INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

TEST ADEQUACY MEASUREMENT FOR REAL-TIME REACTIVE SYSTEM

REN WEI HE

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

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

APRIL 2003

Abstract

Test Adequacy Measurement for Real-time Reactive System

Ren Wei He

Concordia University, 2003

In the context of safety-critical systems, which are real-time reactive systems, testing process process must be integrated with the software development process as a whole, not just with the end product. Consequently, measuring the efficiency of the testing process emerges as an important issue. To be accurate and meaningful, both the measurement and the components to be measured must be precisely described. The measurement should conform to the axioms of measurement theory, and the software development process should be grounded on rigorous formalism. In this paper we introduce test adequacy measurement for measuring the efficiency of testing process integrated into a rigorous environment for developing real-time reactive systems. The test data adequacy measurement associates a degree of test set adequacy according to the test data adequacy criterion to indicate how adequately the testing has been performed. There are two important measures associated with every test data adequacy criterion and minimum number of test cases. The measurement method is illustrated with Train-Gate-Controller design, a bench-mark example in real-time system design.

To my family.

Acknowledgments

I would like to thank my supervisor Dr. Olga Ormandjieva. She guided me throughout my study at Concordia University and played a significant role in shaping my major report. Without her wisdom, her unfailing guidance, constant encouragement and insightful comments, this work would not have been possible. I am really fortunate to have had such a great supervisor to go through this work.

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

On a personal level, I thank my wife Xiao Yun Weng for her support and help, and my son and my pretty baby for their patience, understanding and love.

Contents

List of Figures

1 Introduction.....1

1.1 Real-time Reactive System.....1

1.2 Test Adequacy Measurement.....2

1.3 Case Study Description.....3

2 Test Adequacy Measurement for Real-time Reactive system.....6

2.1 Introduction.....6

2.2 Background.....7

2.2.1 Test Case Adequacy Measurement.....7

2.2.2 Formal Foundation.....9

2.3 Metric-Based Approach.....11

2.3.1 Formal Representation and Abstraction of the Test Cases Domain.....12

2.4 Testing Distance Measurement.....13

2.5 Metric-based Test Set Selection.....16

3 System Design.....18

3.1 Introduction.....18

3.2 Architure Diagram.....18

3.3 Filter Subsystem.....20

3.3.1 Introduction.....17

3.3.2 Classes of the Testcases Filter.....21

3.3.3 Sample Running.....27

3.4 Selector Subsystem.....28

3.4.1 Basic Classes of the Testcases Selector.....30

3.4.2 Sample Running.....35

4 Conclusions..... 41

References.....42

Appendix A: Java Printing Selector.....44

Appendix B: Java Printing Test.....46

Appendix C: Java Printing TestCase.....51

Appendix D: Java Printing File.....53

List of Figures

Figure 1: State Diagram of Class Train.....4

Figure 2: State Diagram of Class Controller.....5

Figure 3: State Diagram of ClassGate.....5

Figure 4: Array Representation of Test Cases.....15

Figure 5: System Architecture Diagram of Test Adequacy Measurement.....18

Figure 6: Original Testcases.....21

Figure 7: Classes of the Testcases Filter.....22

Figure 8: Binary Testcases From the Filter.....28

Figure 9: Selector Class Diagram.....31

1 Introduction

1.1 Real-Time Reactive System

The distinctive feature of a reactive system is the continuous interaction between the system and its environment. The system receives and sends messages through a hardware interface consisting of sensors and actuators, giving rise to stimulus-response behavior. The sequence of intervations depends on several factors, the most influential being the level of coupling between the entities in the environment. In the case of real-time reactive systems, stimulus-response behavior is regulated by timeing constraints. Alarm systems, air traffic control systems, nuclear reactor cotrol systems, railroad crossings and telecommunication systems are typical examples of safety-critical systems involving concurrency and synchronous communicaton between actuators, reactors and reactive entities. Common to all these applications is the notion of reactive behavior, wherein the relationshop between input and output over time, complex sequencing of events and the way they constrain the computations are described. The term reactive was introduced by Harel and Pnueli [HP85] to designate systems that continuously interact with their environment and to distinguish them from the interactive and transformational systems. Two important properties characterize real-time reactive systmes:

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

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. Its quality is essential for the economics of the software development and the reliability of the final product. In order 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 technologies, and provides a formal basis for specification, analysis and refinements of the real-time reactive systems design.

1.2 Test Adequacy Measurement

In software industrial practice, the high cost of development process of large-scale software has put emphasis on the need to prevent problem occurrence, rather than fix errors during the later phases of a software life cycle. The goal of test adequacy measurement is to quantitatively assess software-testing efficiency [AO2000]. We focus on the measurement of the test adequacy. In order to achieve this, the measurement of the test adequacy must be based on the theory of software measurement, and the components and development processes to be measured must be precisely described on the basis of a rigorous formalism. The formal foundation of test adequacy measurement is based on the operational semantics in TROM theory. We illustrate our model and testing measurement on a Train-Gate-Controller example, a bench-mark case study in real-time research community. A tool for automatic gathering of testing measurement data and analyzing it is being implemented as part of TROMLAB [AAM98], a real-time reactive systems development environment.

1.3 Case Study Description

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

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

Figures 1, 2, 3 show the state chart diagrams for the controller, train and gate. Together, they specify the behavior of a Train-Gate-Controller system. When there is no train in the system, the train is in the state "idle", the controller is also in the state "idle", and the gate is in the state "opened". When a train wants to pass through a crossing, it

sends the message to the controller; the train and controller change their states simultaneously. In the state "activate", the controller may receive "Near" messages from other trains or it sends the message "lower" to the gate. In the latter case, the controller and gate change their states simultaneously.

In the state "monitor", the controller continues to receive "Near" messages from other trains or monitor the gate. The train that is in the crossing state may send an "Exit" message to the controller before exiting the gate. Both the controller and the train synchronize on the event "Exit". The train leaves the gate within 6 unit of time from the moment it sends the message "Near" to the controller. The controller changes from state "monitor" only when all the trains have exited from the gate. At that instant, it sends the message to the gate and returns to the state "idle".

The events "Down" and "Up" are time constrained internal events for the gate. The evaluation for the local clocks for the controller is the variables "TCvar1" and "TCvar2". Similarly, their local clocks shown in the extended state chart diagrams govern the time-constrained events for the controller and train.

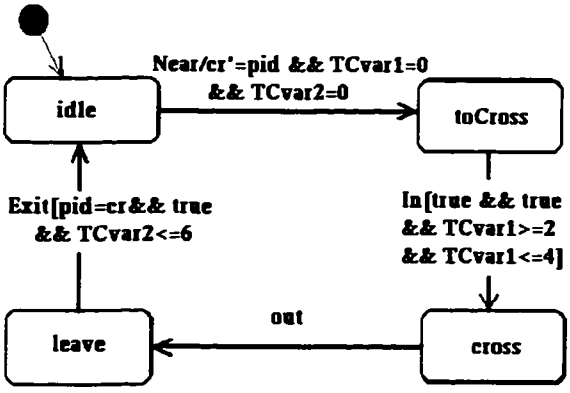

Figure 1: State Diagram of Class Train

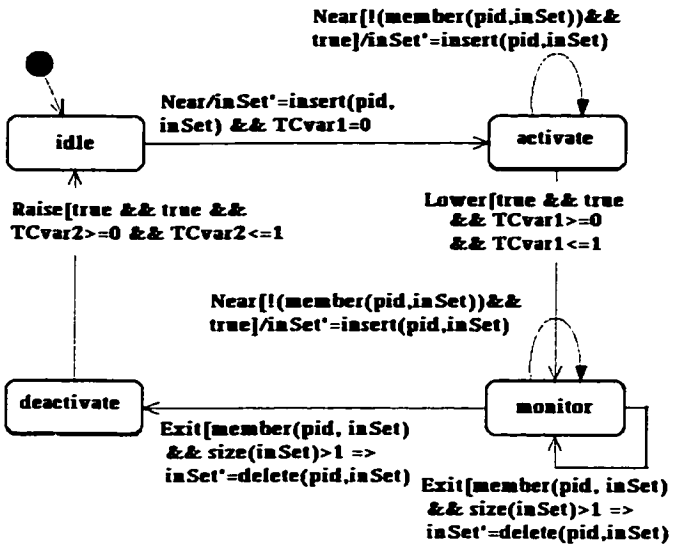

Figure 2: State Diagram of Class Controller

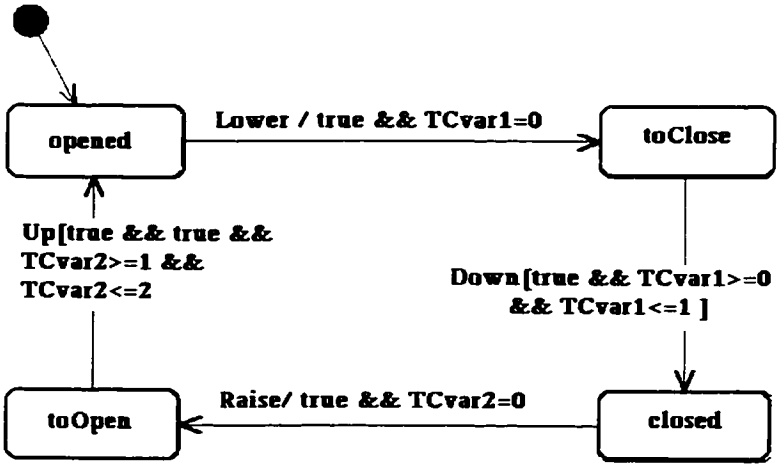

Figure 3: State Diagram of Class Gate

2 Test Adequacy Measurement for Real-Time Reactive System

2.1 Introduction

In the context of safety-critical systems, which are real-time reactive systems, testing process must be integrated with the software development process as a whole, not just with the end product. Consequently, measuring the efficiency of the testing process emerges as an important issue. To be accurate and meaningful, both the measurement and the components to be measured must be precisely described. In this paper we introduce test adequacy measurement for measuring the efficiency of testing process integrated into a rigorous environment for developing real-time reactive systems.

In order to ensure the correctness of the implementation with respect to design, the implementation of the system has to be tested. Testing efficiency can be increased by defining an adequate minimal standards for testing, represented by test adequacy criterion that defines what constitutes an adequate test. Informally, the test adequacy criterion is defined as "a predicate that defines software testing objectives in terms that can be measured to guarantee the correctness of a tested program". The degree of adequacy of testing related to the test adequacy criterion is estimated by adequacy measurement [W86] . Test adequacy criterion plays two essential roles in any testing method: to specify testing requirements, and to determine the observations that should be done during the testing process. The testing requirements specification has two forms. The first form is called test case selection criterion and is an explicit specification for test case selection. The second form is an explicit specification for test set adequacy measurement

when a degree of adequacy in terms of test coverage is associated with each test suite, namely test data adequacy criterion.

To generate a set of test cases from the software product and its own specification, a testing method should be defined using a test case selection criterion. The level of test case adequacy, which is the degree to which the software is tested, is to be evaluated as well. The degree of adequacy of testing related to the test adequacy criterion is estimated by the test adequacy measurement. The measurement of the quality of coverage of the test suite would increase (or decrease) the confidence in tested components. A representational theory of test adequacy measurement and an axiom system to study the properties of test adequacy measures have been proposed and proved to be consistent for test adequacy measurement [W86]. There are two central points when developing a solution for test adequacy measurement: the choice of a formal representation of the test cases domain, and a model of its abstraction. We have chosen to represent formally the test cases domain as a metric space. This approach would allow the use of metric-based test case selection algorithm to select the optimal set of test cases (from the test selection domain), and metric-based coverage evaluation measurement, based on testing distance measure. The formal representation of the test cases domain, a model of its abstraction and the definition of a testing distance are presented.

2.2 Background

2.2.1 Test Case Adequacy Measurement

The *test data adequacy measurement* associates a degree of test set adequacy according to the *test data adequacy criterion* to indicate how adequately the testing has been

performed. The test data adequacy measurement is related only to the test data adequacy criterion and in particular, to the *stopping rule*.

There are two important measures associated with every test data adequacy criterion [FP97] : *test effectiveness ratio* (degree of adequacy of testing) and *minimum number of test cases*. The testing effectiveness is defined as $TE = F/E$, where F is the number of faults discovered and E is the effort measured as "person month". To be able to predict the testing time and resources, the testers would need to know the minimum number of test cases needed to satisfy the test data adequacy criterion for the given software. The definition of a measurement formula of calculating the minimum number of test cases depends on each testing approach. A method based on *Prime Decomposition Theorem* is given by Fenton [FP97] to calculate a *minimum number of test cases measurements* for the structural testing approach. The problem of determining *minimum number of test cases measurement* for error-based and fault-based testing approaches still remains an open problem.

The *test effectiveness ratio measure* or equivalently, *a degree of adequacy of testing* of a program p by a test set t with respect to the specification s, according to the test adequacy criterion C, is a function $Mc(p,s,t)$ with value in the interval. To determine whether or not sufficient testing has been done, the test data adequacy criterion as *stopping rule* (or *predicate rule*) should be considered:

> Given a degree R of adequacy corresponding to the minimum number of test cases r, the stopping rule M_R is True if and only if the adequacy degree is greater than or equal to r, or false otherwise:

$$M_R(p, s, t) = \begin{cases} True, & Iff M(p, s, t) \geq r \\ False, & otherwise. \end{cases}$$

The measurement formula of the test effectiveness ratio measure M depends on the specific testing approach and the corresponding test data adequacy criterion. The higher degree of adequacy M indicates more adequacy testing of p with respect to s, according to C. Zhu et all [ZHM97] point out that M depends on the specific testing approach and the corresponding test data adequacy criterion and discusses three approaches to software testing in this context.

2.2.2 Formal Foundation

In this section we discuss a representational theory of test adequacy measurement and an axiom system to study the properties of test adequacy measures. From **Theory of** Weyuker's Axioms [W86], has proved the consistency of these axioms. Analytical evaluation of testing techniques considers whether the testing criteria meet adequacy axioms.

Let

P = program under test

S = specification of ***P***

D = input domain of ***S*** and ***P***

T = subset of ***D***, used as test set for ***P***

C = test adequacy criterion

- ***P*** is incorrect if it is *inconsistent* with ***S*** on some element of ***D***

- ***T*** is unsuccessful if there exists an element of *T* on which *P* is *incorrect*
- ***A*** test adequacy criterion *C* is *subdomain-based* if it induces one or more subsets, or *subdomains*, of the input domain *D*
- ***A*** subdomain-based criterion *C* typically does not *partition D* (into a set of non-overlapping subdomains whose union is *D*)

Theory: Weyuker's Axioms (1986)

Axiom 1. Applicability

For every ***P***, there is a finite adequate ***T***

Axiom 2. Nonexhaustive applicability

For at least one ***P***, there is a non-exhaustive adequate ***T***

Axiom 3. Monotonicity

If ***T*** is adequate for ***P*** and $T \subseteq T'$, then ***T'*** is adequate for ***P***

Axiom 4. Inadequate Empty Set

The empty set is not adequate for any ***P***

Axiom 5. Antiextensionality

There are programs ***P1*** and ***P2*** such that ***P1*** = ***P2*** and ***T*** is adequate for ***P1*** but not ***P2***

Axiom 6. General Multiple Change

There are programs ***P1*** and ***P2*** such that ***P2*** can be transformed into ***P1*** and ***T*** is adequate for ***P1*** but not ***P2***

Axiom 7. Antidecomposition

There is a program ***P*** with component ***Q*** such that ***T*** is adequate for ***P***, but the subset of ***T*** that tests ***Q*** is not adequate for ***Q***

Axiom 8. Anticomposition

There are programs ***P1*** and ***P2*** such that ***T*** isadeuqate for ***P1*** and ***P2*** ***(T)*** is a adequate for ***P2*** but ***T*** is not adequate for ***P1 ;P2***

2.3 Metric-Based Approach

There are two central points when developing a solution for test adequacy measurement: the choice of a formal representation of the domain of the test cases, and a model of its abstraction. We have chosen to represent formally the test cases domain as a metric space. A metric space is a pair (V, td), where V is a non-empty set and td is a *distance*, or *metric*, such that $td: V \times V \rightarrow R^*$ and the set of distance axioms are satisfied. This approach allows the use of metric-based test case selection algorithm to select the minimal set of test cases (from the set of automatically generated test cases), and metric-based coverage evaluation measurement, both based on the notion of distance between test cases.

$\forall X, y \in V$ the following axioms are satisfied:

Axiom 1. $td(x; y) \geq 0$;

Axiom 2. $td(x; y) = 0 \Leftrightarrow x = y$;

Axiom 3. $td(x; y) = td(y; x)$;

Axiom 4. $td(x; z) \leq td(x; y) + td(y; z)$.

The metric is unique in the sense that there is an order-preserving transformation between two metrics.

2.3.1 Formal Representation and Abstraction of the Test Cases Domain

The test case domain is the set of symbols and terms in the formal specification used to specify the system. The algorithms discussed by Zheng [Zhe02] compute the test cases as follows:

> For a reactive unit A, the set $T_d(A)$ of test cases is computed as
>
> $X_d(A) \cup Y_d(A)$,
>
> Where $X_d(A)$, and $Y_d(A)$ are respectively the state and transition covers of the grid automata $G_d(A)$ with grid size $d = 1/k$, k being the number of clocks in A, a state cover for a state θ in the grid automation is a labeled path from the initial state of $G_d(A)$ to θ. The sequence of labels in a path is the events that take the grid automation from its initial state to the state θ. An event in the grid automation $G_d(A)$ is either an event of A or d. The label d for a transition from the state θ to a state θ' in $G_d(A)$ indicates the passage of time at the state s of A, where both θ, θ' are mapped to s by the construction of the grid automation. As an example, the sequence
>
> *Near?, 1/2, 1/2, In*
>
> may denote a state cover for the grid automation of the **Train** class.

The formal representation that we discuss for developing metrics for test cases is independent of such concrete representations.

In general, let *STC* denote any test set with any arbitrary representation of test cases in it. Our approach consists in abstracting the elements of *STC* as binary strings. This

would allow the introduction of testing distance as information distance in the space of binary strings. Our choice of information distance is justified by the fact that it is an absolute and objective quantification of a distance between individual objects [Be98].

A two-dimensional array *TCA* represents the mapping of a test case into a binary string. The definition of the array *TCA* is as follows:

$$TCA(a, j) = \begin{cases} 1, & \text{if test case } a \text{ contains event } j \\ 0, & \text{otherwise.} \end{cases}$$

Each row of TCA is a mapping of a test case into a binary string. The creation of the array *TCA* reflects the order of appearance of the events in the test case.

2.4 Testing Distance Measurement

Any distance measurement should satisfy the symmetric and triangle properties for a distance. Intuitively, we expect more similarity between test cases when the distance between the two test cases is small. We want to select test cases for test execution from a test set so that the distance between the selected test case and the set of already exercised test cases is not small.

The distance between two test cases $A, B \in STC$ is abstracted as a distance between their binary string representations $a, b \in V$. The distance between $a, b \in STC$ would depend on two factors, namely, the similarity and the dissimilarity between the test cases. Thus we define the testing distance as:

$$Td(a, b) = similarity(a, b) * dissimilarity(a, b)$$

Where *similarity (a, b)* is defined in terms of the longest common prefix of *a* and *b*, and *dissimilarity (a, b)* is expressed in terms of the minimum amount of change necessary to convert the binary string *a* into *b*. The formal quantification models are given below.

Similarity Quantification

Let *LCP(a, b)* be the longest common prefix of the binary string representations *a, b* $\in V$ *of A.* $B \in STC$. We define similarity factor between strings as

$$similarity\ (a, b) = 2^{-length(LCP(a, b))}$$

Note that *LCP (a, b)* is 0 when there is no common prefix, and

$$min\ (length\ (a);length\ (b))$$

When the longest common prefix coincides with one of the strings. The range of the similarity is between 0 and 1. Higher values indicate lower level of similarity between two test cases and diminish the value of a testing distance. The information distance between tow binary strings (elements of a metric space) is computed as the length of the shortest program that translates one string into another.

Dissimilarity Quantification

The dissimilarity measure between two binary strings a and b are calculated as the number of elementary transformations that are minimally needed to transform the string *(a\LCP(a, b))* into the string *(b\LCP(a, b))*. Let us suppose that the abstraction $a \in T$ of some test case $A \in STC$ is the row *a* of the array *TCA*. The set of elementary transformations are (1) adding an event *j* (i.e., setting the value of *TCA(a;j)* to 1), and (2) removing an event *j* (i.e., setting the value of *TCA(a; j)* to 0). The dissimilarity is a

unidimensional spatial proximity measure, defined on the ordinal scale. It satisfies the representation and uniqueness conditions for the unidimensional ordinal scale measures and thus is theoretically valid.

Illustration of Testing Distance Measurement

We illustrate the quantification of the distance between two test cases generated [Zhe02] for the Controller-Gate Subsystem. Consider the two test cases $A,B \in STC$, A = *{Near?.Lower.Down.Exit?.Raise.1/4.1/4.1/4.1/4.Up}*, and B = *{Near?.Lower.1/4.1/4.1/4.1/4.Exit}*. The array representation of the test cases is shown in Figure 4. In this case the values of the model variables are as follows:

$LCP(a,b) = 2$; similarity$(a,b) = 2^{-2}$; dissimilarity$(a,b) = 5$; $td(a,b) = 5/4$.

Event / Testcases	Near?	Lower	Down	Exit?	Raise	1/4	1/4	1/4	1/4	Up	Exit
a	1	1	1	1	1	1	1	1	1	1	0
b	1	1	0	0	0	1	1	1	1	0	1

Figure 4: Array Representation of Test Cases

From *td* (*a*, *b*), we know that if test cases *a*, *b* have more similarly, the distance between them is closer, that mean *td* (*a*, *b*) is small. Otherwise, if test cases *a*, *b* have more dissimilarly, the distance should be larger. In a word, *td* (*a*, *b*) is less, and then test case a, b have more similarly. Otherwise, a, b have more dissimilarly.

2.5 Metric-Based Test Set Selection

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

Test Selection Algorithm

Precondition: $\{V = V \neq \phi \wedge \varepsilon \min > 0 \wedge C = C \wedge A = \phi\}$

Step 1. Initialization (A,V, ε)

Step 2. Create – Test – Set(A, V, ε)

Postcondition: $\{A \neq \phi \wedge (Cost(A) \geq C \vee \varepsilon < \varepsilon_{min} \vee V = \phi)\}$

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

Precondition: $\{V = V \wedge A = \phi\}$

Step 1. t = Longest – test – Case(V);

Step 2. Add(A,t);

Step 3. Remove(V, t);

Step 4. ε = Length(t) – 1;

Step 5. **IF** $\varepsilon <= 0$ **THEN**

$\varepsilon = \varepsilon_{min}$;

ENDIF;

Step 6. $V = 0$;

$V = A$;

$A = 0$;

Postcondition: $\{ V \neq \phi \wedge A = 0 \wedge \varepsilon > 0 \}$

Algorithm for *Create – Test – Set (A, V,* ε *,* ε *min)*

Precondition: $\{ V \neq \phi \wedge A = 0 \wedge \varepsilon > \varepsilon_{min} \}$

While

$\neg (Cost(A) \geq C \vee \varepsilon < \varepsilon_{min} \vee V = \phi)$

IF ($\exists$ *test case t: td (t, V) >=* ε)

Then *Add (A, t); Remove (V, t);*

ENDIF;

$\varepsilon = \varepsilon - 1$;

ENDWHILE;

Postcondition: $\{A \neq \phi \wedge (Cost(A) \geq C \vee \varepsilon < \varepsilon_{min} \vee V = \phi)\}$

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

3 System Design

3.1 Introduction

The project for real-time reactive systems in TROMLAB environment (TROM), will be developed in Java whose advantage of portability is taken into consideration. Therefore, the java swing and JDK have to be used. The required minimum Java version is 1.2. The following introduces the implementation of the system, which is implemented in Java programming language.

3.2 Architecture Diagram

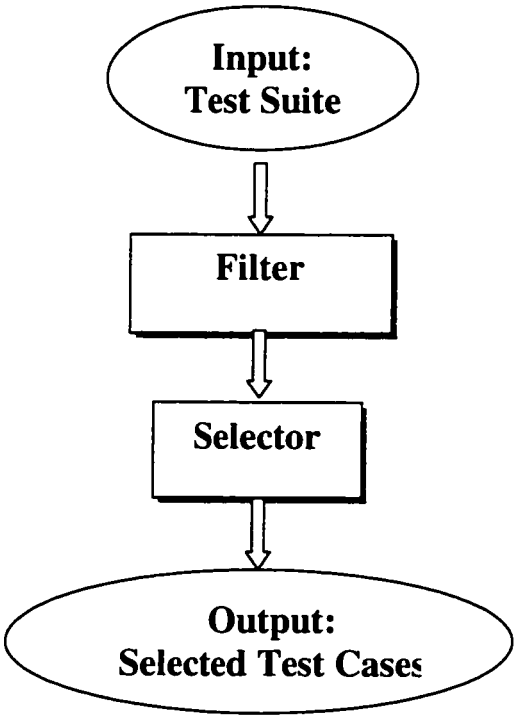

Figure 5: System Architecture Diagram of Test Adequacy Measurement

Figure 5 shows the system architecture diagram and the subsystems of Test Adequacy Measurement. The chosen architectural pattern is Pipe-and-Filter. The input to the system is the test suite generated by the Automotive test cases generator module, part of the TROMLAB environment. The test case generation algorithm produces a minimal set that

can exhaustively test the implementation for all specified interacting properties. Example of input data is shown below:

<<T.toCross, G_C.<G.opened, C.activate>>+0/6> : T/G_C.Near

<<T.toCross, G_C.<G.opened, C.activate>>+2/6> : T/G_C.Near, 1/6, 1/6

<<T.toCross, G_C.<G.opened, C.activate>>+3/6> : T/G_C.Near, 1/6, 1/6, 1/6

<<T.toCross, G_C.<G.opened, C.activate>>+6/6> : T/G_C.Near, 1/6, 1/6, 1/6, 1/6, 1/6, 1/6

Filter: The input of the filter is the original test cases for class testing and system testing. In order to get the reduced set of test cases we have to transform the original test cases to binary strings representing the standard set of test cases. Filter is a subsystem that filters that transforms the original test cases into binary test cases. The input of the filter is the original test cases for class testing and system testing, for example:

<<G.opened, C.activate>+3/4> : C.Near, 1/4, 1/4, 1/4

<<G.toOpen, C.idle>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise

The output of the filter is the set of binary test cases, each of them represent one case that has many events, for example:

1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

this is the set of binary representations of test cases that is passed as input to the selector subsystem.

Selector: This subsystem is the major work of this project. In order to select a sufficient number of tests from a given collection of test cases, the subsystem selector has to be applied in order to select an optimal set of test cases. This optimization would reduce the

cost of the testing process while maintaining the same level of efficiency. The input of the selector is the binary representation of a collection of test cases. The output of selector is a representative subset of test cases.

3.3 Filter Subsystem

3.3.1 Introduction

In this section, we use the original test cases from the railroad-crossing problem. A generalized version of this problem has been considered by Muthiayen and Alagar [AAM98] to formally prove safety properties in their design. We take their verified design and generate test cases as our original test cases and transform it to binary test cases. Mr. Chen introduces some general algorithms and their implementation [Che03]. These algorithms are used in unit testing, pair testing and system testing. In unit testing, after Spec_Parser generates an initial TROM object from class specification file, the algorithm GA (Grid Automaton Generation) to generate a grid automaton from the original TROM, in which its time constraints are decomposed, and then test case generation algorithm to product minimal set that can exhaustively test the implementation for all specified interacting properties. An example of the original testcases generated by the above algorithms is shown below:

<<G.opened, C.activate>+0/4> : C.Near
<<G.toClose, C.monitor>+0/4> : C.Near, G/C.Lower
<<G.opened, C.activate>+1/4> : C.Near, 1/4
<<G.closed, C.monitor>> : C.Near, G/C.Lower, G.Down
<<G.toClose, C.monitor>+1/4> : C.Near, G/C.Lower, 1/4
<<G.opened, C.activate>+2/4> : C.Near, 1/4, 1/4
<<G.closed, C.deactivate>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit
<<G.toClose, C.monitor>+2/4> : C.Near, G/C.Lower, 1/4, 1/4
<<G.opened, C.activate>+3/4> : C.Near, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise
<<G.closed, C.deactivate>+1/4> : C.Near, G/C.Lower, G.Down, C.Exit, 1/4
<<G.toClose, C.monitor>+3/4> : C.Near, G/C.Lower, 1/4, 1/4, 1/4
<<G.opened, C.activate>+4/4> : C.Near, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+1/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4
<<G.closed, C.deactivate>+2/4> : C.Near, G/C.Lower, G.Down, C.Exit, 1/4, 1/4
<<G.toClose, C.monitor>+4/4> : C.Near, G/C.Lower, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+2/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4
<<G.closed, C.deactivate>+3/4> : C.Near, G/C.Lower, G.Down, C.Exit, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+3/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4
<<G.closed, C.deactivate>+4/4> : C.Near, G/C.Lower, G.Down, C.Exit, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+4/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+5/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+6/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+7/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4
<<G.toOpen, C.idle>+8/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, ¼

Figure 6: the Original Testcases

3.3.2 Classes of the Testcases Filter

To filter the original test cases to binary test cases the main classes of the test cases filter are designed: filter, caselistbuilder, testcasebuilder, binarycasebuilder and caslist. The class diagram is shown in the following:

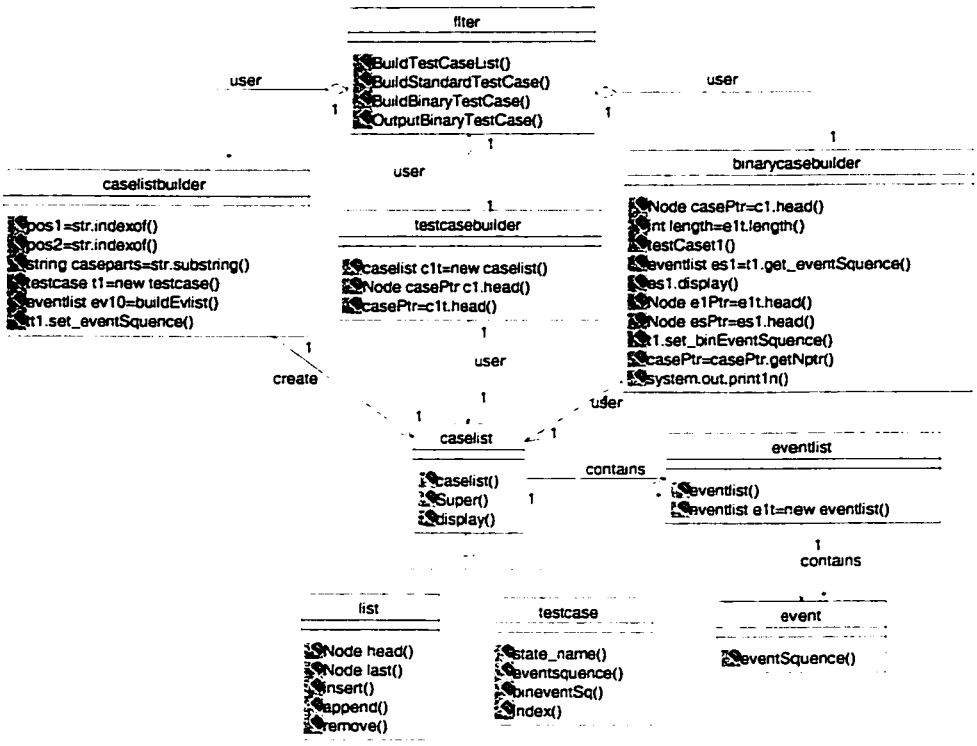

Figure 7: Classes of the Testcases Filter

1. Class: filter --- This is main program that performs the filter's function. There are four main function: buildTestCaseList(), buildStandardTestCase(), buildBinaryTestcase() and output the binary test cases to "cases.txt".

==========

Pseudo Code of filter:

```
public transform(String inStr)
        {
                index=1;//standardEventsequence=new String();
                cl=new caselist();
                buildTestCaseList(inStr);
                elt=new eventlist();
                buildStandardTestCase();
                System.out.println(n);
                buildBinaryTestCase();
        }
```

2. Class: caselistbuilder --- This program reads the strings from the original text file and separate them create the notes that express the each of test cases with index number. In caselistbuilder, there are three subclasses: class list, testcase class and eventlist class. In class list, there are some list operation inside such as insert(), append() and remove(). In class testcase, we definite the format of testcase(node), for example, one test case is one node that include state_name(), eventsequence(), index() and so on. In eventlist, we list the events of test case for caselist. After caselist, we form a sequence of nodes that will be used for testcasebuilder and binarycasebuilder.

Pseudo Code of caselistbuilder:

```
private void buildTestCaseList(String str)
        {
                int pos1=str.indexOf();
                int pos2=str.indexOf();
                String caseparts=str.substring();
                while(caseparts.indexOf(":")!=(-1))
                {
                        pos1=caseparts.indexOf(":");
                  pos2=caseparts.indexOf();
                        String temp;
                        if(pos2!=(-1))temp=caseparts.substring();
                        else temp=caseparts;
                        testcase t1=new testcase();
                        t1.set_index(index++);
                        t1.set_state_name(temp.substring(0,temp.indexOf(":")));
                        String evs=temp.substring(temp.indexOf(":")+1);
                        eventlist evl0=buildEvList();
                        //evl0.display();
                        t1.set_EventSquence();
                        t1.set_signal(false);
                        cl.append(t1);
                        if(pos2!=(-1))
                        {
                        caseparts=caseparts.substring(pos2+1);
                        }
                        else
                        {
                        caseparts="";
                        }
                }
        }
```

3. Class: testCasebuilder --- After caselistbuilder we use the caselist(nodes list), we use these nodes to create one tree and at the same time we get the standard testcase from this tree. First at all, we search the nodes that only have one event. If the nodes only have one event and do not have children, we put the event of nodes to sequence of standard test case. Secondary, If the nodes have one event and also have child or children, we search the children as first step if they have only one event. After that, we already search the all test cases that only have one event and put it as part of standard testcase. After that we redo the first step and secondary step to search the nodes that only have two events and so on. So, we search the whole testcaselist(nodes) and form the standard test case. For example, if the test case list is in Figure 6:

we can get the below standard case that can be matched each of the test cases in test case list.

Standard test case is below:

C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

==========

Pseudo Code of testCasebuilder:

```
private void buildStandardTestCase()
        {
                caselist clt=new caselist();
                Node casePtr=cl.head();
                while(casePtr!=null)
                {
                        testcase t1=(testcase)casePtr.getNodeVal();
                        if(t1.get_EventSquence().length()==1)
                        {
                                testcase t2=new testcase();
                                t2=t1;
```

```
                    clt.insert(t2);
                    t1.set_signal(true);
                }
                casePtr=casePtr.getNptr();
            }
            casePtr=clt.head();
            while(casePtr!=null)
            {
                testcase t1=(testcase)casePtr.getNodeVal();
                eventlist t1el=t1.get_EventSquence();
                Node n1=(Node)t1el.last();
                String s1=(String)n1.getNodeVal();
                s1=s1.trim();
                elt.append(s1);
                Node cPtr=cl.head();
                String evla;
                while(cPtr!=null)
                {
                    testcase t2=(testcase)cPtr.getNodeVal();
                    if((!(t2.get_signal()))&&(isNextNode(t1el,t2)))
                    {
                        t2.set_signal(true);
                        Node n2=(Node)t2.get_EventSquence().last();
                        evla=(String)n2.getNodeVal();
                        evla=evla.trim();
                        elt.append(evla);
                        Node casePtr2=cl.head();
                        while(casePtr2!=null)
                        {
                            testcase t3=(testcase)casePtr2.getNodeVal();
                            if((!(t3.get_signal()))&&
                                (isNextNode(t1el,t3)))
                            {
                                System.out.println();
                                t3.display();
                                testcase t4=new testcase();
                                t4=t3;
                                clt.insert(t4);
                                t3.set_signal();
                            }
                            casePtr2=casePtr2.getNptr();
                        }
                        t1el=t2.get_EventSquence();
                    }
                    cPtr=cPtr.getNptr();
                }
                clt.remove(t1);
                casePtr=clt.head();
            }
        }
```

4. Class: binarycasebuilder --- After testcasebuilder, we get standard test case from testcasebuilder and the whole test cases list from caselist, so in binarycasebuilder class,

we use the standard test to match each of test cases and form the binary test cases. For example, we get the standard test case below:

Standard test case:

C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, ¼

The two test cases from case list:

C.Near, G/C.Lower, ¼

C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

After binary casebuilder, we get binary test case below:

1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

==========

Pseudo Code of binarycasebuilder:

```
private void buildBinaryTestCase()
        {
                Node casePtr=cl.head();
                int length=elt.length();
                while(casePtr!=null)
                {
                        testcase t1=(testcase)casePtr.getNodeVal();
                        eventlist es1=t1.get_EventSquence();
                        es1.display(System.out);
                        Node elPtr=elt.head();
                        Node esPtr=es1.head();
                        String stemp="";
                        while(elPtr!=null)
                        {
                                String s1=(String)elPtr.getNodeVal();
                                s1=s1.trim();
                                String s2;
                                if(esPtr!=null)
                                {
                                        s2=(String)esPtr.getNodeVal();
                                        s2=s2.trim();
                                        if(s1.equals(s2))
                                        {
                                                stemp=stemp.concat("1 ");
                                                esPtr=esPtr.getNptr();
```

```
                    }
                    else stemp=stemp.concat("0 ");
                }
                else stemp=stemp.concat("0 ");
                elPtr=elPtr.getNptr();
            }
            t1.set_binEventSquence(stemp);
            System.out.println("="+stemp);
            casePtr=casePtr.getNptr();
        }
    }
```

3.3.3 Sample Running

3.3.3.1 Original Test Cases

The input to the system is the test suite generated by the automotive test cases generator module. In the sample running the set of original test cases has 25 cases each of them has 25 events in Figure 6 as input of filter.

3.3.3.2 Binary Representations of Test Cases

The input of the filter is the original test cases for system testing. After filter subsystem, the original test cases have been transformed to the binary strings representation of test cases. For example, the above the set of original test cases is changed to binary representation below:

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 8: Binary Testcases From the Filter

3.4 Selector Subsystem

In this section, we focus the selector subsystem that addresses the design issues of the case selector algorithm. By definition, a case is a binary vector specifying the order of sequence of tests to be performed.

The objective of the implementation is to use Java to develop a small software, named, case selector. The principle of selector's functionality is as follows: select a representative subset of test cases from a given collection of test cases under two given testing cost constraints:

(i) testing cost constraint--- the total testing cost should be less than a given value, denoted by **C**.

(ii) distance constraint--- the distances between any two selected cases should not be less than a given constant value, denoted by $\varepsilon_{\_min}$. So, for any two given test cases *case1* and *case2* we need define their distance, denoted by dist(*case1*, *case2*), whose definition will given later.

Therefore, the test cases selection problem can be represented as the following constrained optimization problem.

maximize m(A)

subject to: $A \subseteq V$

dist(*case1*,*case2*) >=$\varepsilon_{\_min}$, for any *case1*, *case2* in A

cost(A) <= **C**

To this end, we define a linear cost function, that is, we assume that the total resource consumptions are directly proportional to the number of the selected test cases. More precisely, if we use A to denote the set of the selected test cases, then there exists a positive constant scalar C, such that

$$cost(A) = C \times m(A)$$

where m(A) is the number of selected test cases. So, the cost constraint can be represented as

$$cost(A) <= C$$

To formulate the distance constraint, the distance between test cases case1 and case2 is defined as the product of their similarity and dissimilarity, which are defined as follows:

The dissimilarity between case1 and case2 is the number of indices at which the elements of A and B are different, and their similarity is defined by

$$2^{-length(LCP(case1,\ case2))}$$

where LCP(*case1*, *case2*) is the longest common prefix of the two cases.

In order to get the initialization test cases, we set an algorithm for initialization test cases, at first, we find the longest test case in set V, and set ε = length (t), and move it to set A. then, $\varepsilon = length - 1$; then we follow ε to find the second longest test case, so on. Finial, the test cases in set A, they have different length each of them. This is the initialization test case that we prepared for test case selector.

To test the test case selector, a case producer that generates a given number of test cases is also developed. Therefore, the project implements two functions--implementation of a case producer and a case producer.

3.4.1 Basic Classes of the Testcases Selector

To implement the main functions------ test case selection, four classes are designed: testCase, test, selector, and fileReader. The class diagram is shown in the following:

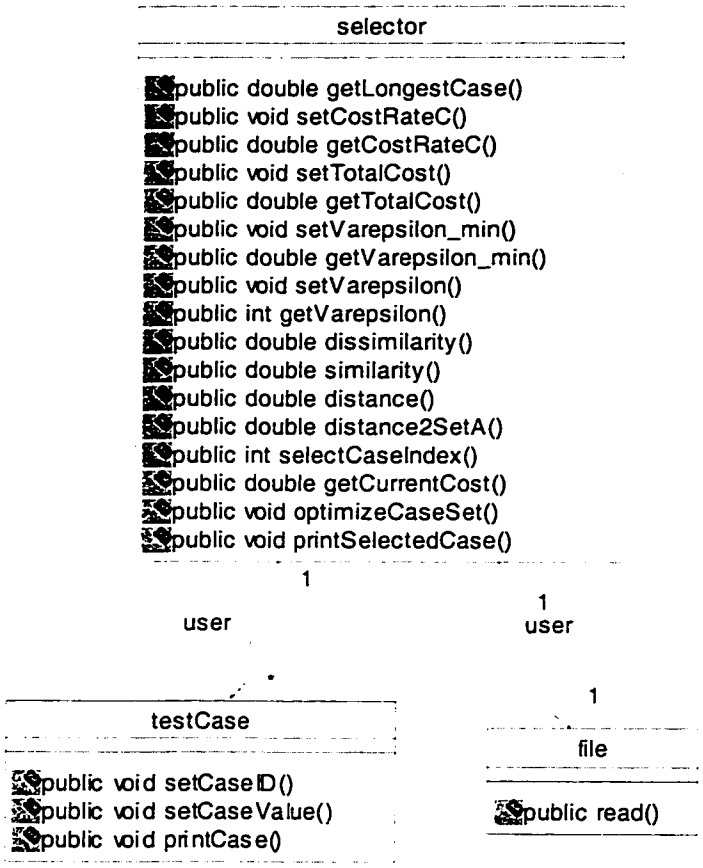

Figure 9: Selector Class Diagram

1. Class selector

==========

--- the main program that performs the selector's function.

Attributes:

---

1) protected static LinkedList caseSetV

--- a linked list consisting of test cases, denoting the entire original test cases available for choosing.

2) protected static LinkedList caseSetA

--- a linked list saving all the selected test cases.

Methods:

---

1) public double getLongestCase()

--- get the number of test cases in the caseSetV

2) public void getInitializationCase()

--- get the initialization test cases from the caseSetV

3) public void setCostRateC(double m)

--- set the cost rate of testing one case

4) public double getCostRateC()

--- get the cost rate of testing one case

5) public void setTotalCost(double m)

--- set up the total cost allowed.

6) public double getTotalCost()

--- return the bound of the allowed cost.

7) public void setVarepsilon_min(double m)

--- set up the minimal distance between the selected test cases.

8) public double getVarepsilon_min()

--- return the minimal distance between the selected test cases.

9) public void setVarespsilon(int m)

--- set parameter varepsilon

10) public int getVarepsilon()

--- return parameter varepsilon

11) public double dissimilarity(testCase case1, testCase case2)

--- calculate the dissimilarity between two test cases, which is defined as the number of indices at which the elements of A and B are different.

12) public double similarity(testCase case1, testCase case2)

--- calculate the similarity between two test cases A and B, which is defined as

$2^{-\text{length(LCP(case1, case2))}}$

where LCP(case1, case2) is the longest common prefix of the two cases.

13) public double distance(testCase case1, testCase case2)

--- calulate the distance between two cases, which is defined as

dissimilarity(case 1, case 2)*similarity(case 1,case 2)

14) public double distance2SetA(testCase case0)

--- compute the distance between a case and the set A of test cases, which is

defined by distance of case 0 to caseSetA = min{distance(case0, case1): case1 is in caseSetA}

15) public int selectCaseIndex()

--- return the indix of the selected case in the caseSetV

16) public double getCurrentCost()

--- return the cost to test all the selected test cases in caseSetA

17) public void optimizeCaseSet()

--- add a test case to caseSetA. If the following conditions

are satisfied

(a) the cost to test the currently selected test cases is less than the predefined

cost bound, that is, we still have some cost quota to use.

(b) there exist some test cases in initialization test cases in SetV with distance between themselves greater than

the predefined varepsilon

then select one test case from caseSetV and add it to caseSetA.

18) public void printSelectedCase()

--- out put all the selected test cases.

2. Class: testCase --- define test case

==========

Attributes:

---

1) private int caseID --- ID number of a test case

2) private int len --- the length of the test case

3) private int[] caseVector --- binary vector storing the test case

Methods:

---

public void setCaseID(int n) --- set an ID to a test case

public void setCaseValue(String s) --- set a string a test case

public void printCase() --- print out the test case

3. Class file

==========

--- open a file, read the file line by line, organize each line as a test case,

and add it into the linked list caseSetV.

Attributes:

---

private String args --- specify the file name to be opened.

Methods:

---

public static void main()

--- open a file, read the file line by line, organize each line as a test case,

and add it into the linked list caseSetV.

3.4.2 Sample Running

3.4.2.1 Set of Original Test Cases

The input to the system is the test suite generated by the automotive test cases generator module. In the sample running the set of original test cases has 25 cases each of them has 25 events in Figure 6.

3.4.2.2 Set of Binary Representations of Test Cases

The input of the filter is the original test cases for system testing. After filter subsystem, the original test cases have been transformed to the binary strings representation of test cases as the input of selector subsystem. The input of selector subsystem is show in Figure 8.

In this sample running, we test parameters as follow

One: let us choose a set of parameters as follows:

C=1, // cost rate

totalCost = 19;

varepsilon_min=0.2;

varepsilon = 30;

With this set of parameters, running selector produces the following results:

G:\project11\hrw>java SetParameter

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 3 setV has:10

case 24: 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

case 3: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 0: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Original Case 0:<<G.opened, C.activate>+0/4> : C.Near

Original Case 3:<<G.closed, C.monitor>> : C.Near, G/C.Lower, G.Down

Original Case 24:<<G.toOpen, C.idle>+8/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

Two: let us choose a set of parameters as follows:

C=1, // cost rate

totalCost = 19;

varepsilon_min=0.02;

varepsilon = 30;

With this set of parameters, running selector produces the following results:

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 5 setV has:8

case 24: 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

case 9: 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 3: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 1: 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 0: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Original Case 0: <<G.opened, C.activate>+0/4> : C.Near

Original Case 1: <<G.toClose, C.monitor>+0/4> : C.Near, G/C.Lower

Original Case 3: <<G.closed, C.monitor>> : C.Near, G/C.Lower, G.Down

Original Case 9: <<G.toOpen, C.idle>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise

Original Case 24: <<G.toOpen, C.idle>+8/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

Three: let us choose a set of parameters as follows:

C=1, // cost rate

totalCost = 19;

varepsilon_min=0.01;

varepsilon = 30;

With this set of parameters, running selector produces the following results:

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 6 setV has:7

case 24: 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

case 13: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 6: 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 3: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 1: 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 0: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Original Case 0: <<G.opened, C.activate>+0/4> : C.Near

Original Case 1: <<G.toClose, C.monitor>+0/4> : C.Near, G/C.Lower

Original Case 3: <<G.closed, C.monitor>> : C.Near, G/C.Lower, G.Down

Original Case 6: <<G.closed, C.deactivate>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit

Original Case 13: <<G.toOpen, C.idle>+1/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4

Original Case 24: <<G.toOpen, C.idle>+8/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

Four: let us choose a set of parameters as follows:

```
C=5, // cost rate
    totalCost = 3;
    varepsilon_min=0.01;
    varepsilon = 30;
```

With this set of parameters, running selector produces the following results:

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 0 setV has:13

No Test Case is Selected

Five: let us choose a set of parameters as follows:

C=3, // cost rate

totalCost = 19;

varepsilon_min=0.01;

varepsilon = 4;

With this set of parameters, running selector produces the following results:

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 4 setV has:9

case 24: 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0

case 13: 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 6: 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

case 3: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Original Case 3: <<G.closed, C.monitor>> : C.Near, G/C.Lower, G.Down

Original Case 6: <<G.closed, C.deactivate>+0/4> : C.Near, G/C.Lower, G.Down, C.Exit

Original Case 13: <<G.toOpen, C.idle>+1/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4

Original Case 24: <<G.toOpen, C.idle>+8/4> : C.Near, G/C.Lower, G.Down, C.Exit, G/C.Raise, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4

Six: let us choose a set of parameters as follows:

C=1, // cost rate

totalCost = 19;

varepsilon_min=2;

varepsilon = 30;

With this set of parameters, running selector produces the following results:

Original case sets

set A has: 0 set V has:25

after Initialization

set A has: 0 set V has:13

after case selection

set A has: 0 setV has:13

No Test Case is Selected

From the results, we can compare **Two** with **One.** We only change the value of ε_min from 0.2 to 0.02 that means case *a* and case *b* have more similarity should be selected. So we have more selected test cases than case **One;** Compare **Three** with **One** and **Two,** we also change the value of ε_min from 0.02 to 0.01, that means case *a* and case *b* have more similarity should be selected. So there are 6 testcases are selected; Compare **Four** with **One, Two, Three,** we only change the value of Totalcost from 19 to 3, and Cost rate is 5, that means Cost rate larger than Totalcost. When Cost rate is larger than Totalcost, there is not testcase was selected. The result also display: No test case is selected; Compare **Five** with **One, Two, Three and Four,** we only change the value of ε from 20 to 4, that means we only want to select 4 testcases from the original test cases, from the result of **Five,** only 4 testcases are selected; Compare **Six** with **One, Two, Three, Four and Five,** we only change the value of ε_min from 0.02 to 2, that means distance between a and b should be larger than 2, but in our original test cases, we do not have *td(a, b)* larger than 2. So no test case is selected. Therefore, if we want more testcases

were selected from original testcases, we only change ε_min to small; if we want less testcases were selected form original testcases, we only change ε_min to large; if we only want a fixed amount testcases were selected from the original testcases, we only set ε equal one fixed value.

4 Conclusions

In this major report we introduce the test adequacy measurement for measuring the efficiency of testing process integrated into a rigorous environment for developing real-time reactive systems. We have chosen to represent formally the test cases domain as a metric space. This approach allow the use of metric-based test case selection algorithm to select the optimal set of test cases (from the test selection domain), and metric-based coverage evaluation measurement, based on testing distance measure. The main objective was to develop a tool that automatizes use of metric-based test coverage evaluation measurement, and incorporate it into the TROMLAB environment.

References

[AAM98] V.S. Alagar, R. Achuthan, D. Muthiayen. *Tromlab: A Software Development Environment for Real-Time Reactive Systems.* (First version 1996, revised 2001) submitted for publication.

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

[AO2000] V.S. Alagar and Olga Ormandjieva. *Testing Measurement in Real-Time Reactive Systems.* Concordia University, Montreal, Canada, 2000.

[Be98] C. Bennett, P. Gacs, M. Li, P. Vitanyi, W. Zurek. *Information Distance* IEEE Transactions on Information Theory (44). $. Pp. 1407-1423. 1998.

[Che03] Minghua Chen *The Implementation of Specification-based Testing System for Real-time Reactive System in TROMLAB* Master major report, Department of Cmputer Science, Concordia University, Montreal, Canada, 2003.

[FP97] Fenton, N and Pleeger, S. Software Metrics: A Rigorous & Practical Approach. Chapman & Hall, 1997.

[HP85] Harel D., Pnueli A. *On the development of reactive systems.* In logic and Models of Concurrent Systems, NATO, Advanced Study Institute on Logics and Models for Verification and Specification of concurrent Systems. Springer Verlag, 1985.

[W86] E.Weyuker, "Axiomatizing Software Test Data Adequacy", Journal of systems and Software (March 1991), pp.207-216.

[W86] E. Weyuker, " Axiomatizing Software Test Data Adequacy", IEEE Transactions on Software Engineering, SE-12, 12, pp.1128-1138, 1986

[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.

[ZHM97] H. Zhu, P. Hall, J. May. *Software Unit Test Coverage and Adequacy*. ACM computing Surveys (29), 4, pp.366-427, 1997.

Appendix A: Java Printing SetParameter(Selector)

```
import javax.swing.*;
import java.awt.*;

public class SetParameter extends javax.swing.JFrame {


    public SetParameter() {
        initComponents();
    }


    public void initComponents() {//GEN-BEGIN:initComponents
        SetEvent se = new SetEvent(this);
        jLabel1 = new JLabel();
        jTextField1 = new JTextField(10);
        jLabel2 = new javax.swing.JLabel();
        jTextField2 = new JTextField(10);
        jLabel3 = new JLabel();
        jTextField3 = new JTextField(10);
        jLabel4 = new JLabel();
        jTextField4 = new JTextField(10);
        jButton1 = new JButton();


        getContentPane().setLayout(new FlowLayout());

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        jLabel1.setText("cost rate");
        getContentPane().add(jLabel1);

        //jTextField1.setText("jTextField1");
        getContentPane().add(jTextField1);

        jLabel2.setText("total cost");
        getContentPane().add(jLabel2);

        //jTextField2.setText("jTextField2");
        getContentPane().add(jTextField2);

        jLabel3.setText("E_min");
        getContentPane().add(jLabel3);

        //jTextField3.setText("jTextField3");
        getContentPane().add(jTextField3);

        jLabel4.setText("E");
        getContentPane().add(jLabel4);
```

```
        //jTextField4.setText("jTextField4");
        getContentPane().add(jTextField4);

        jButton1.setText("SET");
        getContentPane().add(jButton1);
        jButton1.addActionListener(se);
        pack();
        //setVisible(true);


    }//GEN-END:initComponents
    public String getPa1()
    {
       return jTextField1.getText();
     }
     public String getPa2()
    {
       return jTextField2.getText();
     }
     public String getPa3()
    {
       return jTextField3.getText();
     }

    public String getPa4()
    {
       return jTextField4.getText();
     }


    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {//GEN-
FIRST:event_exitForm
        System.exit(0);
    }//GEN-LAST:event_exitForm

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new SetParameter().show();
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JTextField jTextField4;
    private javax.swing.JTextField jTextField3;
    private javax.swing.JTextField jTextField2;
    private javax.swing.JTextField jTextField1;
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration//GEN-END:variables

}
```

Appendix B: Java Printing Test

```
import java.util.*;
import java.io.*;

public class test
{

//attributes

protected static LinkedList caseSetV;
protected static LinkedList caseSetA;

protected static int numberOfCases;
protected static int lengthOfCases=25;

//////////////
//// methods
//////////////

//get the index of the longest test case

public int getLongestCaseIndex()
{
int tmplen = 0;
int ind = 0;

for (int i=0; i<caseSetV.size(); i++)
{
testCase tempCase = (testCase) caseSetV.get(i);
if (tempCase.caseLength()>tmplen)
{tmplen++; ind = i;}
}
return ind;
}

//// get a case with length n

public int getCaseOfLength(int n)
{

int ind=-1;
testCase tempCase;

for (int i=0; i<caseSetV.size(); i++)
{
tempCase = (testCase) caseSetV.get(i);

 if(tempCase.caseLength() == n)
 {ind = i; return ind;}
}
return -1;
}
```

```
////////////////

protected void initialization()
{
int longestInd = getLongestCaseIndex();

// move the longest case to setA

testCase tempCase = (testCase) caseSetV.remove(longestInd);
caseSetA.add(tempCase);

// move case with length L-1 to set A

int longestL = tempCase.caseLength()-1;

while (longestL > 0)
{
int tmpInd = getCaseOfLength(longestL);

if (tmpInd != -1)
{
tempCase = (testCase) caseSetV.remove(tmpInd);
caseSetA.add(tempCase);
}
longestL--;
}

// delete V and move A to V

 caseSetV.clear();

 int tmplenA = caseSetA.size();

 for (int i=0; i< tmplenA; i++)
 {
 tempCase = (testCase) caseSetA.get(i);
 caseSetV.add(tempCase);
 }
 caseSetA.clear();
}

/*
private void initialSetV()
{
   for (int i=0; i<numberOfCases; i++)
   {
   testCase tmp_Case = new testCase();
   tmp_Case.setCaseID(i);
   tmp_Case.produceRandomCase();
   caseSetV.add(tmp_Case);
   }
}
*/

//constructor

public test()
```

```
{
 caseSetV = new LinkedList();
 caseSetA = new LinkedList();
}
/*
// methods for attribute access

public void setCostRateC(double m)
{ C = m;}
public double getCostRateC(){return C;}

public void setTotalCost(double m){totalCost=m;}
public double getTotalCost(){return totalCost;}

public void setVarepsilon_min(double m){varepsilon_min = m;}
public double getVarepsilon_min(){return varepsilon_min;}


public void setVarespsilon(int m){varepsilon = m;}
public int getVarepsilon(){return varepsilon;}

//methods used to calculate the distance between 2 cases*/

static private double dissimilarity(testCase case1, testCase case2)
{
 int[] tempVector1 = case1.getCaseVector();
 int[] tempVector2 = case2.getCaseVector();

 int len = case1.getCaseLength();
 int tem = 0;
 for (int i=0; i<len; i++)
 {
  if (tempVector1[i] != tempVector2[i])
  tem++;
 }
 return tem;
}

static private double similarity(testCase case1, testCase case2)
{
 int[] tempVector1 = case1.getCaseVector();
 int[] tempVector2 = case2.getCaseVector();

 int len = case1.getCaseLength();
 int i=0;
 for (i=0; i<len && (tempVector1[i]==tempVector2[i]); i++);
 return Math.exp(-i);
}

//calulate the distance between two cases

static public double distance(testCase case1, testCase case2)
{
 return dissimilarity(case1, case2)*similarity(case1,case2);
}

// compute the distance between a case and the set A of test cases
```

```
public double distance2SetA(testCase case0)
{
 int tempsize = caseSetA.size();
 double tempDis = numberOfCases+1;

 for (int i=0; i<tempsize; i++)
 {
   testCase tempCase = (testCase) caseSetA.get(i);
   double tempVal = distance(case0, tempCase);
   if (tempVal < tempDis)
   tempDis = tempVal;
 }
 return tempDis;
}

public int selectCaseIndex(double x3)
{
  int tempsize = caseSetV.size();

  testCase tempCase;

  for (int i=0; i<tempsize; i++)
  {
  tempCase = (testCase) caseSetV.get(i);
  if (distance2SetA(tempCase) >= x3)
  return i;
  }
  return -1;
}

/*public double getCurrentCost()
{
 int temp = caseSetA.size();
 return C*temp;
}*/

public void optimizeCaseSet(double x1, double x2, double x3, int x4)
{
 int temp = caseSetA.size();
 double temp1 =  x1*temp;

System.out.println(caseSetV.size());
 while(temp1<x2 && x4>0 && caseSetV.size()>0)
 {
  //varepsilon=varepsilon-1;
  x4 = x4 -1;
  int tempIndex = selectCaseIndex(x3);

  if (tempIndex != -1)
  {
   testCase tempCase = (testCase) caseSetV.remove(tempIndex);
   caseSetA.add(tempCase);
  System.out.println("A size:" + caseSetA.size());
  }
 }
}
```

```
public void printSelectedCase()throws java.io.IOException
{
 int currSize = caseSetA.size();

 if (currSize == 0)
 {System.out.println("No Test Case is Selected");
  return;}

 testCase tempCase;
 int[] caseID = new int[currSize];
 for (int i=0; i<currSize; i++)
 {
 tempCase = (testCase) caseSetA.get(i);
 caseID[i]=tempCase.printCase();
 }

  String fname = "firstcases.txt";
  BufferedReader reader = new BufferedReader( new FileReader( fname ) );
  String line = reader.readLine();
  int row = 1;
  while (line != null)
  {
    for(int i=0; i<currSize; i++)
    {
     if(row == (caseID[i]+1))
     {
       System.out.println("Original Case "+caseID[i]);
       System.out.println(line);
       break;
     }
   }
   row = row + 1;
   line = reader.readLine();
 }
}

}
```

Appendix C: Java Printing TestCase

```
import java.lang.Math;

 public class testCase
 {
 //protected int caseID=-1;
 //protected static int len=0;
 //protected int caseVector[];
 private int caseID=-1;
 private static int len=test.lengthOfCases;
 private int caseVector[];

//constructor
public int caseLength()
{
  int tmplen=0;
  for (int i=0; i< len; i++)
  { if (caseVector[i] == 1)
    tmplen=tmplen+1;
  }
  return tmplen;
}

 public testCase()
 {
  caseID = -1; //empty case
  len = test.lengthOfCases;
  caseVector = new int[len];
 }

 public void setCaseID(int n)
 {
  caseID = n;
 }

 public void setCaseValue(String ss)
 {
      char cc;
      int i=0; int j=0;

      while (i < ss.length())
    {
    cc = ss.charAt(i);

      if (cc=='0')
      {caseVector[j] = 0; j++;}

      if (cc=='1')
      {caseVector[j] = 1; j++;}
    i++;
    } //end while
    len = j;
}
```

```
public void produceRandomCase()
{
      String temps = null;

      for (int i=0; i< len; i++)
        {
         double rnd = Math.random();
         if (rnd<0.5)
         caseVector[i]=0;
         else caseVector[i] = 1;

          //caseVector.charAt(i) = '0';
       //  else temps = temps+"1";
       //  caseVector = temps;
   }
}

  public int getCaseLength(){return len;}
  public int[] getCaseVector(){return caseVector;}

 public int printCase()
 {
    System.out.print("case " + caseID + ":");
    for (int i=0; i< test.lengthOfCases; i++)
    System.out.print(" "+caseVector[i]+" ");
    System.out.print("\n");
    return caseID;
    //System.out.print("case"+ caseID + ":" +caseVector + "\n");
 }

}
```

Appendix D: Java Printing FileReader

```
 import java.util.*;
 import java.io.BufferedReader;
 import java.io.FileReader;
 import java.io.IOException;

class fileReader{
            // VARS

            String args;
            //static LinkedList caseSetV;

            // CONSTRUCTOR
            public fileReader( String args ) {
            this.args = args;
              //  caseSetV = new LinkedList();
              }

            // PUBLIC METHODS

 public void run() throws java.io.IOException
            {
                // This is your new main.
                String fname = args;

                //String fname = "cases.txt";

                BufferedReader reader = new BufferedReader( new
FileReader( fname ) );

                String line = reader.readLine();
                int tempid = 0;

                while(line != null)
                {
                 //System.out.println(line);
                 testCase tempCase = new testCase();

                          tempCase.setCaseID(tempid);
                          tempCase.setCaseValue(line);
                          test.caseSetV.add(tempCase);
                          tempid++;
                          line = reader.readLine();
                         }
             }
}
```