

# **Real-Time Reactive Systems Measurement Tool**

## **TROM-QM: Design and Implementation**

Manjiang Zhuo

**A Major Report**

**In**

**Department**

**Of**

**Computer Science**

**Presented in Partial Fulfillment of the Requirements**

**For the Degree of Master of Computer Science**

**Concordia University**

**Montreal, Quebec, Canada**

**November 7, 2003**

**© Manjiang Zhuo, 2003**



National Library  
of Canada

Bibliothèque nationale  
du Canada

Acquisitions and  
Bibliographic Services

Acquisitions et  
services bibliographiques

395 Wellington Street  
Ottawa ON K1A 0N4  
Canada

395, rue Wellington  
Ottawa ON K1A 0N4  
Canada

*Your file    Votre référence*

*ISBN: 0-612-91164-0*

*Our file    Notre référence*

*ISBN: 0-612-91164-0*

The author has granted a non-exclusive licence allowing the National Library of 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 nationale du 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 autrement reproduits sans son autorisation.

---

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

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

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

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

**Canada**



# **Abstract**

## **Real-Time Reactive Systems Measurement Tool**

### **TROM-QM: Design and Implementation**

**Manjiang Zhuo**

Software engineering is a discipline whose aim is the production of quality software, delivered on time, within budget, and satisfying user expectations. The control on the software development process and products would allow increasing the quality of the final product. Software measurement is the required mechanism to provide feedback and assist the quality control on the software development, testing and maintenance. As measurement procedure is both time and resource consuming procedure, a tool for automatic quality measurement and gathering of measurement data is being designed and implemented. This major report describes the design and implementation of the TROM-QM, software quality measurement system for real-time reactive systems development in the TROMLAB environment. Java has been chosen as the implementation language. TROM-SCMS, the complexity measurement component of the TROM-QM, has been designed, implemented and integrated into the TROM-QM. The main feature of the TROM-QM is its flexibility allowing the integration of more measurements when developed. The integration of additional measurements procedure is outlined. The user manual for the system is included.

## **Acknowledgement**

I would like to thank my supervisor, Dr. Olga Ormandjieva. She gave me helpful guidance and advices all along the way.

I would also like to thank Dr. V.S Alagar as examiner for this report.

Also, I would like to thank my husband Jinyue and my son Zhuo for their patience, understanding and love.

## **Table of Contents**

LIST OF FIGURES .....	VII
LIST OF TABLES .....	IX
CHAPTER 1 INTRODUCTION .....	1
1.1 THE IMPORTANCE OF SOFTWARE QUALITY MEASUREMENT.....	1
1.2 PURPOSE AND PROBLEM STATEMENT .....	2
1.3 REPORT OUTLINE .....	2
CHAPTER 2 BACKGROUND .....	4
2.1 REAL-TIME REACTIVE SYSTEMS.....	4
2.2 TROM FORMALISM .....	5
2.3 TROMLAB .....	10
CHAPTER 3 COMPLEXITY MEASUREMENT.....	12
3.1 COMPLEXITY MODULE IN TROMLAB .....	12
3.2 ARCHITECTURAL COMPLEXITY MEASUREMENT .....	12
3.3 DESCRIPTION OF THE MODULE FUNCTIONALITY.....	21
3.4 DESCRIPTION OF THE COMPONENTS IN THE TROM-SCMS .....	21
3.5 ARCHITECTURE DIAGRAM.....	22

3.6 DATA FLOW DIAGRAM .....	22
3.7 CLASS DIAGRAM .....	23
3.8 SEQUENCE DIAGRAM.....	24
3.9 ALGORITHMS USED IN TROM-SCMS MODULE.....	26
CHAPTER 4 TROM-QM MODULE.....	36
4.1 INTRODUCTION TROM-QM.....	36
4.2 TROM-QM USER INTERFACE.....	36
4.3 DESCRIPTION OF THE COMPONENTS IN THE TROM-QM.....	41
CHAPTER 5 INTEGRATION GUIDE AND USER GUIDE.....	49
5.1 INTEGRATION GUIDE .....	49
5.2 USER GUIDE .....	50
CHAPTER 6 CONCLUSIONS & FUTURE WORK.....	52
REFERENCES.....	53

## List of Figures

Figure 1 LSL Trait for Set.....	7
Figure 2 Anatomy of a Reactive Object.....	8
Figure 3 Template for GRC class.....	10
Figure 4 Templates for System Configuration Specification.....	10
Figure 5 Collaboration Diagrams for a Train-Gate-Controller2 Subsystem.....	17
Figure 6 Train-Gate-Controller2 Subsystem Configuration Specification .....	18
Figure 7 Train-Gate-Controller2 Subsystem: graph for communication links .....	19
Figure 8 Object-Predicate Table Abstractions for the Connected Components for figure 7 .....	19
Figure 9 the Architecture Diagram of AC modules .....	22
Figure 10 the Data Flow Diagram of AC Module .....	23
Figure 11 the Class Diagram of the AC Module.....	24
Figure 12 the Sequence Diagram for AC Module .....	25
Figure 13 User Interface - Measurement Tools User.....	37
Figure 14 User Interface - View of Measurement Results -1 .....	38
Figure 15 User Interface - View of Measurement Result – 2 .....	39
Figure 16 User Interface - View of Measurement Result -3 .....	40
Figure 17 Save as the results in table view to Excel file.....	41
Figure 18 Select SCS file event handling function. ....	42
Figure 19 OK event-handling functions.....	44



Figure 20 Calculating the Architectural Complexity function.....	45
Figure 21 Designs of View Measurement Results.....	48
Figure 22 Result Table Design View .....	50

## **List of Tables**

Table 1 Object Predicate Table .....	27
Table 2 Port-Port Communication Table .....	27

# **Chapter 1 Introduction**

## **1.1 The importance of software quality measurement**

In software industrial practice, the high cost of development process of large-scale software has put emphasis on the need to increase the quality of software development. The quality control in earlier phase is essential for the economics of the software development. Hence, it is necessary to have precise, predictable, and repeatable control over the software development process and product. In the context of real-time systems, which are mostly safety-critical, the quality control is a must. Examples of real-time software systems include alarm systems, air traffic control systems, nuclear reactor control systems and telecommunication systems. Any error occurring in these software systems will course a serious consequence, such as human life loss, or living environment damage. The real-time software system usually is inherently complex. This Major Report describes the design and implementation of a measurement system for managing complexity in real-time reactive systems. The proposed approach is applicable to real-time reactive systems modeled as timed labeled transition systems, and developed according to the process model shown in Figure 1. The Major Report discusses the complexity measurement for developing real-time reactive systems, and describes the design and the implementation of a tool for automatic measurement and data gathering at the design specification level.

## **1.2 Purpose and Problem Statement**

The main goal of this major report is to automate the measurement methods for complexity and reliability, and integrate them into TROMLAB [AAM98], a rigorous framework for the development of real-time reactive systems based on the TROM formalism [Ach95]. The measurement methods are described in [Orm02].

The main contributions of the report are:

- Review and adjustment of the complexity measurement method proposed by Dr. Olga Ormandjieva [Orm02];
- Implementation of complexity measurement algorithms and the maintainability profile (TROM-SCMS);
- Testing the implementation on the gate-train-controller case study and compare the testing results to the theoretical ones;
- Design and implementation of a flexible measurement system TROM-QM;
- Integration of the complexity and reliability under a common user interface to capture the input to the module and display the measurements results.

## **1.3 Report Outline**

The major report is organized as follows: Chapter 2 reviews briefly the TROM formalism, and gives an overview on the TROMLAB environment. Chapter 3 explains the complexity measurement, and illustrates it on a case study. The designs of the complexity measurement component TROM-SCMS, and the key algorithms used in the architecture complexity quantification are also included in Chapter 3. Chapter 4 introduces the TROM-QM system, the user interface and the integration of the measurements modules for complexity and reliability.

Chapter 5 describes how to integrate the other modules to the system. The conclusions and the future work are outlined in Chapter 6.

## Chapter 2 Background

The complexity measurement theory is derived from TROM formalism in the context of TROMLAB. The goal of this chapter is to give an overview of the TROM formalism and the TROMLAB environment for development of real-time reactive systems.

### 2.1 Real-time Reactive Systems

Real-time reactive systems are largely event-driven; interact intensively and continuously with the environment through stimulus-response behaviour and are regulated by strict timing constraints [Orm02]. Their behaviours are controlled by strict timing constraints. The term *reactive* introduced by Harel and Pnueli [HP85] distinguishes the systems that continuously interact with their environment from the interactive and transformational systems. Real-time reactive systems possess two important properties: ***Stimulus synchronization*** and ***Response synchronization***. These two properties distinguish them from other real-time systems. Stimulus synchronization means that the system is always reacting to a stimulus from the environment. Response synchronization means the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment, so that the environment is still receptive to the response.

The main issue in the development of safety-critical systems is to produce a reliable design. The real-time system design is a conceptual solution to the domain problem and is the basis for an implementation of the solution. To

achieve a high level of reliability, the design must be supported by a rigorous formalism. The formal object-oriented method TROM [Ach95] has been studied as a formal basis for the development of real-time reactive systems.

## **2.2 TROM Formalism**

TROM formalism [Ach95] is founded on merging object-oriented and real-time technologies, and provides a formal basis for specification, analysis and refinements of the real-time reactive systems design.

### **2.2.1 Reactive Object Model**

A reactive object is modeled abstractly as a finite state machine augmented with ports, attributes, logical assertions on the attributes, and timing constraints [Ach95]. Synchronous message passing mechanism is being used among the reactive objects communication. Events are defined as external events (input or output) and internal events. An external event can only occur at a reactive object of a specific port type; an internal event occurs at so called *null* port. Two ports are compatible if the set of input messages at one port is equal to the output messages at the other port. A port link connects two compatible ports. All valid messages exchange among the objects in a subsystem is through the port links.

A generic reactive class (GRC) is a collection of reactive objects. The GRC has port types, attributes, logical assertions on the attributes and time constraints.

All the reactive objects, which are generated from the same generic reactive class, have the same attributes. From the object-oriented perspective view, a generic reactive class is a class declaration. Each individual reactive object is an

object of that particular class. TROM formalism is the basis used to specify GRCs and thus reactive objects in a system.

TROM is a three-tier formalism; each tier has its own output and processing mechanism.

### **2.2.2 First Tier – Larch Formalism**

Larch [GH93] supports a two tier specifications. One tier is *Larch Interface Language* (LIL), which is tailored to a specific programming language. The other tier is *Larch Shared Language* (LSL), which is an abstract algebraic specification. Typically, LIL uses the declaration syntax of a specific programming language, and adding annotations to specify the behaviours of the operations. These annotations consist of pre and post conditions.

LSL is a language for specifying state-independent mathematical abstractions of a system. An LSL *trait* is used to describe a mathematical theory. A trait is a basic unit of specification. In each trait, operators are introduced and defined with a set of equations that defines which terms are equal to one another, and may assert additional properties about the sorts and operators. A trait can use another trait by including it in the *includes* section of the specification. Figure 1 shows an example of the Set trait:



```

SetTrait(Set, E) : trait
  includes Integer
  introduces
    emptyset :  $\rightarrow Set$ 
    insert :  $E, Set \rightarrow Set$ 
    delete :  $E, Set \rightarrow Set$ 
    unionn :  $Set, Set \rightarrow Set$ 
    member :  $E, Set \rightarrow Bool$ 
    subset :  $Set, Set \rightarrow Bool$ 
    size :  $Set \rightarrow Int$ 

  asserts
    Set generated by emptyset, insert
    Set partitioned by member
     $\forall x, y : E, s, t : Set$ 

       $\neg(member(x, emptyset))$ 

       $member(x, insert(y, s)) == (x = y) \wedge member(x, s)$ 
       $member(x, delete(y, s)) == (x \neq y) \wedge member(x, s)$ 
       $member(x, unionn(s, t)) == member(x, s) \vee member(x, t)$ 

       $subset(emptyset, s)$ 

       $subset(insert(x, s), t) == member(x, t) \wedge subset(s, t)$ 
       $subset(delete(x, s), t) == subset(s, t)$ 
       $unionn(s, emptyset) == s$ 
       $unionn(s, insert(x, t)) == insert(x, unionn(s, t))$ 
       $size(emptyset) == 0$ 
       $size(insert(x, s)) == \text{if } member(x, s) \text{ then } size(s) \text{ else } 1 + size(s)$ 

  implies
    Set partitioned by subset
     $\forall x, y : E, s, t : Set$ 
       $insert(x, insert(x, s)) == insert(x, s)$ 
       $insert(x, insert(y, s)) == insert(y, insert(x, s))$ 
       $subset(s, t) \Rightarrow (member(x, s) \Rightarrow member(x, t))$ 
    converts delete, unionn, member, subset
    exempting  $\forall i : E$ 
       $delete(i, emptyset)$ 

```

Figure 1 LSL Trait for Set

### 2.2.3 Second Tier – TROM Class

A TROM class is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes and time constraints. Figure 2 shows a TROM object. All the reactive objects generated from generic reactive class communicate with its environment by synchronous message passing.

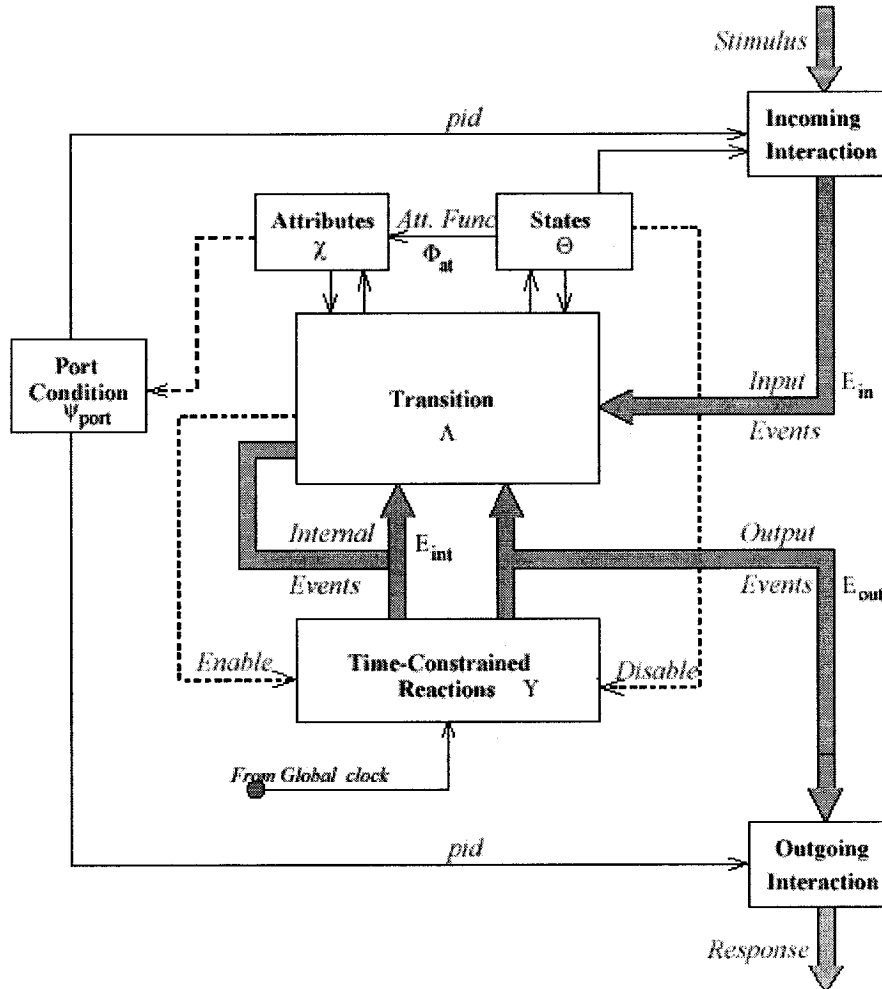


Figure 2 Anatomy of a Reactive Object

A TROM object is made of a set of events, states, attributes and its related functions, transitions and a set of timing constraints. There are three types of events: *external input*, *external output* and *internal* and they are represented

symbolically by  $e?$ ,  $e!$  and  $e$  respectively. A TROM object communicates with its environment by synchronous message passing, which occurs at a port.

An attribute of TROM object can be either one of the following types:

- Abstract data type imported from the first tier
- Port types

Each reactive object instance can have multiple port types. A port is an abstractly modeled bi-directional access point between environment and a TROM object. A port can only process a set of messages defined by its port type. The signature of a port type  $K$  is denoted as  $\mathcal{E}^K$ . The relationships between attributes and states are defined by attribute functions. A transition function describes the state change due to a particular event. A transition is caused by the occurrence of either an internal event or an external event. A timing constraint on a transition specifies the constraint of response to stimulus in terms of time.

A TROM object is an 8-tuple  $(P, \mathcal{E}, \Theta, X, \mathcal{A}, \Phi, \Lambda, \Gamma)$  such that  $P$  represents a finite set of port-types,  $\mathcal{E}$  is a finite set of events,  $\Theta$  is a finite set of states,  $X$  is a finite set of typed attributes,  $\mathcal{A}$  is a finite set of LSL traits,  $\Phi$  is a function-vector,  $\Lambda$  is a finite set of transition specifications and  $\Gamma$  is a finite set of time-constraints. A template of a GRC class is shown in Figure 3.

***Class<name>***

***Events:***

***States:***

***Attributes:***

***Traits:***

***Attribute-Function:***

***Transition-specifications:***

***Time-Constraints:***

***End***

Figure 3 Template for GRC class

### **2.2.4 Third tier – System Configuration Specification**

Each subsystem is the collaboration of the objects instantiated from the second tier. A system configuration specification (SCS) is defined to specify a reactive system or subsystem by composing reactive objects or by composing smaller subsystems. A template for a system configuration specification is shown in Figure 4.

***Subsystem <name>***

***Include:***

***Instantiate:***

***Configure:***

***End***

Figure 4 Templates for System Configuration Specification

## **2.3 TROMLAB**

TROMLAB is a framework for real-time reactive system development built on TROM formalism. The framework includes a number of tools to promote a rigorous development of real-time reactive systems.

The current architecture of the TROMLAB consists of the following components:

- Rose-GRC Translator – [Pop99] module which maps graphical model in Rational Rose to formal specification based on TROM;
- Interpreter – [Tao96] performs syntactically verification on specification and generate an internal representation of it;
- Simulator – [Mut96] creates animation of a subsystem based on internal representation generated by the interpreter;
- Browser for Reuse – [Nag99] an interface to a library which helps the user to navigate, query and access system components during the development;
- Graphical User Interface – [Sri99] an interface for the system developer to interact with the TROMLAB environment;
- Reasoning System – [Hai99] Debugging facility that allows the user to query the system behaviour based on interact queries;
- Verification Assistant – [Pom99] an automated tool that extracts mechanized axiom from real-time reactive systems;
- Test Cases Generator – [Zhe02] and [Che02] an automated tool for generating test cases from specifications.

This Major Report describes the design and implementation of the TROM-QM, a new TROMLAB automated tool for software quality measurement and measurement data collection and analysis.

The next Chapter introduces the design and implementation of the TROM-SCMS, complexity measurement component of the TROM-QM.

## **Chapter 3 Complexity Measurement**

### **3.1 Complexity Module in TROMLAB**

The complexity measurement system proposed in [Orm02] includes testability, functionality, complexity and reliability measures. One of the main goals of this Major Report is the implementation the complexity measurement component responsible for evaluating the complexity of a real-time reactive system based on SCS specifications in the TROMLAB environment. Architectural complexity is viewed in terms of how the software components interact through message passing mechanism, without considering the complexity of its components. The architectural complexity measures have to quantify objectively the amount of information exchanged between the objects.

For the architectural complexity measurement purposes, all the information related to the interaction between objects via message exchanges, has to be extracted from the TROMLAB design specification.

### **3.2 Architectural complexity measurement**

The complexity measurement and management in real-time reactive systems proposed in [Orm02] quantifies the level of complexity of a system specified in TROM formalism.

#### **3.3.1 Approach**

A system in TROMLAB is assumed to consist of a collection of reactive objects and there is interaction among these objects. The object behaviour is concerned with its state changes and interaction with other object (message passing). The

total structural complexity of the software system is a function of the internal complexity and the architectural complexity. This report is considering the architectural complexity only.

The architectural complexity could be simply to view in terms of how the software components interact through message passing mechanism. The information transfer between the components is synonymous with the complexity of interactions with the software system. So measuring the amount of information transfer is assessing the complexity management in real-time reactive systems [Orm02].

### 3.3.2 Mathematic Model for Architecture Complexity

The purpose of the architectural complexity measurement is a comparison of different designs in terms of the complexity of interactions within the objects. All the information related to the interaction between objects via message exchanges has to be extracted from the TROMLAB design specification. The architectural complexity measures quantify objectively the amount of information exchanged between the objects [Orm02]. An object-predicate table is abstracting the interactions between the objects. Each row in the table represents an object in the system, and each column represents a communication port. The formal definition of the object-predicate table is given below:

$$Object\_Predicate\_Table(object_i, port_j) = \begin{cases} 1, & \text{if } port_j \text{ belongs to } object_i \\ & \text{or is linked to a port of } object_i, \\ 0, & \text{if this is not the case.} \end{cases}$$

The concept of *excess-entropy* was introduced to quantify the complexity of information exchange based on the object-predicate table abstraction. The *excess-entropy*  $C$  is defined as the difference between the sum of the entropies of parts (subsystems, or *partitions*, from which the system is composed) and the entropy of the whole (system).

The formulas for calculating the entropy  $H$  and the excess-entropy  $C$  are:

$$(A): H = \log_2 n - \frac{1}{n} \sum_{i=1}^m n_i \log_2 n_i$$

$$(B): C = \sum_{i=1}^m H_i - H$$

$$(C): H_i = \log_2 n_i - \frac{1}{n_i} \sum_{j=1}^{k_i} p_j \log_2 p_j$$

Where  $m$  is the number of partitions in a system with  $n$  objects. Each partition  $m_i$  has  $n_i$  objects forming a subset of the set of  $n$  system's objects.  $H_i$  is the entropy calculated on the object-predicate table abstracting the *partition* <sub>$i$</sub> .  $k_i$  is the number of different rows configurations in the object-oriented table corresponding to the *partition* <sub>$i$</sub> ,  $p_j$  is the number of rows with the same configuration in the *partition* <sub>$i$</sub> .

And

$$n_i = \sum_{j=1}^{k_i} p_j$$

Architectural Complexity is defined as

$$AC = \frac{C}{C_{\max}}$$

$$C_{\max} = (n-1) \log_2 n$$



Local Architectural Complexity (LAC) is evaluated as a relative contribution of one object's interactions to the architectural complexity of a design.

If the given object belongs to only one set of partition, The *LAC* is defined as:

$$LAC_i = H_i / C_{max}$$

If the given object belongs to at least two partitions, then *LAC* is defined as

$$LAC = \frac{\sum_i H_i}{C_{max}}$$

### 3.3.3 Architectural Slice Extraction

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. Maintenance can be difficult if the coupling in the architectural design is unjustifiably high. In [Orm02], the maintainability is estimated based on the complexity of interactions between the objects in the design architecture.

There are two aspects of complexity of modification: a) *primary modification* due a given change; b) *secondary modification* describing the system's parts affected by the primary modification due to the effect of a given change.

An approach in assessing quantitatively the maintenance effort is related to the secondary as proposed in [Orm02]. A new static analysis technique – **Architectural Slicing** - useful in design complexity comparison is proposed. Architectural slicing extracts all the design objects relevant to the computations of a given one.

The algorithm for extraction of the architectural slices is defined as follows [Orm02]:

1. The given object belongs to only one set of interacting objects. In this case, the above set of objects would define the architectural slice for the participating objects.
2. The given object belongs to at least two sets of interacting objects. In general, one object may belong to different sets of interacting objects, in this case, the architectural slice is the union of the sets to which the interacting object belongs.

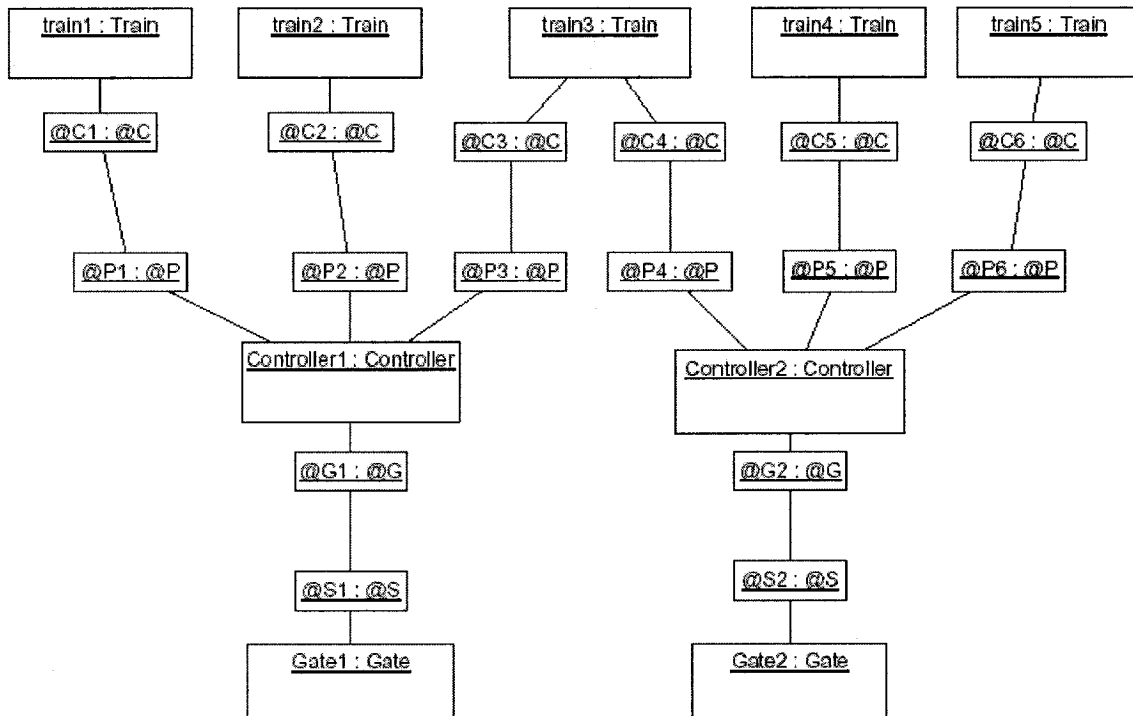
We illustrate the complexity measurement and maintainability assessment on a case study in the following section.

### **3.3.4 Gate-Train-Controller Case Study**

The Railroad crossing problem has been considered as a benchmark example by researchers in real-time systems community. In this section, we illustrate the theoretical computation results of the architectural complexity on the Gate-Train-Controller case study. Here we only introduce the information related our purpose. For more detail information of the problem, please refer to [Orm02].

The UML model for the Train-Gate-Controller problem has three generic reactive classes: Train, Controller and Gate. Each of these classes has aggregate of port types. There is an association between two port types, if the corresponding of the two generic reactive classes exist communication, since the GRC communication is through its port type. All communication information between the GRC are

defined in subsystem configuration specification file. Figure 5 illustrates the collaboration of a Train-Gate-Controller2 Subsystem with five trains, two gates and two controllers.



**Figure 5 Collaboration Diagrams for a Train-Gate-Controller2 Subsystem**

The formal specification of the above subsystem configuration is as shown in Figure 6.

## SCS TrainGateController

### Includes:

### Instantiate:

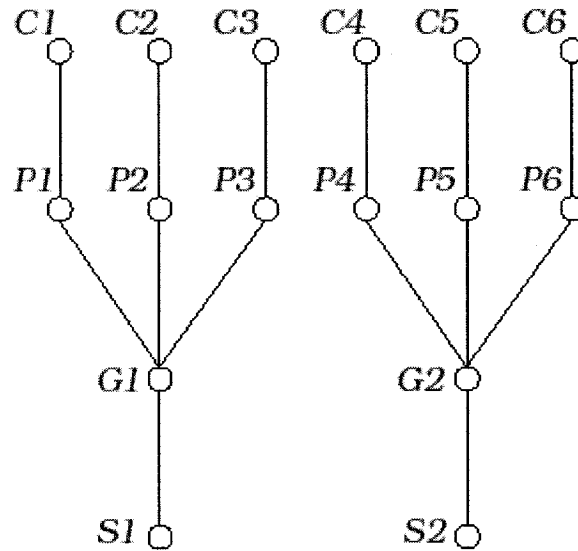
```
gate1::Gate[@S:1];  
gate2::Gate[@S:1];  
train1::Train[@C:1];  
train2::Train[@C:1];  
train3::Train[@C:2];  
train4::Train[@C:1];  
train5::Train[@C:1];  
controller1::Controller[@P:3,@G:1];  
controller2::Controller[@P:3,@G:1];
```

### Configure:

```
Controller1.@G1:@G<->gate1.@S1:@S;  
Controller2.@G2:@G<->gate2.@S2:@S;  
Controller1.@P1:@P<->train1.@C1:@C;  
Controller1.@P2:@P<->train2.@C2:@C;  
Controller1.@P3:@P<->train3.@C3:@C;  
Controller2.@P4:@P<->train3.@C4:@C;  
Controller2.@P5:@P<->train4.@C5:@C;  
Controller2.@P6:@P<->train5.@C6:@C;  
End;
```

**Figure 6 Train-Gate-Controller2 Subsystem Configuration Specification**

The interactions between the objects in Figure 6 are represented as a graph in Figure 7:



**Figure 7 Train-Gate-Controller2 Subsystem: graph for communication links**

The object-predicate tables for the two connected components of the figure7 are shown in Figure 8:

	t1	t2	t3	c1	g1
C1	1			1	
C2		1		1	
C3			1	1	
P1	1			1	
P2		1		1	
P3			1	1	
G1				1	1
S1				1	1

	t3	t4	t5	c2	g2
C4	1			1	
C5		1		1	
C6			1	1	
P4	1			1	
P5		1		1	
P6			1	1	
G2				1	1
S2				1	1

**Figure 8 Object-Predicate Table Abstractions for the Connected Components for figure 7**

In this example, the total number of components  $n$  is 16, the number of partitions  $m$  is 2.  $n_1 = 8$  in partition  $m_1$ ,  $n_2 = 8$  in partition  $m_2$ ,  $k_1 = k_2 = 4$ , and  $p_j = 2$  for  $\forall j$  in [1..4].

$$AC' = \frac{2(\log_2 8 - \frac{1}{8} \sum_{i=1}^4 2 \log_2 2) - (\log_2 16 - \frac{1}{16} \sum_{i=1}^2 8 \log_2 8)}{15 \log_2 16}$$

- LAC for the objects  $t_1, t_2, c_1, g_1$  :

$$LAC' = \frac{H_1}{C_{max}} = 0.03$$

- LAC for the objects  $t_4, t_5, c_2, g_2$ :

$$LAC' = \frac{H_2}{C_{max}} = 0.03$$

- LAC for the object  $T_3$  :

$$LAC' = \frac{H_1}{C_{max}} + \frac{H_2}{C_{max}} = 0.06$$

### Architecture Slice

The GRC objects in the system are  $t_1, t_2, t_3, t_4, t_5, c_1, c_2, g_1$  and  $g_2$ . By the definition the slices for the objects  $t_1$  and  $t_2$  consist of the objects  $\{t_1, t_2, t_3, c_1, g_1\}$ . The slice for the object  $t_3$  consists of the objects  $\{t_1, t_2, t_3, t_4, t_5, c_1, g_1, c_2, g_2\}$ . The slices for the object  $t_4$  and  $t_5$  consist of the objects  $\{t_3, t_4, t_5, c_2, g_2\}$ .

One of the main goals of the major report is to develop TROM-SCMS - a system to compute the architectural complexity, local architectural complexity, and the maintainability profile. The implementation of the complexity measurement component is described in the rest of the sections of this chapter.

### 3.3 Description of the module functionality

The TROM-SCMS is not a standalone module. It is integrated with other measurements in TROM-QM module, part of the TROMLAB environment.

The input to the TROM-SCMS system is the system's SCS file, which is generated from Rose-GRC Translator [Pop99]. The TROM-SCMS provides the APIs to get this file as input data and output Architectural complexity result, local architectural complexity and architectural slices.

### 3.4 Description of the components in the TROM-SCMS

The TROM-SCMS consists of four main components: ***Evaluator***, ***OPTable***, ***ObjectAndPort***, and ***Scs\_Parser***.

***Scs\_Parser***: AC and reliability modules share this class. It has a function to parses the SCS file, to get the ports, objects and the relationship between the ports and the objects. (Here the objects represent GRC object, ex: train1, gate1. the ports are port object, ex: P1, G1).

***ObjectAndPort***: Encapsulate objects and ports with the object name and it's class name (for example, **train1**, **P1** are object name, and **Train** and **P** are the class name).

***OPTable***: Encapsulate a table by two vectors (port list and object list) and a two-dimension array.

**Evaluator.** This is most importance component in the TROM-SCMS system. It possesses two tables, one is the object predicate (object-port) table, and another is port-port table. All of our results are come from these two tables.

### 3.5 Architecture Diagram

The architecture diagram of the TROM-SCMS system is shown in the Figure 9.

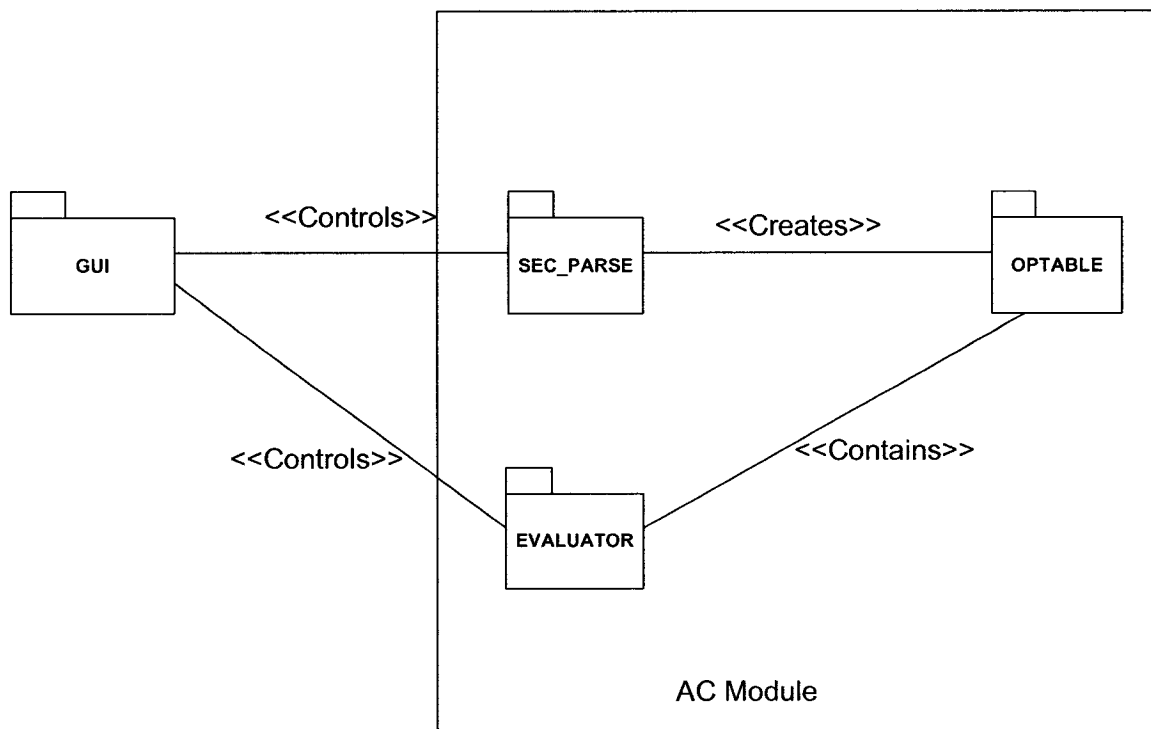


Figure 9 the Architecture Diagram of AC modules

### 3.6 Data Flow Diagram

The TROM-SCMS module takes an input file (SCS) from client (GUI), then parses the input file to produce *OPTable* objects, which are aggregated by the



Evaluator. The module's client then could get the result from Evaluator, which has several APIs to provide the results.

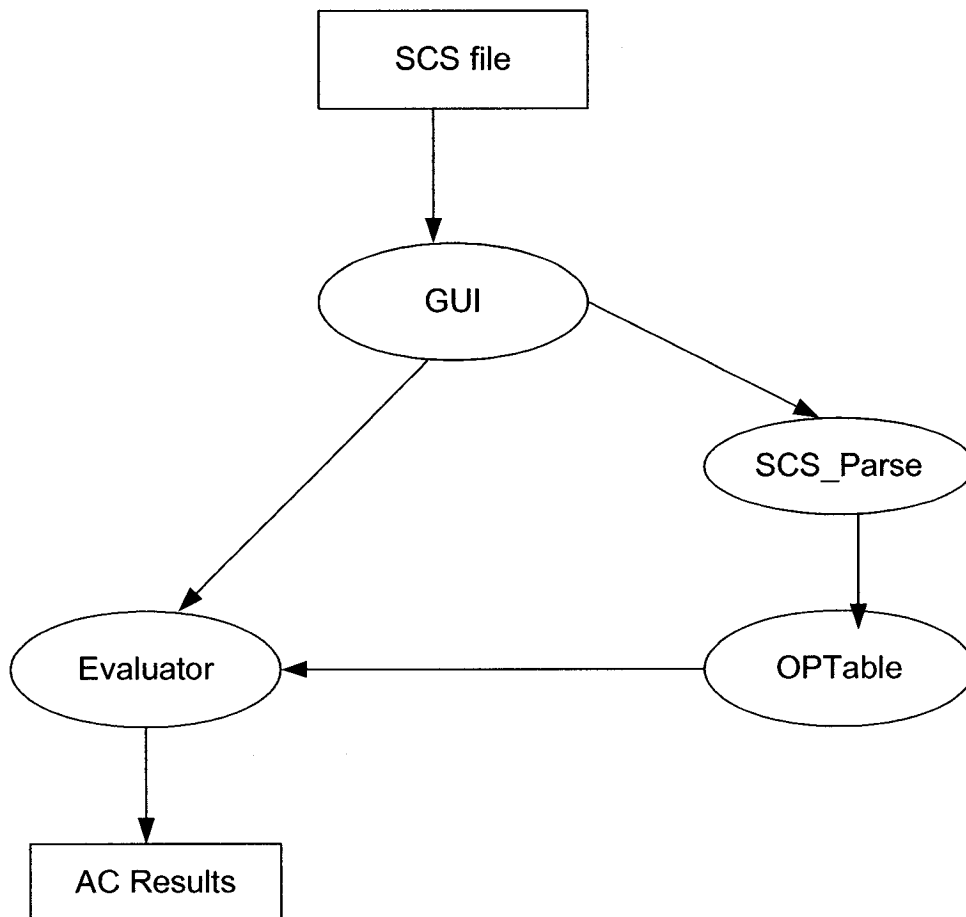


Figure 10 the Data Flow Diagram of AC Module

### 3.7 Class Diagram

The class diagram is shown in Figure11. It illustrates all classes in the AC module and their relationships.

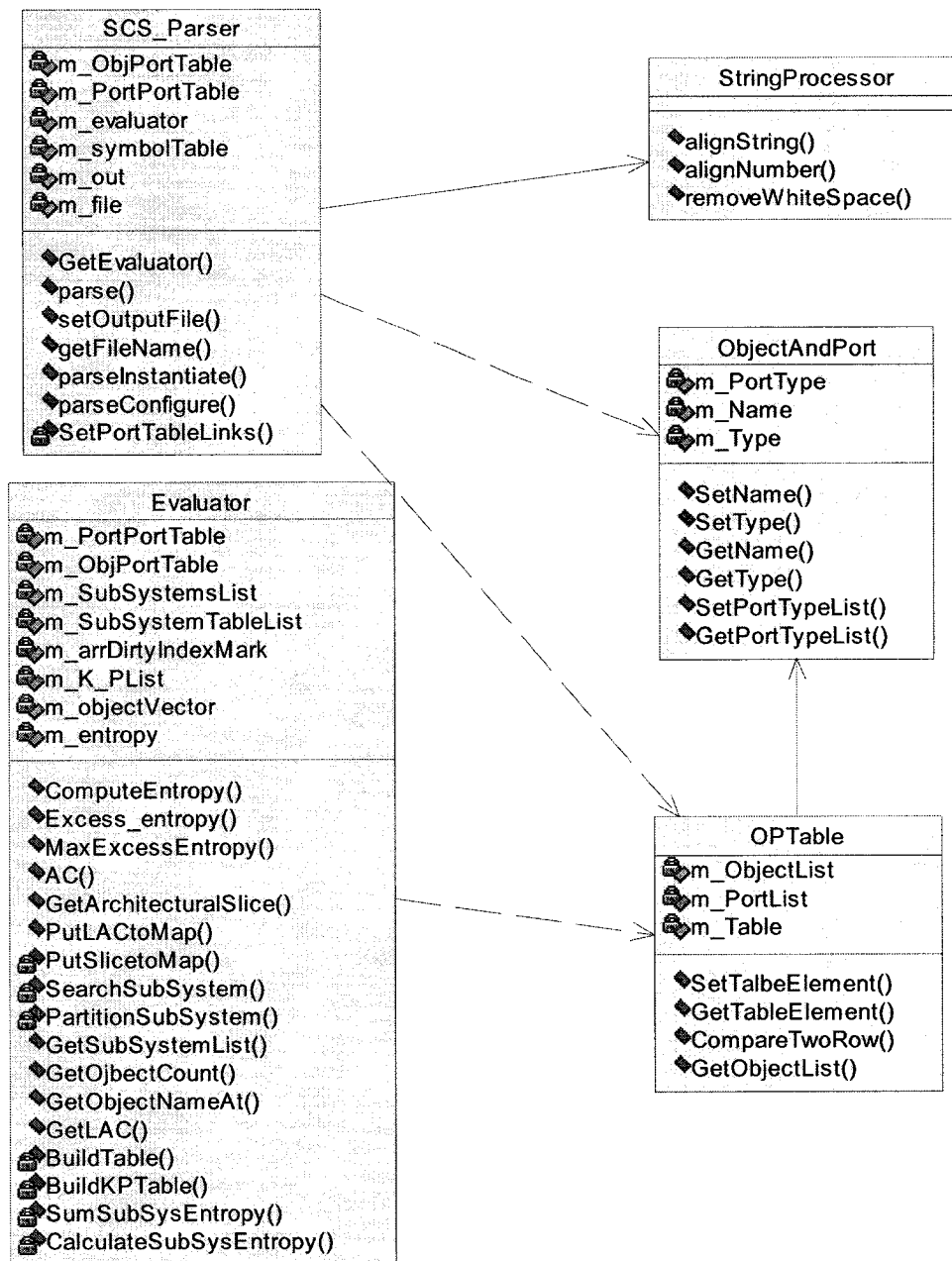
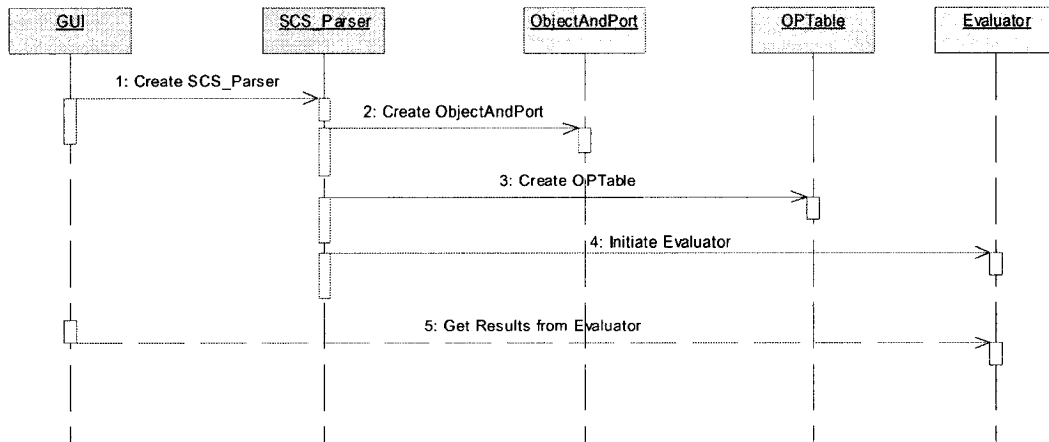


Figure 11 the Class Diagram of the AC Module

### 3.8 Sequence Diagram

The sequence diagram dynamically illustrates the flow of the control of the AC component.



**Figure 12 the Sequence Diagram for AC Module**

From the sequence diagram we can know that the GUI creates SCS\_Parser object; inside of the SCS\_Parser, the ObejctAndPort objects are created to initialize the OPTable objects. With the OPTable objects, Evaluator object could be created. At last the GUI could get the results from Evaluator object.

The following section introduces the algorithms implemented in TROM-SCMS.

### 3.9 Algorithms used in TROM-SCMS module

This section describes key algorithms used in computing AC, LAC and getting the object's slice in detail. The key point in TROM-SCMS system is dealing with two tables; one is object-predicate (object-port) table; another is port-port. We will use the Gate-Train-Controller case as example to explain the key algorithms used in this module.

#### 3.9.1 Algorithm of producing the object-port and port-port tables

As the Gate-Train-Controller case example, the object predicate table and port-port table are shown as table 1 and table2.

In Table1, each column represents the GRC object, and row represents the port object. Since the port C1 is belongs to object train1, so there is a link between them, mark as 1 in the table. C1 is link to P1 and P1 belongs to object control1, so C1 has a link to control1 too. This is according to the object predicate table definition [Orm02].

	Train1	Train2	Train3	Train4	Train5	Train6	Gate1	Gate2	Control 1	Control 2
C1	1								1	
P1	1								1	
C2		1							1	
P2		1							1	
C3			1						1	
P3			1						1	
C3			1							
P7			1							
C4				1						1
P4				1						1
C5					1					1
P5					1					1
C6						1				1

P6						1				1
S1							1		1	
G1							1		1	
S2								1		1
G2								1		1

**Table 1 Object Predicate Table**

In Table2 both column and row represent port object. This table is obtained by parsing the configuration section in SCS file. If the pair of ports exist a link in configuration, we put a link in the table between the two ports. Since the controller1 and controller2 have two types of ports, there exist links between these two types of a port. Thus we have link between G1 and P1, P2, and P3; G2 and P4, P5, and P6.

We will use this table to find the partitions and generate the subsystem table.

	C1	P1	C2	P2	C3	P3	C3	P7	C4	P4	C5	P5	C6	P6	S1	G1	S2	G2
C1		1																
P1	1															1		
C2				1														
P2			1													1		
C3						1												
P3					1											1		
C3								1										
P7							1											1
C4										1								
P4									1									1
C5												1						
P5											1							1
C6														1				
P6													1					1
S1																1		
G1		1		1		1									1			
S2																		1
G2								1		1		1		1			1	

**Table 2 Port-Port Communication Table**

### 3.9.1.1 Data type to represent the table

We define a data type **OPTable** to represent the table. In **OPTable**, there are two Vectors: One stores the objects in the column; another store the objects in the row, and a two-dimension integer arrays to represent the link between the column and the row. We also define **ObjectAndPort** data type to represent the GRC object and port. Both GRC object and port object have name and type. GRC has aggregated of port types. We define **ObjectAndPort** data type with *name*, *type* and *port types list* to encapsulate both GRC objects and ports object.

### 3.9.1.2 Produce the tables

The input data for the TROM-SCMS is SCS file. The TROM-SCMS system gets the file, and parses it to produce the object predicate table and port-port link table.

There are two functions to parse SCS files and produce two **OPTable** objects.

Function ***ParseInstantiate()*** only produce the objects vector.

Function ***parseConfigure()*** has produced the port vector and set links in object predicate table and port-port link table.

***Pseudo code:***

***Pre-Condition: {The SCS file is in correct format}***

***Post-Condition: {Object predicate table and Port-port table is produced}***

Void ***ParseInstantiate()***

**Begin**

**Loop** read a line from SCS file

Parse text in the line to find the GRC object name and type in  
“Instantiate” section.

Create an instance of ObjectAndPort; initiate with the name and  
type.

Parse port types, which belong to the GRC objects.

Set the above instance’s port type list.

Add this completed GRC object to column (GRC object list) vector  
in **object predicate table**

**End loop**

**End ParseInstantiate**

Void ***parseConfigure ()***

**Begin**

**Loop** read a line from SCS file under the “**Configure**” section.

Parse string before the symbol “<->” in the line

Get GRC object and its port name and port type.

Create an instance of ObjectAndPort; initiated with the port name  
and port type.

Add above instance to row (port list) vector in the object predicate  
table.

Find the column index in the **object predicate table** by using the GRC object name.

Set link at the **object predicate table** by using the column index and current row.

Add above instance to column (port list) in the port-port table.

Add above instance to row (port list) vector in the port-port table.

Parse string after the symbol "<->" in the line

Get GRC object and its port name and port type.

Create an instance of ObjectAndPort; initiated with the port name and port type.

Add above instance to row (port list) vector in the object predicate table.

Find the column index in the **object predicate table** by using the GRC object name.

Set link at the **object predicate table** by using the column index and current row.

Add above instance to column (port list) in the port-port table.

Add above instance to row (port list) vector in the port-port table.

### **End Loop**

Set link at Port-port table.

### **End *parseConfigure***



### 3.9.2 Algorithms for Calculating AC, LAC and Generating the Object

#### Slice

Component **Evaluator** is responsible for calculating Architecture complexity and local architecture complexity, and find out the list of GRC object slices. By the definition of the AC and LAC, we have to heavily use the **object predicate table** and **port-port table**. Hence we define two **OPTable** objects in **Evaluator** component.

#### 3.9.2.1 Data member in evaluator component

Two **OPTable** objects ( *m\_PortPortTable*, *m\_ObjPortTable*): One represents **object predicate table**; the other represents the **port-port table**.

Vector *m\_SubSystemTableList*: The system could be partitioned to several subsystems. Each subsystem has its own object predicate table. This vector data member *m\_SubSystemTableList* holds all of the subsystem objects predicate tables.

Vector *m\_K\_Plist*: In calculating the partition entropy  $H_i$ , we have to find the  $p_j$ , which is the number of rows with the same configuration. The vector data member *m\_K\_Plist* stores  $p_j$ . If there are five kinds of different configuration, the total elements in the vector *m\_K\_Plist* are five. The number of same configuration  $k_i$  is stored by each element.

Vector *m\_SubSystemsList*: before we can calculate the architecture complexity, we have to partition the system to subsystem and build an object predicate table for each of the subsystem. The ports in subsystems are exclusive. We use a

vector to store the subsystem's ports and add all of these vector objects to the vector *m\_SubSystemsList*.

### 3.9.2.2 Calculate Architecture Complexity

The main functions for calculating the architectural complexity are ***PartitionSubSystem()***, ***SearchSubSystem()*** and ***buildTable()***.

***PartitionSubSystem()***: Partition the system to subsystem, output is the vector *m\_SubSystemsList*, which holds the subsystem ports list.

***SearchSubSystem()***: this is a key function used to partition the system. It takes two parameter as inputs, one is the *row*; the other is a *subsystem* (Vector). This function will loop the *row* from first column to last column to find the links. If there exists a link in the [*row*, *column<sub>i</sub>*], and the *column<sub>i</sub>* is not existed in the *subsystem*, add the *column<sub>i</sub>* to the *subsystem*. Pass the *column<sub>i</sub>* as a parameter to the ***SearchSubSystem*** function. Let it recursively search the links. At last, all ports existed an interaction with the port in this *row* will be found and the index of the column of the ports will be added to the *subsystem* vector.

***buildTable()***: build the subsystem's object predicate table. With the result of ***PartitionSubSystem()***, we have the number of partition of the system (size of the *m\_SubSystemsList* ), then we could build subsystem's object predicate table.

***Slice()*** : a Slice's constructor. It takes a GRC's object name as parameter and produces a slice into data member *m\_vectorSlice*. The two data member is defined as public; this class is private for the ***Evaluator*** class. So after constructor *Slice*, we can get the GRC object's slices

**Pseudo code:**

Pre-Condition: {The system object predicate table and the port-port table have built properly.}

Post-Condition: {Subsystem table is produced}

void **PartitionSubSystem()**

**Begin**

**Loop** port-port table' port list (*column*)

**if** current row is marked as dirty then

create a Vector *aNewSubSystem*

**SearchSubSystem**( *column*, *aNewSubSystem*);

Add the *aNewSubSystem* to *m\_SubSystemsList*.

Mark the current row as dirty

**End if**

**End loop**

**End**

Void **SearchSubSystem** ( int *aRow*, Vector *aNewSubsystem* )

**Begin**

**Loop** port-port table' port list (*column*)

**If** there is a link at [*row*, *column<sub>i</sub>*] **then**

**If** the *NewSubsystem* not contain the index of the *column<sub>i</sub>*;

**then**

Add this index to the *NewSubsystem*.

Mark this *column<sub>i</sub>* as dirty.

***SearchSubSystem***(*column<sub>i</sub>* , *aNewSubsystem*)

**End if**

**End if**

**End Loop**

**End *SearchSubSystem***

Void ***buildTable*** ( Vector *subsystemList*, OPTable *aNewTable*):

**Begin**

**Loop** subsystem's port list

Get the port's index from the subsystem port list (this index is the index of row in the object predicate table)

**Loop object predicate table** object list (column)

**If** there is link in [*row*, *column<sub>i</sub>*] **then**

Add this **GRC object** to *aNewTable*

**End if**

**End Loop object predicate table**

**End Loop** subsystem's port list

Set the links in subsystem object predicate table

**End *buildTable***

***Slice***( String *Objectname* )

**Begin**

**Loop** *m\_SubSystemTableList* (number of tables)

Search the *Objectname* in each table.

**If** found *Objectname in the table* **then**

Add all the **GRC objects** in this table to *m\_vectorSlice*.

**End if**

**End Slice**

## Chapter 4 TROM-QM Module

### 4.1 Introduction TROM-QM

The quality measurement for real-time reactive systems in TROMLAB quantifies and analyses several aspects of software quality, such as testability, functionality, complexity and reliability. Each of them is implemented independently. However, a measurement tool to measure a real-time system quality is required to integrate all of these modules together and provide a graphical user interface for ease of use. The main goal of this report is to integrate the measurement components, and to design and develop a common user interface. The provided by TROM-QM User Interface is described in the following section.

### 4.2 TROM-QM User Interface

There are two main components for the GUI user interface in TROM-QM. The first frame is the **Measurement Tools** shown in Figure 13. User can optionally select one or more of the testability, functionality, reliability or complexity measurements to assess the complexity or/and reliability of the system being developed at the design phase. When one is selected, the corresponding browsing file button will be enabled, and then user could chose input files. Reliability needs three kinds files: GRC specification files, synchronous product machine specification file and system configuration file [Lee03]. The **Select GRC file** button allows the user to multi-select files. Other file browse buttons are single file selection. The text boxes display the files path chosen by the user.

The input file for the Architecture Complexity is SCS file, same as the one in reliability measurement. Hence, if the reliability is selected, there is no need to browse files for Architectural complexity. The button of **Select SCS file** for AC is disabled in this case. It will be enabled only when the Architecture complexity option is selected and Reliability option is unselected.

We have error checking functionality included in this interface as well. If a measurement is selected, but the text fields (file path) are empty, it will display error message when pressing the **OK** button.

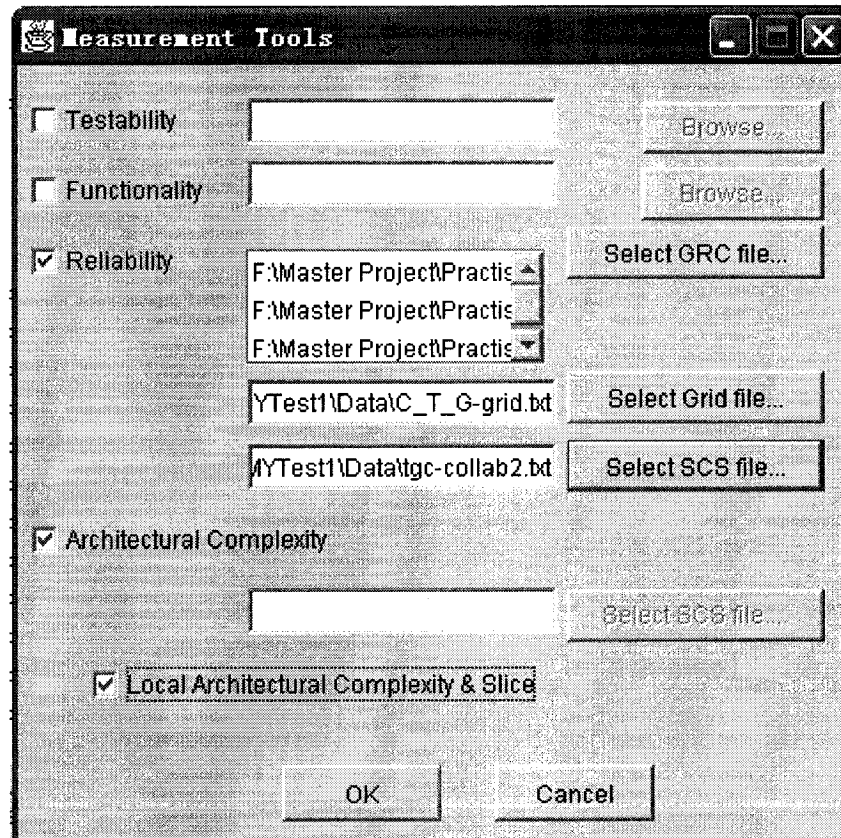


Figure 13 User Interface - Measurement Tools User

The second frame displays the measurements results. There are three different views of the results.

The first view is text view shown in Figure 14. It displays the selected measurement results in text description.

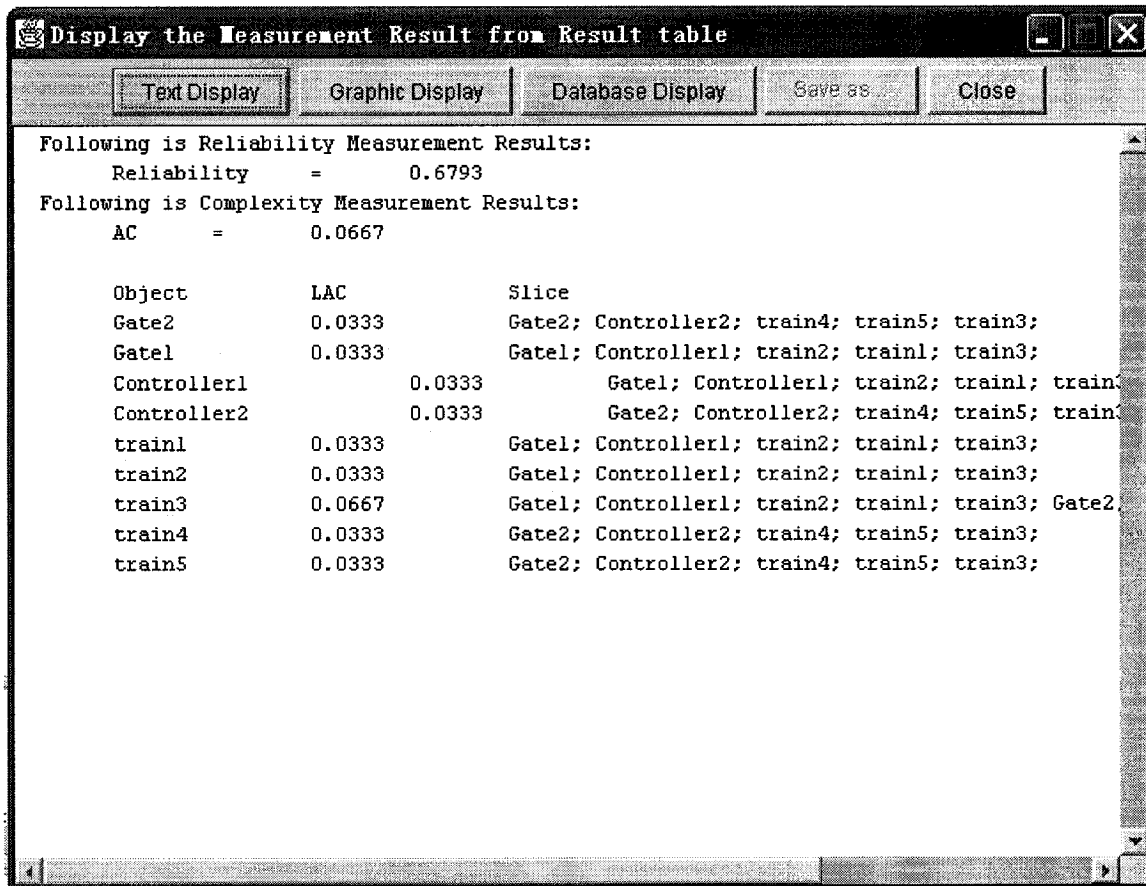


Figure 14 User Interface - View of Measurement Results -1

The second view is the graphic view, which displays the measurement results graphically. The right bar graph shows the LAC values. The left graph shows the result of reliability and architectural complexity. The range for both reliability and architectural complexity are 0 to 1 by default. In some case we may get the measurement results are too small to display in the graph. We design to let user



reset the upper bound of the range. Thus increasing the percentage of the measurement result and it could display properly in the graph. From the Figure 15 we can see there is a **set upper bound check box**. If user checks this box, the four edit boxes for AC, Reliability, Testability and Functionality will become editable, and the **Update** button will be enabled. Then user could set appropriate values, and press the **Update** button, the upper bound value will update to the bottom table and the graphic will be adjusted. This value will also be updated to database.

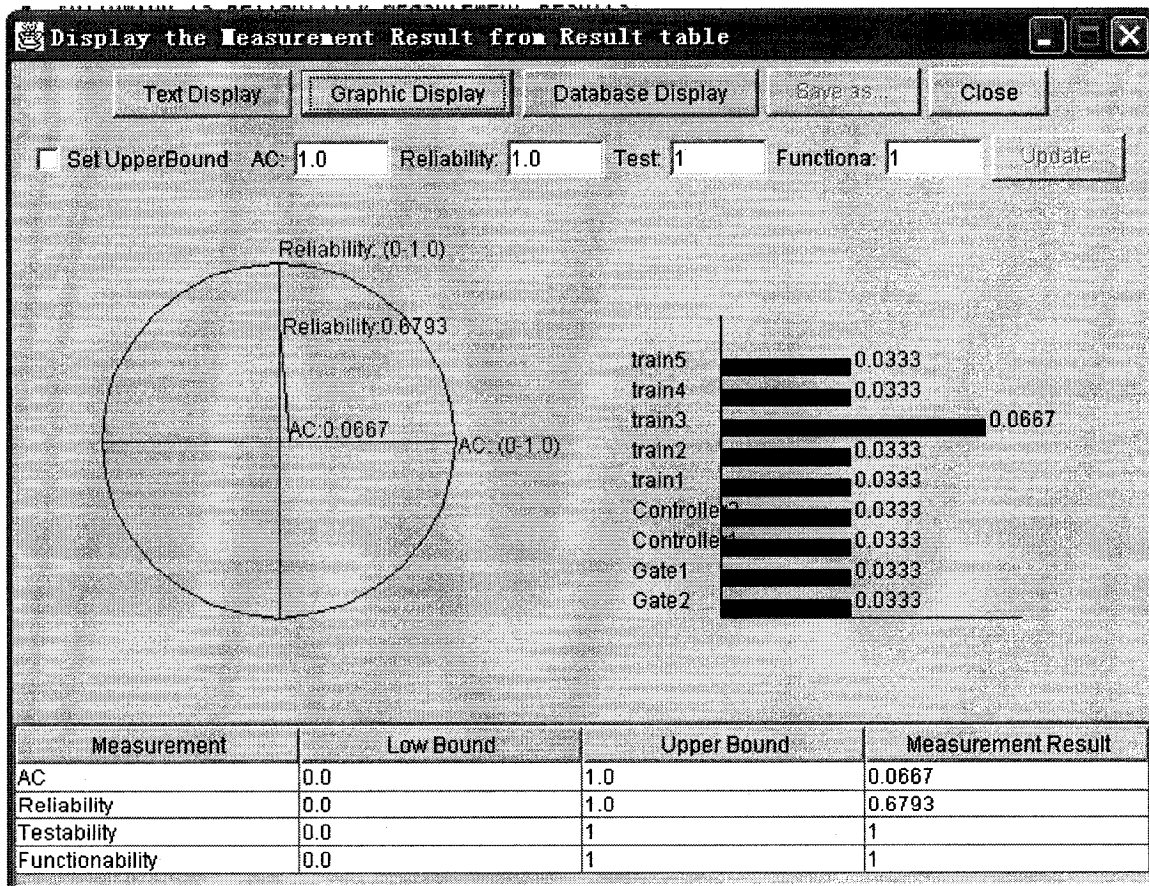
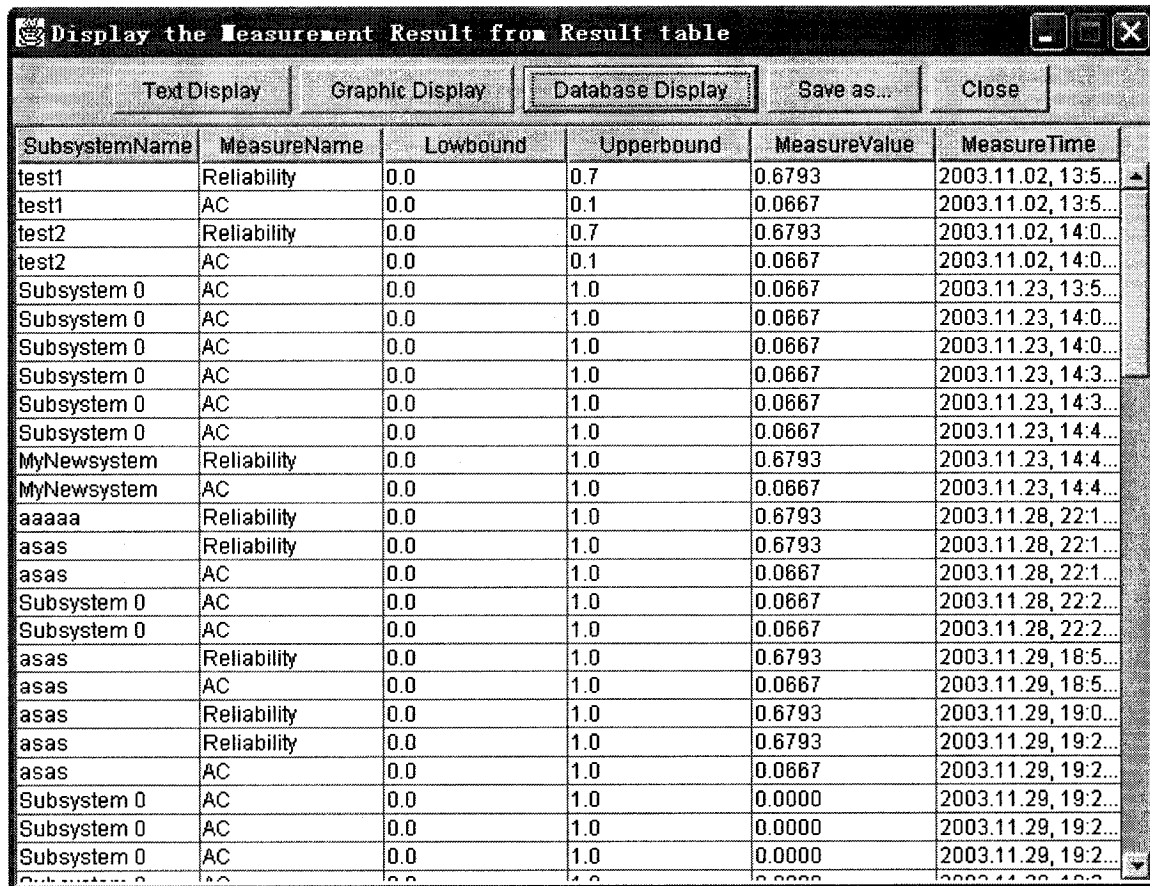


Figure 15 User Interface - View of Measurement Result – 2

The third view shows the database in table (Figure 16). This table is same as the database table, except the measure time. The measurement time value in the database is represented with millisecond, in this table it is formatted to normal time expression.



SubsystemName	MeasureName	Lowbound	Upperbound	MeasureValue	MeasureTime
test1	Reliability	0.0	0.7	0.6793	2003.11.02, 13:5...
test1	AC	0.0	0.1	0.0667	2003.11.02, 13:5...
test2	Reliability	0.0	0.7	0.6793	2003.11.02, 14:0...
test2	AC	0.0	0.1	0.0667	2003.11.02, 14:0...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 13:5...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 14:0...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 14:0...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 14:3...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 14:3...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.23, 14:4...
MyNewsystem	Reliability	0.0	1.0	0.6793	2003.11.23, 14:4...
MyNewsystem	AC	0.0	1.0	0.0667	2003.11.23, 14:4...
aaaaa	Reliability	0.0	1.0	0.6793	2003.11.28, 22:1...
asas	Reliability	0.0	1.0	0.6793	2003.11.28, 22:1...
asas	AC	0.0	1.0	0.0667	2003.11.28, 22:1...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.28, 22:2...
Subsystem 0	AC	0.0	1.0	0.0667	2003.11.28, 22:2...
asas	Reliability	0.0	1.0	0.6793	2003.11.29, 18:5...
asas	AC	0.0	1.0	0.0667	2003.11.29, 18:5...
asas	Reliability	0.0	1.0	0.6793	2003.11.29, 19:0...
asas	Reliability	0.0	1.0	0.6793	2003.11.29, 19:2...
asas	AC	0.0	1.0	0.0667	2003.11.29, 19:2...
Subsystem 0	AC	0.0	1.0	0.0000	2003.11.29, 19:2...
Subsystem 0	AC	0.0	1.0	0.0000	2003.11.29, 19:2...
Subsystem 0	AC	0.0	1.0	0.0000	2003.11.29, 19:2...

Figure 16 User Interface - View of Measurement Result -3

The **Save as** button is disabled if the results view is not database table view. When the table view is visible, the **Save as** button is enabled. If user presses it, it will pop standard **Save as Dialog** shown in Figure 17 to allow user save the table view as a Microsoft Excel file.

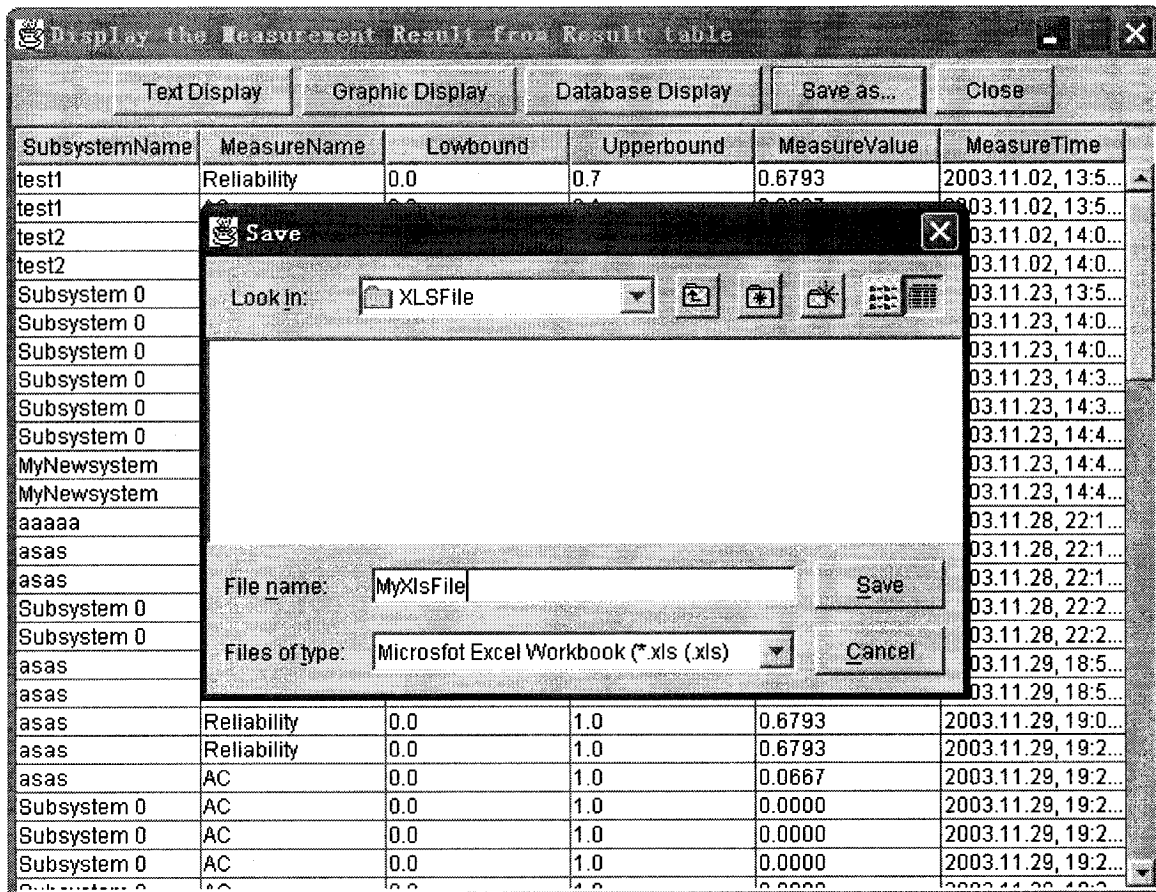


Figure 17 Save as the results in table view to Excel file

### 4.3 Description of the components in the TROM-QM

The TROM-QM contains three classes. **MyApplication1**, **Frame1**, and **CardDeck**. The three classes are specified as follows:

**MyApplication1**: contains main function to run the application.

**Frame1**: this is the key component. It presents the user interface to accept the user input files, parse the files and call the corresponding measurement modules to calculate the result and update the result to database

**CardDeck:** Display the result in different views, which includes text view, graphic view and table view. It also provides user to save as the results in table to Microsoft Excel file.

#### 4.3.1 Measurement Tools Interface

The **Frame1** gets the input files from user's action and parse these files. *Browse()* function call the *JFileChooser* to pop the standard choose file dialog. Each **Select file** button has event handling function to set the file paths to the file path fields. Figure 18 shows an example of the event handling function of **Select SCS file** button in architectural complexity.

```
void Browser_actionPerformed(ActionEvent e) {  
    Vector vfileName = new Vector();  
    if ( !BrowseFile(vfileName, false))  
        return;  
    File file = (File) vfileName.elementAt(0);  
    String strPath = file.getPath();  
    jTextField6.setText(strPath);  
}
```

**Figure 18 Select SCS file event handling function.**

*BrowseFile()* function takes a vector and a Boolean as parameters and pop Open file dialog. If the Boolean parameter is true, then open dialog allows multi-selection files. The selected files will add to the vector parameter. If the Boolean parameter is false, the open dialog only allows single file selection. Thus the vector has only one file. In this example, we pass false to *BrowseFile*, so the

*vfileName* vector only has one file. We get this file's path and set it to the text field in GUI user interface.

**Ok** button event handling function performs the following jobs:

- Connect database
- Call corresponding calculating function if it is checked.
- Update database if any measurement occurred.
- Create Measurement Results Dialog to display the result.

Example of the function of *OK\_actionPerformed()* is shown in Figure 19:

```
void OK_actionPerformed(ActionEvent e) {  
  
    ConnectToDatabase();  
  
    if (bjCheckBox1Sel){  
        ComputerTextability();           //not implementation yet  
        UpdateDatabase(1);  
    }  
    else  
        m_resultTable.put(sTest, "0");  
  
    if (bjCheckBox2Sel){  
        ComputerFunctionality();          //not implementation yet  
        UpdateDatabase(2);  
    }  
    .....  
    if (bjCheckBox3Sel){  
        InitGRC();  
        InitGrid();  
        initSCS(true);  
  
        .....  
        ComputerReliability();            //calculate Reliability  
        .....  
        UpdateDatabase(3);  
    }  
    .....  
    .....  
  
    if (scsFileCreated_only){  
        ComputerAC();                     //Calculate Architectural Complexity  
        UpdateDatabase(4);  
    }  
}
```

```

.....
    m_ResultFrame = new CardDeck(this);
.....
}

```

Figure 19 OK event-handling functions

In the *OK\_actionperform()* function we have put the function *ComputerTextability()* and *ComputerFunctionality()*, which are not implemented in this major report.

In this TROM-QM we now have two functions for quantifying correspondingly the architectural complexity and reliability: **ComputerAC** and **ComputerReliability()**. Before we invoke functions *ComputerAC()* or *ComputerReliability()* we have to parse input files and initiate some objects. We have *InitGRC*, *InitGrid* and *InitSCS* function to deal with the input GRC specification files, synchronous product machine specification file and system configuration file respectively. The following example show the *InitSCS()* function.

```

void initSCS(boolean blsReliability){
    String strFile;
    .....
    strFile = jTextField6.getText();
    File file = new File (strFile);
    m_scs = new Scs_Parser(file);
    .....
}

```

It gets the file path from the text fields and create a file object; then pass to *Scs\_Parser* constructor, which parse the SCS file and generating a object of *Evaluator*.

**ComputerAC()** is responsible to calculate architecture complexity and local architecture complexity by using the evaluator object. The results are put into data member *m\_resultTable* (Map) in order to display the result in the next interface. The slice result is put in another data member *m\_SliceTable* (Map).

```
private void ComputerAC()
{
    evaluator = m_scs.GetEvaluator();
    double AC = evaluator.AC();
    DecimalFormat form = new DecimalFormat("##0.0000");
    m_resultTable.put(sAC, form.format(AC));
    evaluator.PutLACtoMap(m_resultTable);
    evaluator.GetArchitecturalSlice();
    evaluator.PutSlicetoMap(m_SliceTable);
}
```

Figure 20 Calculating the Architectural Complexity function

*Evaluator* class is a component in the **AC** module. It is well constructor by the *Scs\_Parser* class. With the *Evaluator* object we could get the result of AC, and the objects' slice. Here we put the AC result to *m\_resultTable* data member and put the slice to another data member *m\_SliceTable*. We use two *Map* data members here, since the object's name is a key in both *m\_resultTable* and *m\_SliceTable*.

**ComputerReliability()** is responsible to calculate system reliability. The reliability module is a ready component. We just integrate this module to the TROM-QM. The **ComputerReliability** function is the interface to reliability module. When we get the result we have to put the result to *m\_resultTable* in order to display it later.

**UpdateDatabase()** is responsible to update the measurement database when there is any measurement occurred. This function accepts an integer as a parameter. It is updated the database according to the parameter passed in.

We use Microsoft Access database and the table is predefined. We have to register the predefined database as ODBC data source. The measure time in the database table is primary key. We get this value from system time in millisecond.

#### **4.3.2 Design of Measurement Results Interface**

All measurement results display in the Measurement Results Interface. We design this frame by using CardLayout layout manager to arrange three components (panels) into a “deck”. Only the top card (panel) is visible. So we have three different views of the results: text view, graphic view and database view.

Class **CardDeck** is derived from JFrame. It takes the *Frame1* as a parameter in constructor. All necessary information of the calculation results is got from the parameter.

There are two panels on main container, **button panel** and **deck panel** shown in figure 21. **Button panel** is on the north of the container; and **deck panel** is on the south of the container. There are four buttons on the button panel.

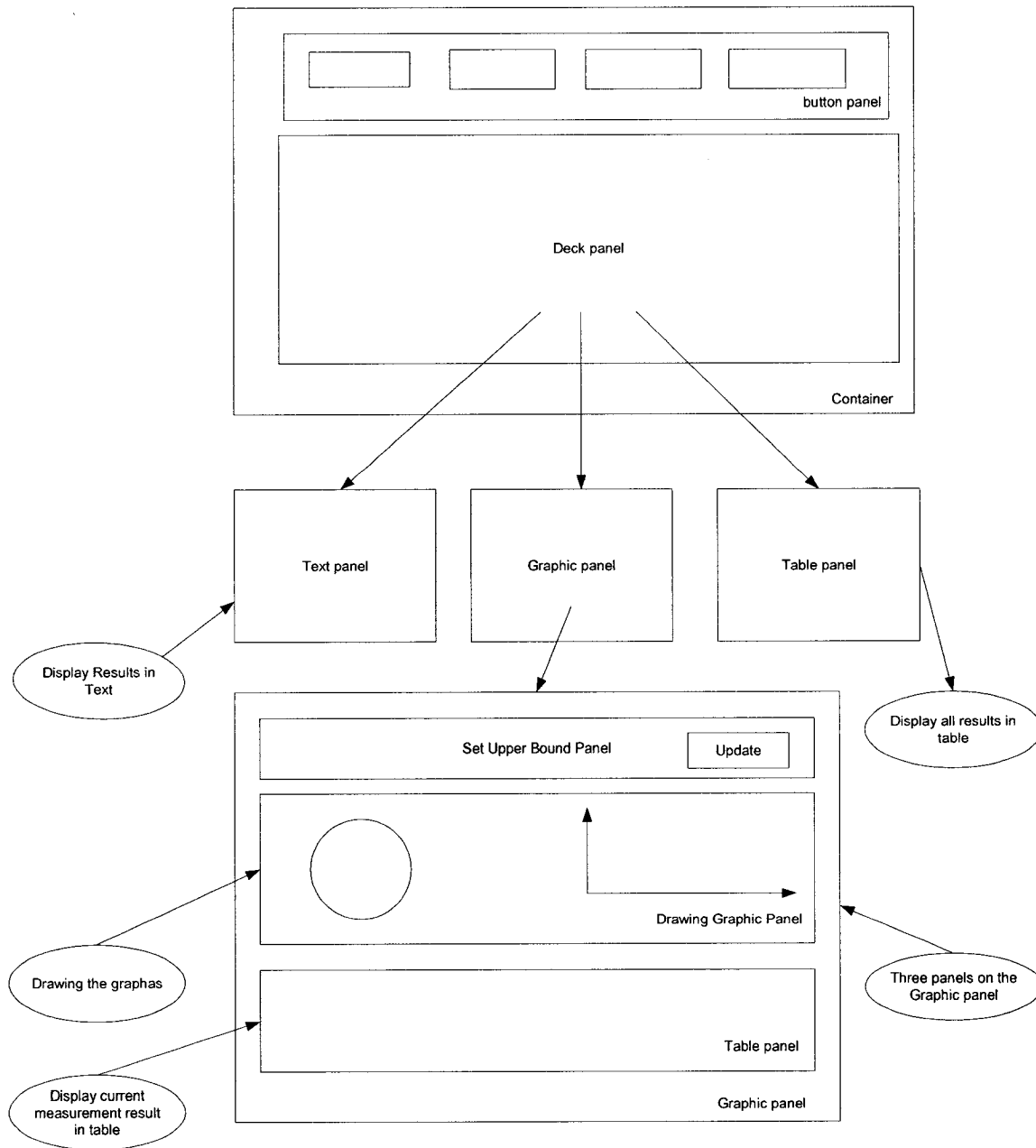
**Deck panel:** we deck three panels on the Deck panel: text panel, graphic panel and table panel. Text panel is visible by default. When a button in the button panel is pressed, an action is performed to make a corresponding panel on the Deck to be visible

**Text panel:** it shows the measurement result in text format.



**Graphic panel:** it shows the measurement result in graphic, which contains three panels: **set upper bound value** panel, **graphic drawing** panel and **table view** panel. The **graphic drawing** panel is responsible to draw the two graphics.

**Table Panel:** it shows all the measurement results stored in database. The table view is same as the database except the measure time field. We format it as a normal time representation in the display table instead of the milliseconds. When this panel is on the top of the deck, the button **Save as** will enable. When the **Save as** button is pressed, it prompts users to enter a file name. After the **Save** is pressed, the result in the table will be saved as a Microsoft Excel file.



**Figure 21 Designs of View Measurement Results**

In this chapter, we have introduced the key components of TRO-QM in detail. In the following Chapter 5 we emphasize on the steps of integration a new modules to TROM-QM.

## Chapter 5 Integration Guide and User Guide

### 5.1 Integration Guide

The measurements of architecture complexity and reliability are already integrated to TROM-QM. The testability and functionality modules could be integrated to TROM-QM when they are ready, following the steps as described in this Chapter.

In the **Frame1** class: the main jobs include:

- Provide an API to get the calculated result;
- Update OK button event handling function
- Update the function **UpdateDatabase()** to make the measurement result into database.

In the **CardDeck** class: the main jobs include:

- Write **FormatTestResult()** and **FormatFuncResult()** functions in order to display the result in text view.
- Update **DrawCircle()** function.
- Update **UpdateDatabase()** function.

The function **ComputeAC()** and **ComputerReliability** could serve as examples for the new API of **ComputerTextability()** and **ComputerFunctionality()**. Inside these APIs function the result should be put into the data member **m\_resultTable** (Map). The key for testability and functionality are already defined as *sTest* and *sFunction*. The entire jobs related to the quantification of testability and functionality should be done inside the APIs.

## 5.2 User Guide

The steps of running the TROM-QM are as follows:

- Create a Microsoft Access Database file; the file name is not important.

For example *measurement.mdb*

- Create a table “**Result** “ with the design view as figure 22. The name of the table and the name of the columns are hardcode in source code; we should use the exact table name and column name like figure 22.

Result : Table						
	SubsystemName	MeasureName	Lowbound	Upperbound	MeasureValue	MeasureTime
	test1	Reliability	0.0	0.7	0.6793	1067799451465
	test1	AC	0.0	0.1	0.0667	1067799451480
	test2	Reliability	0.0	0.7	0.6793	1067799677199
	test2	AC	0.0	0.1	0.0667	1067799677215
	Subsystem 0	AC	0.0	1.0	0.0667	1069613832059
	Subsystem 0	AC	0.0	1.0	0.0667	1069614158559
	Subsystem 0	AC	0.0	1.0	0.0667	1069614316293
	Subsystem 0	AC	0.0	1.0	0.0667	1069616238686
	Subsystem 0	AC	0.0	1.0	0.0667	1069616302655
	Subsystem 0	AC	0.0	1.0	0.0667	1069616485748
	MyNewsystem	Reliability	0.0	1.0	0.6793	1069616699452
	MyNewsystem	AC	0.0	1.0	0.0667	1069616699483
	*					

Figure 22 Result Table Design View

- Or copy the *measurement.mdb* from this Major Project folder to the hard disks on your PC.
- Registering the *measureasure.mdb* as an **ODBC Data Source**. The Data Source name should be “**MeasurementTools**”, which is hard code in source code too.

- Run the batch file **RunMeasurement.bat** in the major project folder. The Figure 13 will display. Select the measurements you want to calculate and choose the appropriated input files.
- For the Architectural complexity, the input file is system configuration specification, which produced by the TROMLAB system. For Reliability, the input system is GRC specification files; synchronous product machine specification file and system configuration file. The GRC specification files could be multi-selection.
- If Reliability is checked, the SCS file in AC is not needed to select again.
- Press the **OK** button, the measurement results will display in a text format (Figure 14).
- By press **Graphic Display** button user could view graphic results. In this view, user could reset the upper bound range of the measurement result by selecting the **Set upper bound check box**. The edit boxes and **Update** button will be enabled. By set the values in the edit boxes, and press the **Update** button, the new upper bound values will reset in bottom table and the graphic.
- By press the **Database** view button user could view measurement result in database, and the table could be saved as Microsoft Excel file. The file could be viewed and edit by using Microsoft Excel.

## Chapter 6 Conclusions & Future Work

This project is design and implementation of complexity algorithms and maintainability profile (TROM-SCMS) and integration of the complexity and reliability under a common user interface (TROM-QM). TROM-QM and TROM-SCMS are working in TROMLAB environment. The theories on the architectural complexity computation and the main algorithms used for calculation, as well as the integration of the measurement modules are presented in this Major Report. A case study has been introduced as an example to show the complexity calculation algorithm. This Major Report also describes in detail the TROM-QM user interface design and implementation, and introduces how to integrate the other modules into this system.

The future work should consider the following enhancements:

- The testability and functionality should be integrated to TROM-QM.
- The TROM-QM is using the output files from TROMLAB Translator Tool and the Simulator Tool. TROM-QM should be integrated to TROMLAB.
- We assume the input files format is correct, so the error checking and handling are not sufficiently implemented.
- The statistic data analysis for the measurement results should be implemented when there is enough measurement data collected.

## References

- [AAM98] V.S.Alagar, R.Achuthan, D.Muthiayen. *TROMLIB: A Software Development Environment for Real-Time Reactive Systems*. (first version 1996, revised 2001), Submitted for Publication
- [Ach95] R. Achuthan, *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems*. PhD. thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1995
- [Che02] M. Chen. *The Implementation of Specification-based Testing System for Real-time Reactive System in TROMLIB Framework*. Master Major Report, Department of Computer Science, Concordia University, Montreal, Canada, December 2002
- [GH93] J.V. Guttag and J.J. Horning. *Larch: Language and Tools for Formal Specifications*. Springer Verlag. 1993.
- [Hai99] G. Haidar. *Reasoning System for Real-Time Reactive Systems*. Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999
- [HP85] D. Harel, A. Pnueli. *On the Development of Reactive Systems*. In Logic and Models of Concurrent Systems, NATO, Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems Spring Verlag, 1985
- [KKKC96] Kim E., usumoto S., Kikuno T., Chang O. *Heuristics for Computing Attribute Values of C++ Program Complexity Metrics*. IEEE Transactions on Software Engineering, 104-109, 1996

[Lee03] F.A Lee Reliability Measurement Based on the Markov Model for Real-time Reactive Systems: Design and Implementation

[Mut96] D.Muthiayen ***Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment***. Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, October 1996

[Nag99] D.Muthiayen ***Real-Time Reactive System Developemnt – A Formal approach based on UML and PVS***. PhD Thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2000

[Orm02] O. Ormandjieva, ***Deriving New Measurements for Real-Time Reactive Systems***. PhD. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2002

[Pom99] F. Pompeo ***A Formal Verification Assistant for TROMLIB environment***. Master Thesis, Department of Computer Science, Concordia University, Montreial, Canada, November 1999

[Pop99] O. Popistas. Rose-GRC Translator: ***Mapping UML Visual Models onto Formal Specifications***. Master Thesis, Department of Computer Science, Concordia University, Montreial, Canada, April 1999

[Sir99] V. Srinivasan. ***Graphical User Interface for TROMLIB Environment***. Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.



[Tao96] H. Tao. Static Analyzer: ***A Design Tool for TROM***. Master Thesis,  
Department of Computer Science, Concordia University, Montreal, Canada,  
August 1996

[Zhe02] M.Zheng. ***Automated Generation of Test Suits from Formal Specifications of Real-Time Reactive Systems***. Ph.D. Thesis, Department of  
Computer Science, Concordia University, Montreal, Canada, 2002.