Systematic selection

In this method we select a subset based on some rules and criteria, which are called *canonical scenario set* [31]. This way is suitable when you want to have a systematic evaluation and it can contribute an automatic evaluation with some tool support.

Scenario enumeration is easy and direct. In the following we will discuss the details of the systematic selection.

4.4.2 Guideline on systematic selection

To do systematic selection on ES, the following steps are required:

1) Analyze the application characteristics and identify key factors (e.g. stimuli events etc.) that affect system behaviours.
2) Design a method to partition the all possible scenarios set based on the combination of identified concerned factors, and decide test range for all factors.
3) Design canonical selection rule and select canonical scenarios from each partition to construct a canonical set ES.

In addition, designing and applying a selection method for a canonical scenario set usually requires us to consider the following issues:

1) Environment assumption

   An environment stimuli scenario cannot be arbitrary. It should follow the environment assumption and conforms to its constraint. For example, in railroad crossing case, a train's event sequence must in the order of "near→in→exit", any other orders should not present.

2) Usage coverage and partition granularity of possible usage scenario set

   The coverage of the canonical set is important since usually we wish to test the design as fully as possible. However, the larger the canonical set we use, the more

the process costs. The partition method and the granularity of partition area are two important factors that affect the coverage and the size of the canonical set.

3) Application characteristics

Application characteristics affect a lot in canonical set selection. For example, for timed critical application, timing is an important factor to be considered in the selection method. For production system, such as production cell case study, the ordering of the input is an important factor.

4) Characteristics of the metric to be defined

Different metrics require different subset of scenarios to be tested. For example, maximum throughput in production cell example introduced in section 3.2.6, requires higher rate of input environment stimuli scenarios. System inability metric requires larger set of environment stimuli scenario set than correctness metric does.

4.4.3 Test data generation models

In general, interaction scenarios between a system and its environment can be numerous and complicated. Since the application domain that we are studying is real time reactive system type, we can assume that in most applications interactions between the system and its environment are reactive type. This makes our modeling easier since complicated interactions are out of the scope of our present work.

A generic real time reactive system and its environment can be described as follows:

- The system as a whole is an object that has a limited set of event types. Each object event type may have some parameters.

- The environment consists of *a set of active objects*, each of which has a limited *set of event types*. Each event type may have some *parameters*.

If we compute all possible scenarios by considering all possible number of objects, all possible event types, all possible parameter values, all possible time schedules, resulting set can be unbounded. Hence, a universal approach is not possible. However, a real application may not have such complexity on all dimensions. We can identify some interaction patterns and design proper event generation solution for each pattern. We introduce three typical interaction patterns and discuss about the relevant canonical set selection.

Pattern I

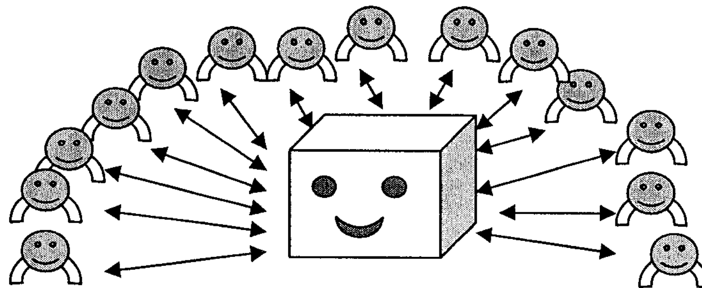

Figure 4-4 Interaction pattern I

In this interaction pattern, there is a large number of environment stimulus events dynamically generated. All interactions follow a simple request and response model. Typical examples of this pattern are: web service system, monitor system, radar system, etc.

In this pattern, event ordering and event parameters are usually out of the concern. What really concerns us is dynamic number of concurrent arrival stimulus events. The

environment data generation should reflect the dynamic number of concurrently generated events in each time unit.

Therefore, the selection of ES can be based on number of concurrent stimulus events at a given time. Time and concurrent event volume are the two factors by which we can partition the whole possible test data set (Figure 4-4).

The partition granularity, namely length of time partition unit and size of concurrent volume partition unit, and the test range, namely the time range and the concurrent volume range, should be decided based on individual application characteristics.

The next step is to decide selection criteria. This decision should consider issues that we already mentioned, such as environment assumption, usage coverage, etc. For example, the dynamic volume change usually is not arbitrary, we may follow some dynamic range assumption to reduce some unreasonable scenarios.

In addition, we may introduce composite factors based on existing factors, such as high volume sustaining time, to assist our data selection.

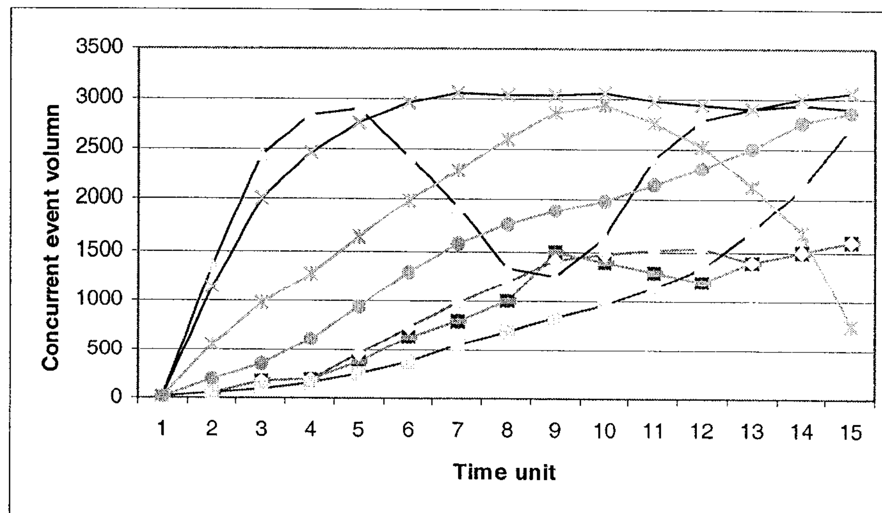

Figure 4-5 Partition and selection

Pattern II

In this interaction pattern, the interaction between a system and its environment does not involve concurrency and it receives different types of events or same type of events with different parameters, and outputs simple events of same type. Its output usually can be affected by the ordering of input events. A typical example of this pattern is production line case introduced in section 3.2.6.

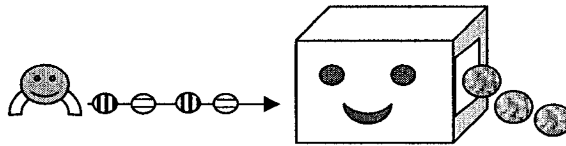

Figure 4-6 Interaction pattern II

In this type of systems, event ordering and input event occurring speed are our concerned factors. Event ordering affects correctness measurement data and inability measurement data, while input event speed affects maximum throughput measurement data. Event ordering is a generic factor. When it comes to a real application, some composite factor may be more effective to apply. For example, in Production line example, consecutive number of the same input part is an effective factor, that can be applied in partitioning scenarios and selecting canonical scenarios.

Pattern III

In this interaction pattern, each environment object must produce a chain of events conforming to predefined order, and event chains occur concurrently. Typical examples of this pattern are railroad-crossing system, order processing system, etc.

In this interaction pattern, both time and event ordering are concerned factors. However, these two factors are overlapped and in such case we can't choose both as partition factor at the same time. The most intuitive way is: we can use event ordering as partition factor, and then we can use time factor to select canonical scenarios.

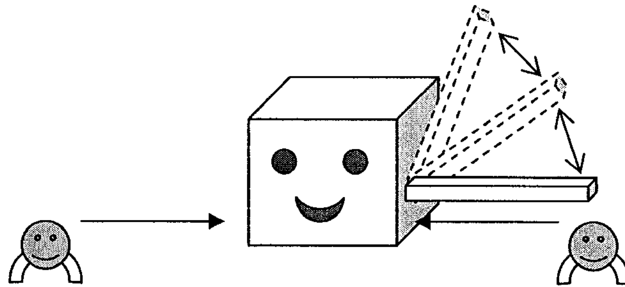

Figure 4-7 Interaction pattern III

4.5 Case Study --- Evaluating the design of Generalized Railway Crossing

We introduced the Railroad-Crossing problem and showed a design solution of this problem in chapter 2. We will apply our evaluation theory explained in this chapter to evaluate this design solution.

As we said before, different type of application might be suitable for different set of performance metrics that can construct meaningful evaluation result. Hence the first thing in the evaluation activity is to identify performance metrics. In this case we include <correctness, response time, inability> in our performance evaluation since these attributes are applicable for railroad-crossing case. We follow the performance formulas that we introduced in the section 4.3 to illustrate these performance metrics, which each

metrics should include a single scenario metric function *f*, an aggregation function *g*, and an environment stimuli scenario set *ES* respectively.

4.5.1. System Output Correctness

This is to check whether the design can produce output that complies with the requirement. For doing this, the following tasks need to be done in the evaluation process.

- Task1: Elicit rules in scenario legality check

We have explained our legality model for correctness validation in section 4.3.2, in which selected simulated scenarios must pass all legality rules in order to become legal scenarios. We show legality rules in this case that can evaluate simulated scenarios. By analyzing stimuli-response functionality of the Railroad crossing system, we conclude with the following three legality rules:

<Rule 1>: The gate must be closed before any train arrives in the crossing.

<Rule 2>: Gate can't open when there is a train in the crossing area.

<Rule 3>: The gate must open when all trains exit.

In our legality rule model, these rules can be defined as $Rj(s_i, s_{i+1}) = P(s_i) \rightarrow Q(s_{i+1})$ according to the formula (4) we introduced in section 4.3.2, refer formula (4) about definition of s_i, s_{i+1}. We get the following rules representations based on formula (4) for this case.

<Rule 1>:

P: When scenario-segment is not empty AND has no event "*closed*".

Q: It should not have event "*In*" at the next non-empty time unit.

<Rule 2>:

P: When scenario-segment is not empty AND numbers of event "*Near*", "*In*", "*Exit*" are not equal.

Q: The events at the next non-empty time unit are not supposed to be "*opened*".

<Rule 3>:

P: When scenario segment is not empty AND number of event *Near*, *In*, *Exit* are equal with the last event of the scenario segment as Exit,

Q: The events at the next non-empty time unit are supposed to be *opened.*

So far, we got rules for checking a scenario. Correctness metrics is to check whether system outputs are correct according to our requirements based on a canonical set of environment stimuli scenarios as test input. Next we need to get canonical environmental scenario set for archiving this goal.

- Task 2: Select canonical environmental scenario set

We use this case as an example to show how to select canonical environmental scenario set based on system characteristics.

In this case, several trains may traverse the crossing concurrently, therefore the scenarios may contain different numbers of environmental objects. We start with a single train traversing since this is the basic information to get scenarios with multiple environmental objects. [31]

When a single train traverses the crossing, there are following possible scenarios according to environmental assumption mentioned in chapter 2 case study:

1. Near (0), In (3), Exit (4),
2. Near (0), In (3), Exit (5),
3. Near (0), In (4), Exit (5),
4. Near (0), In (3), Exit (6),
5. Near (0), In (4), Exit (6),
6. Near (0), In (5), Exit (6),

7. Near (0), In (4), Exit (7),
8. Near (0), In (5), Exit (7),
9. Near (0), In (5), Exit (8),

The events in every scenario will only involve events that need to be *observed* for the evaluation. Hence, for the Train object the events are *Near*, *In*, and *Exit*. Trains may come at different times that *Near* event can be any time value. However our evaluation is independent of *Near* event time, it is sufficient to consider valid sequence with *Near (0)*. [31]

As we can see, in the situation of single environmental objects, environmental scenarios have the same *sequence pattern* (*Near, In, Exit*) namely the same event ordering, but with different event time attributes that indicates trains traversing in different speeds.

Now we have a completed scenario set for a single object interacting with the system. These are basic configurations about potential trains traversing the crossing, which we can use these configurations to construct multiple environmental object scenarios.

For instance, in the situation of two trains traversing the crossing, two trains can be any two of the nine configurations, and they may approach the crossing in different time. We can get all sequence patterns of two trains situation by combining the configurations of two trains. An example of a sequence pattern for two trains is as follow:

train1.Near,train2.Near,train1.In,train2.In,train1.Exit,train2.Exit

And an example of a scenario for two trains is as follows:

train1.Near(0),train2.Near(1),train1.In(2),train2.In(3),train1.Exit(4),train2.Exit(5)

By this way, we can get the sequence patterns and environment scenarios for any number of trains. We need to consider how to select canonical set of scenarios among them since it is not wise to evaluate all of them. Coverage issue is important in this case since the system must be able to handle all environment situations for safety concerns. Therefore all of sequence pattern should be preserved for coverage consideration [30] [31].

The whole scenario set is partitioned by sequence patterns so that scenarios in each partition have the same sequence pattern, in other words the same event ordering, but with different event time attributes. To get a canonical scenario set is to select canonical scenarios in each partition based on some criteria, which selected scenarios from each partition constitute a canonical environment scenario set. Defining proper criteria is the key in selection jobs in which criteria is related to functional requirement. We can elicit and define criteria according to function points and the coverage issue is about function coverage, in other words the criteria must consider and cover all function points.

In this case, we will give criteria that elicited from function points of this case. There are two timing function points in this case that are opening and closing gate. System output events *Open* and *Close* will be injected into each environment scenario to construct simulated scenarios. So when we select canonical scenarios from each partition set, selected scenarios should be representative in terms of these two timing function points. Here are two rules for selecting canonical scenarios from each partition.

Selection Rule 1) Consider time distance between first *Near* event and first *In* event of each environment scenarios in a partition, and select three scenarios that have minimum, maximum, middle value of time distance between these two events.

Selection Rule 2) Consider time distance between last *Exit* event and first *Near* event of each environment scenarios in a partition, and select three scenarios that have 0, 1, 2 time units time distance between these two events.

Certainly it has coverage advantage when selecting more scenarios, however we have to be aware that evaluation cost will be increased accordingly. So, it is an empirical decision about how many scenarios should be chosen from each partition. In this case, we decide to select three scenarios.

Let's look at a scenario partition of this case. Assume we have following environment scenarios in a partition set. Based on the above selection rules, we will choose three representative scenarios for each rule.

1.--train1.Near(0),train1.In(3),train1.Exit(4), train2.Near(4), train2.In(7) ,train2.Exit(8)
2.--train1.Near(0),train1.In(3),train1.Exit(4), train2.Near(5), train2.In(8) ,train2.Exit(9)
3.--train1.Near(0),train1.In(3),train1.Exit(4), train2.Near(6), train2.In(9) ,train2.Exit(10)
4.--train1.Near(0),train1.In(3),train1.Exit(4), train2.Near(7), train2.In(10) ,train2.Exit(11)
5--train1.Near(0),train1.In(3),train1.Exit(4), train2.Near(8), train2.In(11) ,train2.Exit(12)

Select scenarios according to rule 1:

As each scenario has the same time distance between first *Near* and first *In* event, it is the same to choose any three scenarios. Let's say we choose first three.

Select scenarios according to rule 2:

The first three scenarios satisfy the rule 2 since they hold 0,1,2 time distance between last *Exit* event and first *Near* event.

Now we get two sets of selected scenarios for two rules individually. We found two sets are exactly the same. Therefore it is enough to have one set as a canonical scenario set for this partition.

In chapter 5, we have implemented a tool to generate and select a canonical environment data. By interacting with the tool, we will get a whole set of environment data with different number of environmental objects generated from the tool.

- Task 3: Get simulated scenarios

Canonical scenarios will go through simulation execution and evaluation process. We got three canonical scenarios in last example. These three scenarios and a version of design artifacts are input to simulation execution to construct three simulated scenarios. The expected result is as follows:

*1-- train1.Near(0),**gate.Close(1)**,train1.In(2),train1.Exit(4), train2.Near(4), train2.In(7) ,train2.Exit(8),**gate.Open(10,11)***
*2-- train1.Near(0), **gate.Close(1)**,train1.In(2),train1.Exit(4), train2.Near(5), train2.In(8) ,train2.Exit(9), **gate.Open(11,12)***
*3-- train1.Near(0), **gate.Close(1)**,train1.In(2),train1.Exit(4), train2.Near(6), train2.In(9) ,train2.Exit(10), **gate.Open(12,13)***

-- *gate.Open(10,11)* means there are two possible time attribute of this event that are 10, or 11.

However, when we use chapter 2 design version in simulation execution, we got simulated scenarios of No 2, 3 as follows:

*2'-- train1.Near(0), **gate.Close(1)**,train1.In(2),train1.Exit(4), train2.Near(5), **gate.Open(7)**,train2.In(8) ,train2.Exit(9)*
*3'-- train1.Near(0), **gate.Close(1)**,train1.In(2),train1.Exit(4), train2.Near(6), **gate.Open(7)**, train2.In(9) ,train2.Exit(10)*

- Task 4: Evaluating simulated scenarios against rules

Now we evaluate three simulated scenarios with evaluation rules defined in Task 1.

For each simulated scenario, we have single scenario metric function $f(s)$ as follows:

$$f(s) = \forall s_i \bullet \; s_i \in \{ s_i \} \; \bullet (\; r_1(s_i, s_{i+1}) \wedge r_2(s_i, s_{i+1}) \wedge r_3(s_i, s_{i+1}) \;)$$

For correctness metric, each $f(s)$ result has to be true, so the aggregation function is as follows:

$g(\{ f \}) = \forall s \bullet \; s \in \{ SS \} \; \bullet f(s)$, *where SS is the set of simulated scenarios generated from environment canonical set.*

We have used our tools to generate and select canonical scenario set and used our evaluation tool to check scenario with our rules. The evaluation result shows scenario No 2', 3' violate evaluation rules No 2 and No. 4. We located where in the design may cause the violation by following the indication of legality rules.

Figure 4-8 showed a modified design solution in which a transition is added. We redo the evaluation process with the modified design version, the violation problem is disappeared in this version.

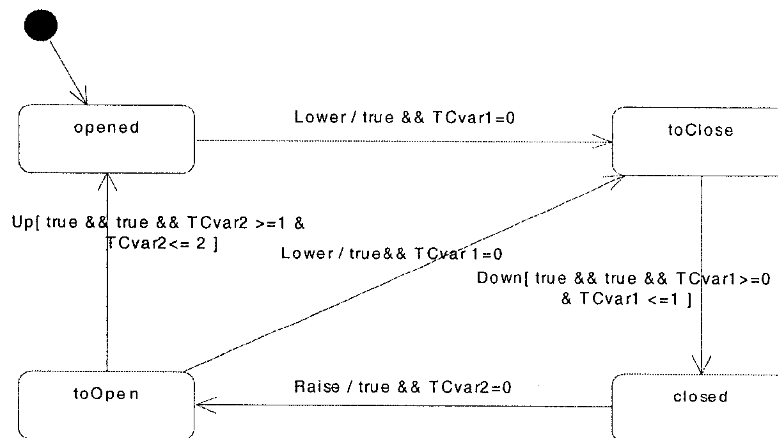

Figure 4-8 Modified design version

4.5.2. Evaluating response time

For a given scenario, response time metric is to evaluate time distance between two specific events. In this case, we may want to evaluate response time – time distance between *gate.open* event and *train.exit* event that trigger the gate opening.

We give a function that can get time of last-train-exit:

LastTrainExitTime(s)= getEventTime (getEvent(lastExitEvent, s), s);

We give a function that can get time of gate-open:

GateOpenTime(s)=getEventTime(getEvent(gateOpen, s), s);

Therefore, the function to calculate time distance is as below:

Function *f(s) = LastTrainExitTime(s)– GateOpenTime(s);*

We give a scenario example below to calculate its time distance between *gate.open* event and *train.exit* event.

Example scenario s1:

train1.Near(0),gate.Close(1),train1.In(2),train1.Exit(4), train2.Near(4), train2.In(7) ,train2.Exit(8),gate.Open(10)

Calculation:

LastTrainExitTime(s1)= getEventTime (getEvent(lastExitEvent, s1), s1)=8

GateOpenTime(s1)=getEventTime(getEvent(gateOpen, s1), s1)=10

f(s1) = GateOpenTime(s1) - LastTrainExitTime(s1)=2

By evaluating all simulated scenarios in the set SS, we can get aggregation information such as shortest gate opening time, longest gate opening time, or average gate opening time.

g= maximum ({(f(s),s)}); where s∈SS and SS is simulated scenario set.

g=minimum ({(f(s),s)});

g=average ({(f(s),s)});

4.5.3. Inability Evaluation

Certain environment stimuli patterns are out of system ability to handle, in this situation the system can't give satisfied response. Evaluating inability is to find out the range of these stimuli that the system can't handle.

In this case, the system needs at least *two* time units to close the gate after receiving first *Near* event. When gate is opened, and a train sends event *In* in less than two time units after sending event *Near*, the system can't response correctly in this situation (to close gate). For example in below situation, the system can't close the gate in time:

*train1.Near(0),train1.In(1),**gate.Close(2)**, train1.Exit(2),gate.Exit(3)*

Hence, the range of stimulus data that the system can't handle is the trains that have time distance less than *two* time units between event *Near* and *In*.

Chapter 5 Tool support for automatic evaluation process

Up to now, we have introduced our approaches on object-event-scenario requirement modeling and scenario based design evaluation. In this chapter, we are going to elaborate our idea on the tool support for automatic evaluation process.

5.1 Design evaluation system

Tool support is very important in software engineering process since effective tool support can reduce developers' work, improve developers' productivity, and realize some automation process. In our case, we planed to develop a set of software tools to assist our scenario-based requirement engineering process including system context modeling and automatic design evaluation. In the current stage, our tool support focuses on processes of automatic design evaluation.

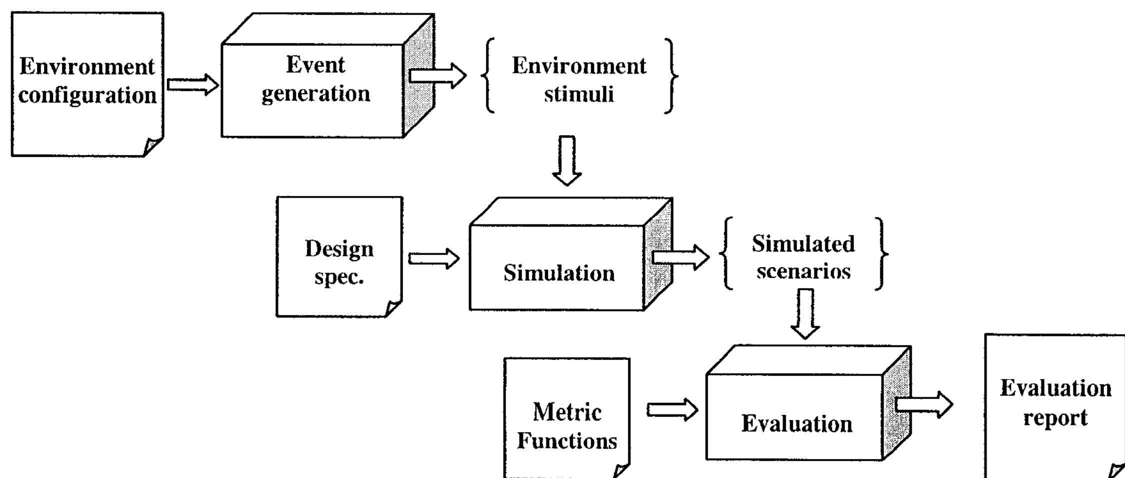

Figure 5-1 Design evaluation system

The diagram in figure 5-1 illustrates three main processes in Design Evaluation System. The objectives of these three processes in a design evaluation system are as follows:

Event Generation process can generate environment stimulus events i.e. it can simulate the characteristics of system environment to produce stimuli events according to the predefined configuration. The generated stimulus events are the input for Simulation process.

Simulation process can simulate the execution of a design i.e. it can take inputs of environment stimulus events generated in Event Generation Process, and produce system events according to the design specification. The simulated scenarios are the input of Evaluation Process.

Evaluation process can evaluate a design performance based on the simulated scenarios. By specifying a set of metric functions, it can evaluate each scenario, aggregate the evaluation result for all simulated scenarios, and finally produce the performance evaluation report to the user.

5.2 Event generation

The objective of tool support in event generation is about how to generate test data automatically, i.e. environment stimuli sequence set ES, which are used in design simulation and performance metrics evaluation. ES should be produced based on some predefined generation rules.

5.2.1 Generation Algorithm

Event generation generally follows the model we introduced in section 4.4. The current version of the implementation mainly support interaction pattern III. In this section, we are going to illustrate the algorithm what we have used in our generation tool implementation, which is to generate a canonical set of environment stimuli sequences.

In interaction pattern III, potentially there could be 1 to N objects in the environment to interact with a system concurrently, which constitute many different *event sequence pattern*. Event sequence pattern is the ordering of events without considering the timing of events. For the same sequence pattern, there are many possible *timed event sequence* TES, namely sequences with the same sequence pattern but different event time attributes. Event sequence pattern is important in pattern III for the test coverage consideration, hence we will generate and preserve all possible event sequence patterns in our generation process.

When event number increase, time configuration of individual object will increase exponentially. When number of objects increase, event sequence pattern will increase exponentially. We can see that it will take longer time to get result when object numbers increase.

Events in an environment scenario are in a timed order sending to the system. In our algorithm, we take assumption as follows:

- All environmental objects interacting with the system are in an identical behaviour pattern, namely the same sequence pattern;
- The first stimulus in an event sequence is not constrained.

5.2.1.1 Single object TES set

First we will consider generating a set of timed event sequence (TES) for a single object interacting with the system. This will be the basic information for generating sequences with multiple objects.

From environment assumptions in the requirement, we know time constraints of all events for an environment object. We specified below:

event1, event2[r_{21},r_{22}], event3[r_{31},r_{32}], event4[r_{41}, r_{42}]..., (Relative time $r_{21}<r_{22}$, $r31<r32$, $r_{41}<r_{42}$...)

(or event1(t), event2(t1, t1'),event3(t2, t2'),event4(t3, t3')..., (Absolute time t<t1<t1'< t2<t2'<t3<t3'..)

In the following formula, we get time range of each event, except first event that is unconstrained: *Range(e)=* $r_h - r_l + 1$*, where* r_h *is up-bound of an event time constraint,* r_l *is lower-bound of an event time constraint.*

Now we can calculate total number of *timed-event-sequences* (TES) for a single object. Here is the formula to do that.

Total number= Rang(Event2) x Rang(Event3) x ... xRang(Event$_v$) , v is the number of events for an environment object.

Total number= $(r_{22}-r_{21}+1) \times (r_{32}-r_{31}+1) \times (r_{42}-r_{41}+1) \ldots \times (r_{v2}-r_{v1}+1)$

As it is sufficient to take first event t_0 to be 0, we calculate absolute time value for each event by assigning first event $t_0=0$.

$t_v(\text{absolute time}) = t_{v-1}(\text{absolute time}) + t_v(\text{relative time})$

Now we generate timed event sequences set (S) for single object. For single objects, all timed event sequences are in the same event sequence pattern but with different timing features.

5.2.1.2 Multiple objects TES set

Based on single object TES set, we can generate TES set for multiple environmental objects.

As environmental objects may start interacting with the system at the different time, it will produce more event sequence patterns than the situation that all objects starting at the same time. In this situation, we need to consider non-overlapping distance, which is the time lapse between first event of an object and last event of its preceding object, or to consider shift-distance-value, which is time lapse between first event of an object and first event of its preceding object. Non-overlapping distance is equivalent to shift-distance-value that either one can be used in generation algorithm. In our implementation of the generator, we use non-overlapping distance. In our generator implementation, user can give any expected value to Non-overlapping distance, which is a selectable non-overlapping shifting parameter.

For generating TES set for multiple objects, we follow the below steps.

Step 1: Generating sequences with first event of each object starting at the same time.

We product single-object-sequences n-1 times. $S \otimes S \ldots \otimes S$. *product n-1 times, where n is the number of objects, and S is TES set for a single object.*

Step 2: Generate sequences with first events of each object starting at the different time.

We do it by shifting 2th, 3th , .. N^{th} timed event sequences. N is the number of objects. Each shift will produce a set of timed event sequences. We shift until reaching the expected non-overlapping distance.

Step 3: Filtering process

So far, we get a quite large amount of timed event sequences, in this step we will select a canonical set of sequences.

Step 3.1 Partition timed event sequences by event sequence pattern.

We need to get event sequence pattern from timed event sequences TES and preserve all event sequence pattern. It is important to preserve all different ***event sequence*** pattern (scenario). Timed event sequences are then grouped by event sequence pattern. TES in each group have the same event sequence ordering pattern.

Step 3.2 Select TES that have representative timing feature.

For each event sequence pattern, select representative timing feature and output as canonical set. We will select representative timing feature in each event sequence group to further narrow down canonical set.

Different applications have different ***criteria*** to select TES, which are domain oriented. We give an example about selection criteria in railroad crossing case, referring section 4.6.

5.2.2 Generator and filter

Based on the generation method described in previous section, we developed a software component implementing the generation process. The component includes two parts that we named them as generator and filter.

As illustrated in figure 5-2, the input of generation component is environment assumptions that include events name, events time constraints, events ordering. Based on object event configuration, generator will process the input to generate single object timed event sequences. By scheduling single object timed event sequences and merge them, generator produces timed event sequences for multiple objects. Generator will output timed event sequences to filtering process in which the event sequence (ES) is retrieved first from its relevant timed event sequences. There are certain filter rules for each event sequence. Filter is created for each ES with different rules. In each event sequence group, some of TES will be selected based on filter rules. All TES from each event sequence group constitute a canonical set.

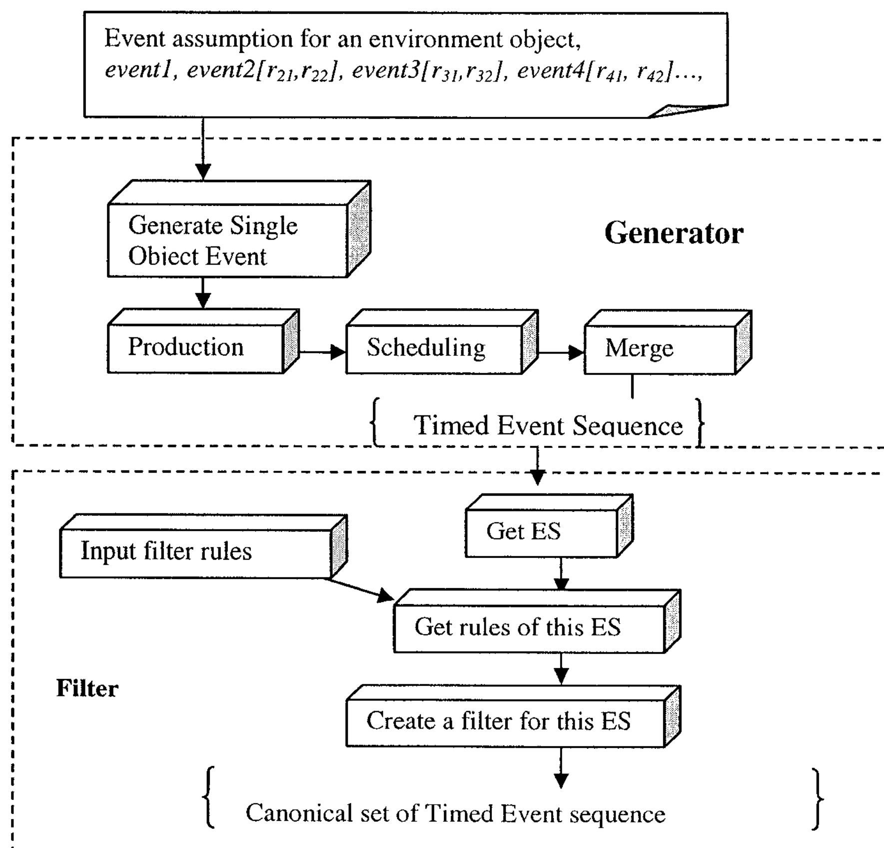

Figure 5-2 event sequence Generator and Filter

5.2.3 Component design

Based on the generation and filtering algorithms described in previous sections, we have implemented a component named "Environmental Stimuli simulator" (**ESS**). Figure 5-3 is the class diagram that shows the basic architecture of the environmental stimuli simulator.

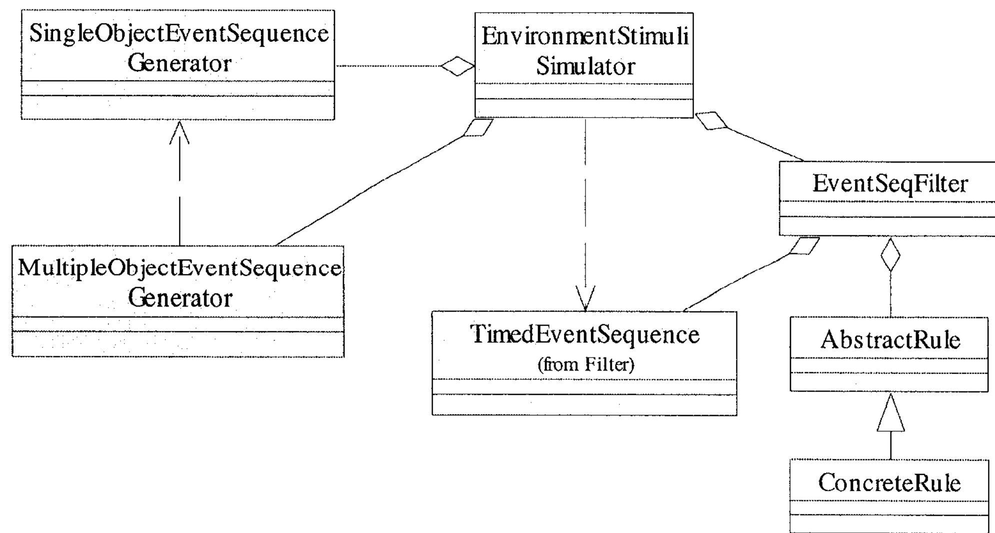

Figure 5-3 class diagram of Environmental stimuli simulator (ESS)

Component ESS contains single-object event sequence generator and multiple-object event sequence generator, which produce timed event sequences of one or multiple objects. ESS has the aggregation relationships with class *SingleObjectEventSequenceGenerator* and class *MultipleObjectEventSequenceGenerator* . Simulator contain event sequence filter as you see in the class diagram that ESS has an aggregation relationship with class *EventSeqFilter.* In this case, the timed event sequences generated in the generator will be filtered in the Event Sequence Filter. A filter need filtering rules to decide whether keep or filter out a timed event sequence. Hence,

class *EventSeqFilter* has an aggregation relationship with class Abstract rule. And filter is also a container to store timed event sequence that has satisfied the rules, so class *EventSeqFilter* has an aggregation relationship with class *TimedEventSequence*. We have applied a *policy* design pattern in our framework design. This policy design pattern will allow user to integrate different rules. Figure 5-4 is a detailed class diagram of component design for Environmental stimuli simulator.

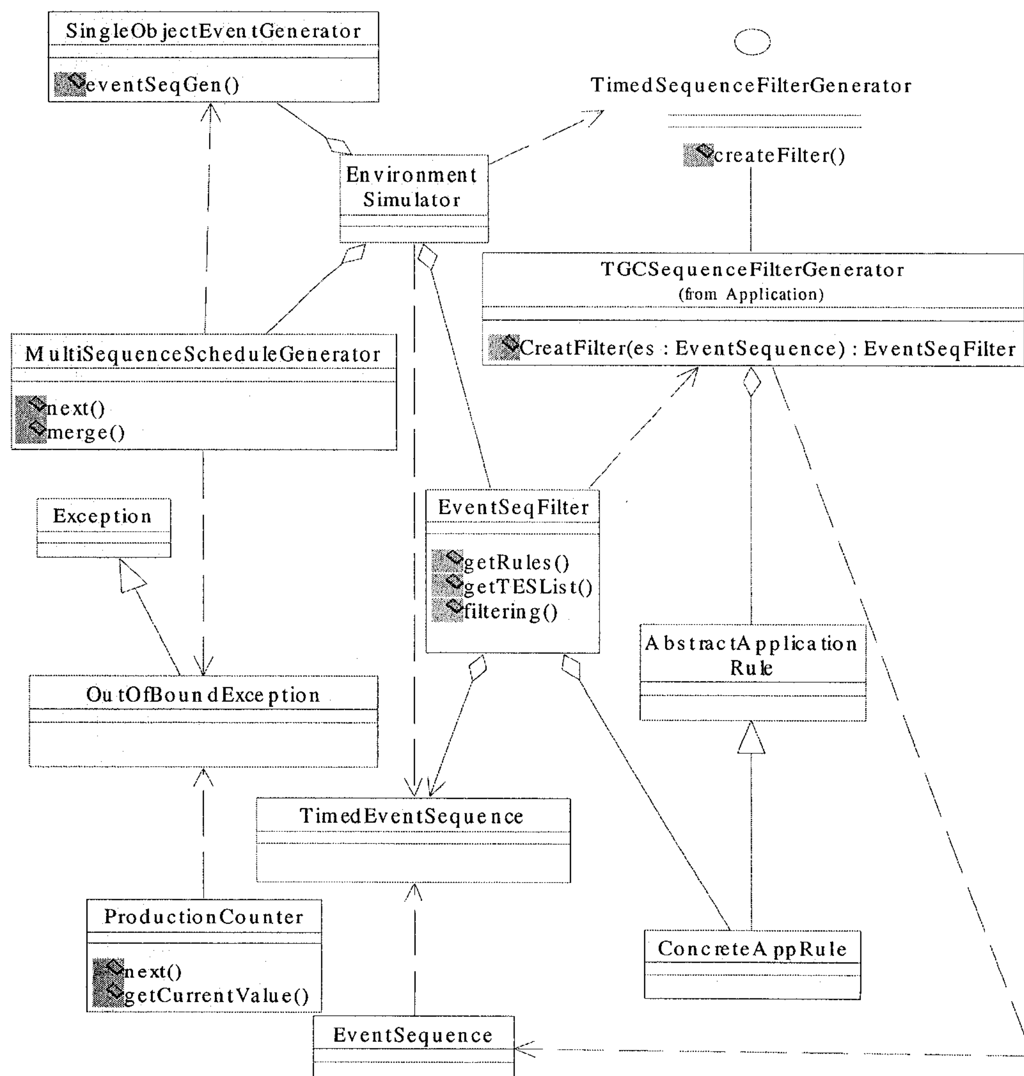

Figure 5-4 class diagram of Generator and filter

In the filter component, there are two dynamic factors to be considered in this framework design. One is the set of filter rule that are application related, different event sequence pattern need different filter rules. A sequence pattern may need some of rules (not all of them) because a pattern may not satisfy preconditions of all rules. Hence a filter must be created for each individual event sequence pattern by embedding relevant rule combinations.

Users should provide filter rules based on system timing features, this can be done by implementing interface *TimedSequenceFilterGenerator*. We give an example with Railroad crossing case, as you see in the class diagram, *TGCSequenceFilterGenerator* implement method *createFilter(EventSequence)* to create a filter for each sequence pattern by embedded relevant rules in the filter.

Figure 5-5 shows the working activities on the architecture of figure 5-4.

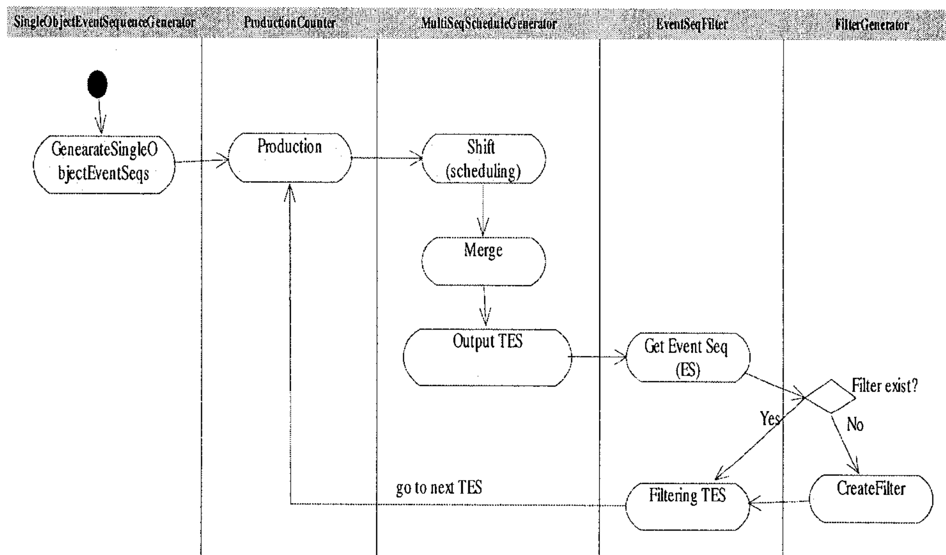

Figure 5-5 Activity diagram of Environmental stimuli simulator (ESS)

A filter may have more than one rule. Each rule in a filter has a container that is to keep a predefined number of timed event sequences (TES) that satisfy the rule, and the container has a predefined size. When a new TES arrives, filter will go through each rule with this new TES. In each rule, filter will compare new TES with those TES already in the container. If this new TES can compete an existed TES in the container in terms of rule criteria, the filter will update the container with this new TES by filtering out the old TES. Each container may have repetitive TES as illustrated in Figure 5-6. In the final filtering result, only one copy is kept for repetitive TESs.

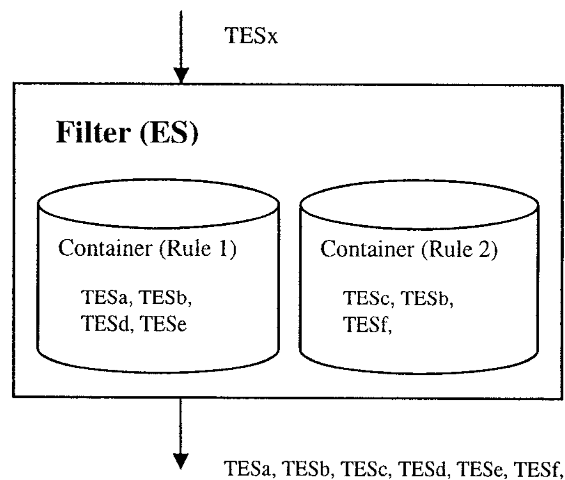

Figure 5-6 Filter and TES

We have attached Generator GUI snapshots in the appendix.

5.3 Simulation

The objective of Simulation component in our Design Evaluation System is to simulate the execution of a specific design.

In TROM framework, there already exists a simulation software, which is mainly used for debugging a design. We apply the same simulation model of that software in our design evaluation. However, since the simulation functionality in TROM simulator is highly integrated with its usage application, i.e. debugging a design, and it is hard coded, the TROM simulator can not be directly be integrated into our Design Evaluation System. Therefore, in order to utilize TROM simulator, some refactory work should be done on TROM simulator:

Componentization

In this refactory work, we need to separate the implementation of the simulation model from its usage so that the simulation functionality becomes an independent component. In addition, a set of API should be provided to make the component customizable and controllable.

Enrichment

Based on the componentized simulator, we need to let some runtime parameter open to the user so that the simulation can serve better for design evaluation.

Some performance simulation feature, such as resource consumption, should be added to the simulator.

5.4 Evaluation

Evaluation process is to compute evaluation result from simulated scenarios based on preconfigured metric functions. The objective of tool support in evaluation process is to realize automatic mechanism in the evaluation.

5.4.1 Evaluation Algorithm

As introduced in 4.3, performance functions include *single scenario metric function f* and *an aggregation function g*. For each metric, there are two types of evaluation rules that are applied to f and g during the evaluation process:

- Evaluation Rules related to f are used for evaluating a single scenario.
- Evaluation Rules related to g are used for aggregating single-scenario-evaluation results, they are aggregation rules.

The evaluation algorithm can be illustrated by the following pseudo code:

```
Given a set of simulated scenarios SS, and an aggregation
evaluation rule Rg, and a single-scenario-evaluation rules Rs:

i=0;
done := false;
While (done<>true)
      singleSResult=SingleScenarioEvaluation(SSi,Rs);
      SaveResult(singleSResult, ResultArray);
            if( i++ = = size(SS) )  done = true;
End While
EvaluationResult=Aggregate(ResultArray,Rg);

Note:
-SingleScenarioEvaluation(SSi, Rs) is a function that computes
evaluation rule Rs with each SSi  to get single-scenario-
evaluation result.
-SaveResult(singleSResult,ResultArray) is a function that can
save each single-scenario-evaluation result into "ResultArray"
for later aggregate each single scenario result.
-Aggregate(ResultArray, Rg) is a function that can computes
aggregation evaluation rules with a set of single-scenario-
evaluation result to get final performance result.
```

For instance: Evaluating average response time. Evaluation rule of single scenario Rs is about how to get response time from each simulated scenario in simulated scenarios set (SS has N scenarios), which is to get time distance of two specified events of a scenario. After retrieving response time of each scenario in the scenario set ($RespT_1$, $RespT_2$, $RespT_3$...$RespT_N$), Aggregation Evaluation rule Rg is about how to calculate average response time:

$$AverageRespT = (RespT_1+RespT_2+RespT_3+...+RespT_N) / N$$

5.4.2 Expression library

Evaluation rules related to g and f are represented as *well-formed formula (wff).* Any *wff* has a natural syntax expression tree that clearly displays the hierarchy of rule operations of this *wff*. All of these operations in a rule can be represented as nodes in the expression tree.

We have designed a program model based on evaluation rules structure explained above. The model will allow users to easily construct any evaluation rules and compute the result. We illustrate the program model in the following class diagram, Figure 5-11.

Evaluation rule is the dynamic part that varies in different system evaluation. Dynamic rule construction and automatic rule value computation are the key requirements in the design. We apply the *composite* design pattern with some modification (Figure 5-7). An operand can be an operation or it can be a leaf node, the operand of an operation.

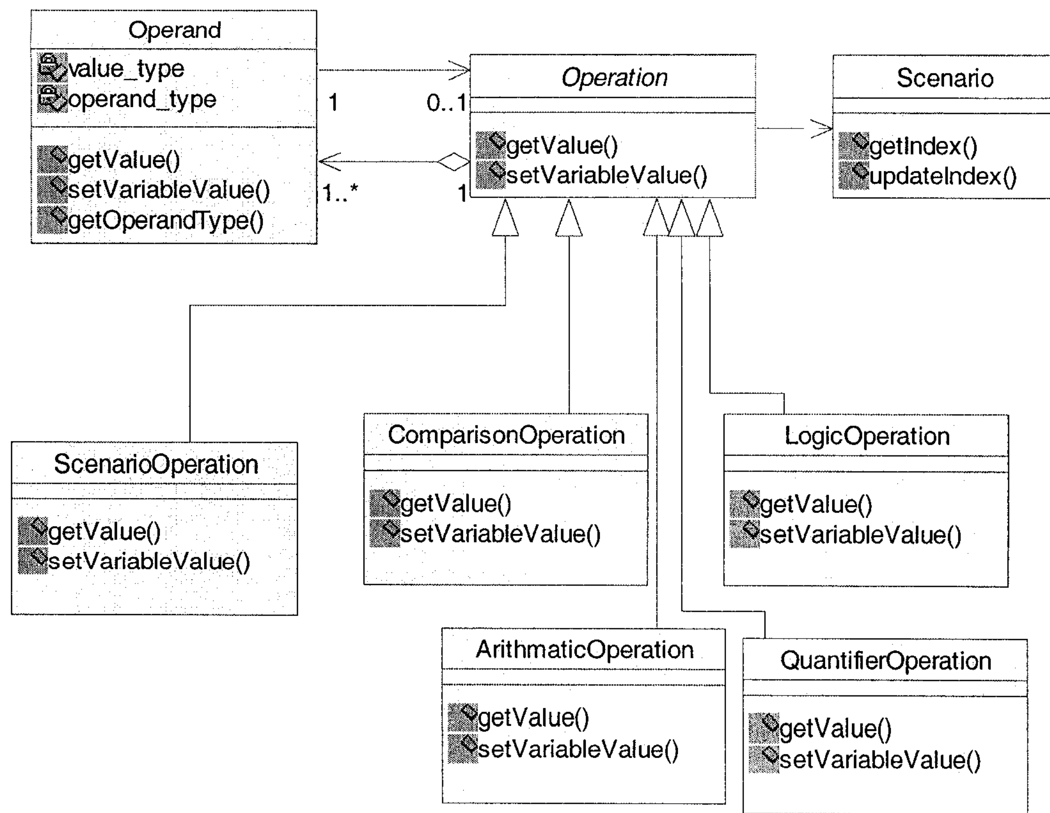

Figure 5-7 Program model design

Abstract class operation has two important methods: *getValue()* can compute a value for a concrete operation according to operation type; *setVariableValue()* allow user to set value to a variable in the operation.

Each operation has a few operands. From the class diagram we can see class operation and class operand has an aggregation relationship with cardinalities one to many. Operations that are unary or binary one have one or two operands respectively. For example logical NOT operation is a unary operation that has only one operand. Logical AND / OR operation are binary operations that have two operands.

An operand has three possible types that it can be a ***constant***, an ***operation***, or a ***variable***. Hence, class *operand* has an attribute *operand_type* to indicate these three

possibilities. From the class diagram we can see class operand class operation has an association relationship with cardinality one to one (when the operand is an operation type) or one to none (when the operand is not operation type).

The operands of each operation can be different data type, for example Integer, Boolean, or String etc. The attribute *value_type* in class *operand* indicates the data type of an operand. Class *operand* has two methods *getValue(), setVariableValue()* that have the same signature as class *operation*. These two methods are involved in recursive functional call by traversing the tree to calculate the operation value or to set a value to a variable in the tree.

As each variable has a different name, a variable can be found by its name, and the value is passed through recursive function call.

To help users easily build rules within this model, we have implemented an expression library. This library includes many basic operations for users to invoke. There are many different types of fundamental operations, so we can group operations in different way. For example, the operations can be grouped by the number of operands. In this case, we can have unary operations set and binary operations set etc. The operations can also be grouped by their functions. In this case, we can have logic operations, arithmetic operations etc. In our implementation of expression library we chose the latter solution. Our library include operations as follows:

- ***quantifiers*** operations (existential, universal),
- ***logic*** operations (disjunction, conjunction, negation, etc...),
- ***arithmetic*** operations (addition, subtraction, etc…),

- ***scenario*** operations. There are many scenario operations in the library that can retrieve different information from a simulated scenario.

We illustrate how to use this library with case study explained in section 4.5. To realize automation in the evaluation process, rules elicited in this example (refer 4.5) have to be computable. We show some functions in the following that have been implemented in our tool:

SizeOf(S_i) : Calculate the number of events in the scenario under evaluated.

getLastSysEvent(S_i) : Get last system event.

getTLastSysEvent(S_i) : which time slot that last system event is located.

getLastEvent(S_i) : Get last event in this scenario.

getEnvNum(S_i, *e*) : Get number of environmental event e in this scenario.

Logic operations such as OR, AND, etc.

Comparison operations such as EQUAL(a, b) , etc.

Note: S is the scenario under evaluated; S_i is a scenario-segment of S; S_{i+1} is the next scenario-segment of S_i

When working with our evaluation tool, legality rules have to be built using the functions that our tool has provided. In the following, we show the legality rules for this case that are represented using the functions in the library:

<Rule 1>

$\text{SizeOf}(S_i)>0 \wedge \text{NOT}(\text{ IsLastSysEvent ("closed" , Si) })$

$\rightarrow \text{NOT } (\exists e \bullet e \in \text{Sub}(S_{i+1}, \text{Si}) \mid e= \text{"}In\text{"})$

< Rule 2>

SizeOf(S_i)>0 ∧

(IsLastEvent(*Exit"*, Si) ∧ NOT

(getEnvNum(*"Near"*, 0, getLastEnvEvent(*"Exit"*)) =

getEnvNum(*"In"*, 0, getLastEnvEvent(*"Exit"*))

∧ getEnvNum(*"Near"*, 0, getLastEnvEvent(*"Exit"*)) =

getEnvNum(*"Exit"*, 0, getLastEnvEvent(*"Exit"*)))

∨ IsLastEvent(Env, "In",Si))

→ NOT (∃e•e∈ Sub(S_{i+1}, Si) | e= *"opened"*)

<Rule 3>

SizeOf(S_i)>0 ∧ IsLastEvent(*"Exit"*, Si) ∧ (

(getEnvNum(*"Near"*, 0, getLastEnvEvent(*"Exit"*)) =

getEnvNum(*"In"*, 0, getLastEnvEvent(*"Exit"*))

∧ getEnvNum(*"Near"*, 0, getLastEnvEvent(*"Exit"*)) =

getEnvNum(*"Exit"*, 0, getLastEnvEvent(*"Exit"*))))

→ ∃e•e∈ Sub(S_{i+1}, Si) | ((e=" *opened* " ∧ ∨ e= " *Near* ") ∧

e.time<= getLastEnvEvent(*"Exit"*)+3)

These are three rules that can check each scenario segment in a simulated scenario. All scenario segments must satisfy these rules for the simulated scenario to become a legal scenario.

5.4.3 Scenario Legality evaluation

In this section, we show an implementation of scenario legality evaluation by applying our program model introduced in previous section. Scenario legality evaluation is about correctness metrics that has been explained in 4.5.1.

Correctness metrics is the most sophisticated one to evaluate among other metrics, we give detailed correctness evaluation algorithm in the following. Usually system requirements are specified as a set of legality rules Rs instead of one single rule. Therefore, the evaluation involves checking that an execution scenario ES is legal for all scenario legality rule Ri in the rule set Rs.

With the understanding of these, the algorithm of correctness evaluation can be illustrated by the following pseudo code:

```
Given a simulated scenario S and a set of rules {Rj};
Si := ϕ ;
Sinext := next(S, Si);
done := false;
While (done<>true)
      For all Rj in set {Rj}
            if Rj(Si, Sinext) <> true
                  report failure(Rj,Si,Sinext);
                        return;
            end if
      end for
      if Sinext = ϕ and Si = S
      report "S is legal"
            done := true
      else
            Si := Si ∪ Sinext;
            Sinext := next(S, Si),
      End if
end while
```

Figure 5-8 Scenario legality Evaluation algorithm

next(S, Si) is a scenario operation function that returns Si's next sub-scenario in S.

Each simulated scenario will go through evaluation process to be checked by evaluation rules. As for the case of correctness metrics, each scenario has to be checked against legality rules. In Railroad Crossing case study showed in section 4.6, three legality rules have been elicited from functional requirement hence each scenario has to pass these three rules to become legal scenario (section 4.5 (1)).

Each legality rule can be represented in a tree structure. By using basic operations provided in the library, we can develop rules with the program model. Refer 5.4.2 to see three expression rules for Railroad Crossing case study.

Below we show our testing result about evaluating simulated scenarios (scenario A, B, C, D) with these three legality rules (Rule 1, 2,3). In the following figures we illustrate simulated scenarios and its evaluation result interpretation. For scenario A, we attached the copy of evaluation result in Figure 5-9. Note: for example, *train0.near(0)* means *train0* sends "*near*" event at time 0.

Scenario A:	*Result Interpretation:*
train0.near(0), *train1.near(1),* *sys.closed(2),* *train1.in(4),* *train0.in(5),* *train1.exit(5),* *train0.exit(6),* *sys.opened(8),* *train2.near(9),* *sys.closed(11),* *train2.in(12),* *train2.exit(13),* *sys.opened(16).*	Scenario A is a legal scenario. Scenario A satisfies legality rule1, rule 2, and rule 3.

```
Scenario A
Scenario segment index: -1,0
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Scenario segment index: 0,1
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Scenario segment index: 1,2
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Scenario segment index: 2,3
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 3,5
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 5,6
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 6,7
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Scenario segment index: 7,8
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 8,9
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 9,10
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 10,11
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Scenario segment index: 11,12
Logic IMPLICATION result:P(false),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Scenario segment index: 12,13
Logic IMPLICATION result:P(true),Q(true)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Logic IMPLICATION result:P(false),Q(false)->Impl(true)
Evaluation result= true
```

Figure 5-9 The Evaluation Result of Scenario A

Scenario B:	*Result interpretation*:
train0.near(0), train1.near(1), sys.closed(2), train1.in(4), train0.in(5), train1.exit(5), train0.exit(6), sys.opened(8), train2.near(9), sys.closed(11), train2.in(12), train2.exit(13),	Scenario B is not a legal scenario. Scenario B violates rule 3. Rule 3 says "The gate must open after all train exit"

Scenario C:	*Result interpretation*:
train0.near(0), train1.near(1), train1.in(4), train0.in(5), train1.exit(5), train0.exit(6), sys.opened(8), train2.near(9), sys.closed(11), train2.in(12), train2.exit(13), sys.opened(16).	Scenario C is not a legal scenario. Scenario C violates rule 1 that rule 1 says "The gate must close when there is a train approaching the crossing"

Scenario D:	*Result interpretation:*
train0.near(0), train1.near(1), sys.closed(2), train1.in(4), train0.in(5), train1.exit(5), sys.opened(8), train0.exit(9), train2.near(9), sys.closed(11), train2.in(12), train2.exit(13), sys.opened(16).	Scenario D is not a legal scenario. Scenario D violates rule 2 that rule 2 says "The gate can't open when there is at least a train remaining in the crossing"

Chapter 6 Conclusion and future work

6.1 Thesis Review

In this work, we have explored a requirement engineering approach for modeling real time reactive systems and evaluating various design solutions.

We developed an object-event-scenario modeling method to describe system context and behavioural characteristics and various usages of a system. This method is intuitive, user-centered, and easy to apply.

Based on our object-event-scenario model, we further developed a scenario-based design performance evaluation method for evaluating various design solutions. This method applies a black-box strategy, by which we can simulate a design execution, test a simulated system, and finally evaluate a design. This approach provides a theoretical foundation for automating design evaluation process. In addition, we provided a sound guidance on how to design performance metrics, and how to design a canonical stimuli scenario set as test data in a systematic design performance evaluation. We gave patterns on some common metrics such as correctness, response time, throughput etc.

Besides theoretical proposal, we also presented supporting tools for automatic design evaluation process, such as Evaluator, Event generator.

6.2 Future work

Our work provides a strategy for requirement engineering in developing real time reactive systems. It also provides a direction for some further research and development on this direction as follows.

Some possible future works are:

- Variant interaction patterns require further studied and their relevant canonical set selection methods can be studied and developed.

- Applied application domains should be further studied and our methodology should be adapted so that it can be more effective.

- Simulation strategy and tool support can be enriched with additional features to support more effective performance evaluation.

- Design Evaluation System should be further developed to be more integral. Tool support for modeling phase should be designed and developed.

Bibliography

[1] Axel van Lamsweerde, "Requirements Engineering in the Year00: A Research Perspective", Proceedings of the 22nd International Conference on Software Engineering, June 4-11, 2000, Limerick Ireland. ACM, 2000

[2] Pamela Zave, "Classification of Research Efforts in requirements engineering", second IEEE International Symposium on Requirements Engineering, March 27 - 29, 1995, York, England. IEEE Computer Society 1995.

[3] Object Management Group, "UML profile for Schedulability, Performance, and Time Specification", Version 1.0, September 2003

[4] Bjorn Regnell, Requirements Engineering with Use Cases—a Basis for Software Development, Ph.D. thesis, ISRN LUTEDX/TETS--1040--SE+225P, Department of Communication Systems, Lund Institute of Technology, Lund University, Lund, Sweden,1999

[5] Simonetta Balsamo, Moreno Marzolla, "A Simulation-Based Approach to Software Performance Modeling", Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003. ACM, 2003

[6] Bruce Powel Douglass, " Doing Hard Time—Developing Real-time systems with UML, objects, frameworks, and patterns" Addison-Wesley, 1995, ISBN: 0201498375.

[7] Alan C. Shaw, "Real-Time Systems and Software", John Wiley & Sons, 2001, ISBN 0-471-35490-2.

[8] Weidenhaupt, K. Pohl, K., Jarke, M., Haumer, P., "Scenario Usage in System Development : A Report on Current Practice", 3rd International Conference on Requirements Engineering (ICRE '98), Putting Requirements Engineering to Practice, April 6-10, 1998, Colorado Springs, CO, USA, Proceedings. IEEE Computer Society 1998, ISBN 0-8186-8356-2 .

[9] Craig Larman, "Applying UML and Patterns, second edition", Prentice Hall, 2002, ISBN 0-13-092569-1

[10] Ian Sommerville.'Software Engineering' fifth edition, ISBN 0-201-42765-6,1996

[11] Hermann Kopetz, 'Software Engineering for Real-Time: A Roadmap', ICSE 2000, 22nd International Conference on on Software Engineering, Future of Software Engineering Track, June 4-11, 2000, Limerick Ireland. ACM, 2000

[12] Ivar Jacoba, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard, "Object-Oriented Software Engineering--Use Case Driven Approach", Addison-Wesley Professional; 1st edition,1992, ISBN 0201544350

[13] V.S.Alagar, R. Achutham, D.Muthiayen "TROMLAB: An Object-oriented framework for real-time reactive system development". Technical report, Department of Computer Science, Concordia University, Montreal, Canada, 2001

[14] O. Popistas, "Rose-GRC translator: mapping UML visual models onto formal specifications", M.S. Thesis, Department of Computer Science, Concordia University, Montreal, Canada,1999.

[15] H. Tao, "Static Analyzer: A Design Tool for TROM". M.S. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1996.

[16] Ghayath Haidar, "Reasoning system for real-time reactive systems", Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 2000.

[17] V. Srinivasan, "Graphical User Interface for TROMLAB Environment". M.S Thesis Department of Computer Science, Concordia University, Montreal, Canada, 1999

[18] M. Haydar, "Parameterized events for designing Real-time reactive systems" M.S. Thesis , Department of Computer Science, Concordia University, Montreal, Canada , 2001

[19] Olga Ormandjieva, "Deriving New Measurements For Real-Time Reactive Systems", PhD Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 2002.

[20] Mao Zheng, "Automated test case generation from formal specifications of real-time reactive systems", PhD Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2002.

[21] R. Achuthan. A Formal Model for Object-Oriented Development of Real-Time Reactive Systems. Ph.D. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1995.

[22] L. Zhang. Implementing Real-Time Reactive System from Object-Oriented Design Specifications Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 2000.

[23] 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, October 2000.

[24] Fenton, N and Pleeger, S. Software metrics: A Rigorous & Practical Approach. PWS Publishing, 2nd edition, revised printing, 1998, ISBN 0-534-95425-1

[25] Simonetta Balsamo, et al "Model-Based Performance Prediction in Software Development: A Survey", IEEE Transactions On Software Engineering, Vol 30, No.5, May 2004.

[26] Daniel Galin, Software Quality Assurance—from theory to implementation, Pearson Education Limited 2004. ISBN 0201709457

[27] Rhapsody, **www.ilogix.com**. (Rhapsody Model driven development environment)

[28] **www.telelogic.com** (TAU suite)

[29] Axel van lamsweerde, "Goal-Oriented Requirement engineering: A Guided Tour", 5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada. IEEE Computer Society 2001, ISBN 0-7695-1125-2

[30] Bashar Nuseibeh, Steve Easterbrook "Requirement Engineering: A Roadmap" Proceedings of International Conference on Software Engineering (ICSE-2000), 4-11 June 2000, Limerick, Ireland

[31] V. S. Alagar, O. Ormandjieva, S.H.Liu "Scenario-Based Performance Modelling and Validation in Real-Time Reactive Systems", Proceedings of Software Measurement European Forum, Italy, 28-30 January 2004

[32] V. S. Alagar , O.Ormandjieva, S.H.Liu. Jian Shen "Performance Assessment in Real-time Reactive System", 7th IASTED International Conference on Software Engineering and Applications, Marina del Rey, CA, USA November 3-5 2003

Appendix

A. Generator snapshots

Below are some snapshots from generator and filter implementation. Figure A-1 shows input event name and time constraints in object configuration window. Figure A-2 is the generation result for the single object. Figure A-3 is the generation result of two objects. Figure A-4 is the filtering result from the generated data set as seen in Figure A-3.

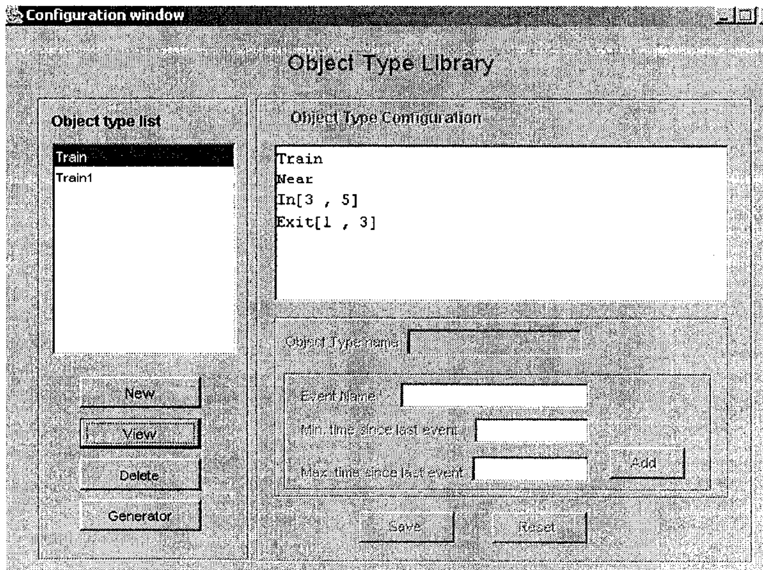

Figure A-1 object configuration screen

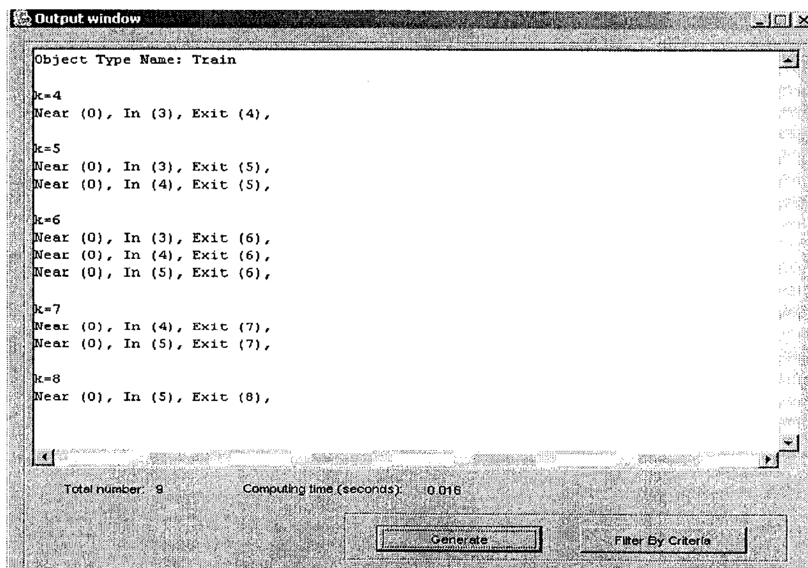

Figure A-2 single object generation

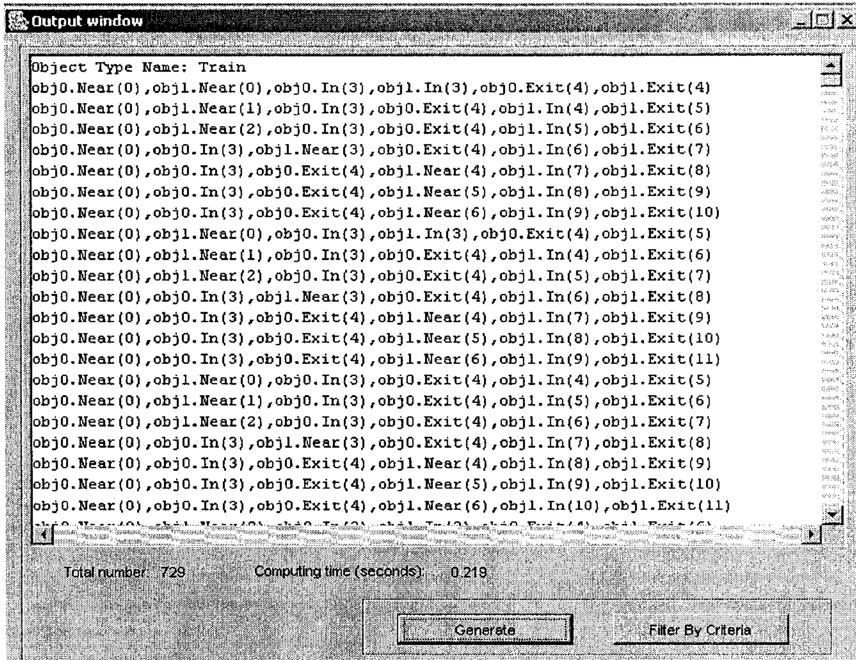

Figure A-3 multiple objects generation

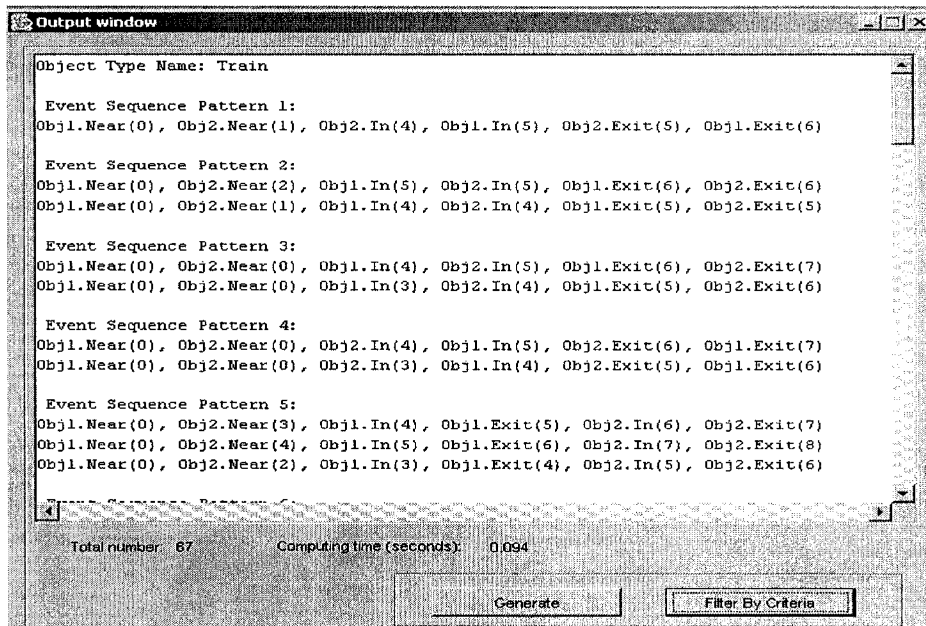

Figure 5-11 Filtering result

B. Expression library (part)

```
public abstract class Operation
{
        public static final int OBJECT = 900;
        public static final int BOOLEAN = 901;
        public static final int INTEGER = 902;
        public static final int DOUBLE = 903;
        public static final int FLOAT = 904;
        public static final int STRING = 905;
        public static final int SCENARIO=909;

        protected int value_type; //vaule type of an operation result such as Boolean, Integer etc.
        protected int operation_type; // type of operations such as Logic AND, OR etc.
        protected Operand[] operands;

        public void setValueType(int v_type)
        { value_type=v_type;
        }
        public int getValueType(){ return value_type; }

        /**
         * Set a value to a variable in an operation
         * @param variableName String
         * @param data Object
         */
        public void setVariableValue(String variableName,Object data)
        {
         for(int i=0; i<operands.length;i++)
           operands[i].setVariableValue(variableName, data);
        }
        /**
         * Calculate an operation result
         * @return Object
         */
        public abstract Object getValue() throws Exception;
}

public class Operand
{       public static final int CONSTANT=1;
        public static final int OPERATION=2;
        public static final int VARIABLE=3;

        int operand_type;
        int value_type;

        Object data;
        String variable_name;


        public Operand (int operand_type, Object d, int dtype) throws Exception
        { value_type=dtype;
         switch (operand_type)
```

```
{ case CONSTANT:
    this.operand_type=operand_type;
    this.data = d;
    break;
  case OPERATION:
    if(d instanceof Operation)
    { if( ((Operation)d).getValueType()!=dtype )
        throw new Exception("operation value type doesn't match operands!");
      this.operand_type=operand_type;
      this.data = d;
    }
    else throw new Exception("Operand data doesn't match the specified type.");
    break;
  case VARIABLE:
    throw new Exception("invalid constructor call.");
  default:
    throw new Exception("invalid operand type.");
 }

}
public Operand(String v_name,int dtype) throws Exception
{
 if(v_name==null||v_name.length()==0) throw new Exception("invalid variable name.");
 this.variable_name=v_name;
 operand_type = VARIABLE;
 value_type=dtype;
}
/**
 * This function is to assign a value to a variable. The Variable is recognized by its name.
 * @param v_name String
 * @param value Object
 */
public void setVariableValue(String v_name,Object value)
{ switch(operand_type)
 {
  case VARIABLE:
    if (v_name != null && this.variable_name.equals(v_name))
      this.data = value;
    break;
  case OPERATION:
    ((Operation)this.data).setVariableValue(v_name,value);
    break;
 }
}

public int getDataType()
{ return value_type;
}
/**
 * return value of the operand
 * @throws Exception
 * @return Object
 */
public Object getValue() throws Exception
{
 if(operand_type==OPERATION)
```

```
                return ((Operation)data).getValue();
              else
                return data;
            }
            public Object getData()
            { return data;
            }
            public String getVariableName()
            {return variable_name;

            }
}

public class LogicOperation extends Operation
{
  public final static int AND = 101;
  public final static int OR = 102;
  public final static int NOT = 103;
  public final static int IMPLICATION=104;

  public LogicOperation(int op_type,Operand[] opds) throws Exception
  { value_type=Operation.BOOLEAN;
    if (opds ==null) throw new Exception("Oprand argument can not be empty.");
    switch(op_type)
    { case AND:
      case OR:
      case IMPLICATION:
        if(opds.length==2 && opds[0]!=null && opds[1]!=null)
        { if (opds[0].getDataType()==Operation.BOOLEAN&&
opds[1].getDataType()==Operation.BOOLEAN)
          { operation_type = op_type;
            operands = opds;
          }
          else
          { throw new Exception("Logical operand is not boolean type.");
          }
        }
        else throw new Exception("Invalid number of arguments");
        break;

      case NOT:
        if(opds.length==1 && opds[0]!=null)
        {
          if (opds[0].getDataType()==Operation.BOOLEAN)
          {
            operation_type = op_type;
            operands = opds;
          }else
          {
            throw new Exception("Logical operand is not boolean type.");
          }
        }
        else throw new Exception("Invalid number of arguments");
    }
  }
  public Object getValue()throws Exception
```

```
{ boolean result;
 switch(operation_type)
 {
  case AND:
   Boolean and_left = (Boolean)operands[0].getValue();
   if(and_left.booleanValue()==false) result=false;
   else
   { Boolean and_right = (Boolean)operands[1].getValue();
    result = and_left.booleanValue() && and_right.booleanValue();
   }
   System.out.println("Logic AND result:"+ result);
   return new Boolean(result);

  case OR:
   Boolean or_left = (Boolean)operands[0].getValue();
   if(or_left.booleanValue()==true) result=true;
   else
   { Boolean or_right = (Boolean)operands[1].getValue();
    result = or_left.booleanValue() || or_right.booleanValue();
   }
   System.out.println("Logic OR result:"+ result);
   return new Boolean(result);

  case NOT:
   Boolean not_opd = (Boolean)operands[0].getValue();
   result=!not_opd.booleanValue();
   System.out.println("Logic NOT result:"+ result);
   return new Boolean(result);
  case IMPLICATION:
   Boolean imp_left = (Boolean)operands[0].getValue();
   Boolean imp_right = (Boolean)operands[1].getValue();
   if(imp_left.booleanValue()==true && imp_right.booleanValue()==true)
       { System.out.println("Logic IMPLICATION result:P(true),Q(true)->Impl(true)");
    System.out.println();
    return new Boolean(true);
   }
       if(imp_left.booleanValue()==true && imp_right.booleanValue()==false)
   { System.out.println("Logic IMPLICATION result:P(true),Q(false)->Impl(false)");
    System.out.println();
    return new Boolean(false);
   }
   if(imp_left.booleanValue()==false && imp_right.booleanValue()==true)
   { System.out.println("Logic IMPLICATION result:P(false),Q(true)->Impl(true)");
    System.out.println();
    return new Boolean(true);
   }
       if(imp_left.booleanValue()==false && imp_right.booleanValue()==false)
   { System.out.println("Logic IMPLICATION result:P(false),Q(false)->Impl(true)");
    System.out.println();
    return new Boolean(true);
   }
 }
 return null;
}
}
```