Journal of Computer Science 4 (6): 454-462, 2008
ISSN 1549-3636
© 2008 Science Publications

The Conceptual Design of Module Documentation Based Testing Tool

[1]Salmi Baharom and [2]Zarina Shukur
[1]Faculty of Computer Science and Information Technology, University Putra Malaysia, Malaysia
[2]Faculty of Science and Information Technology, University Kebangsaan Malaysia, Malaysia

Abstract: Software testing plays an important role to assure the quality of software and can be highly effective if performed rigorously. Studies found that testing can benefit from formal specification as it provides precise description of expected software behavior and most importantly, it is in a form that it can be manipulated easily for automation purpose. Grey-box testing approach usually based on knowledge obtains from specification and source code while seldom the design specification is concerned. In this study, an approach was described with an example of circular queue for testing a module with internal memory from its formal specification based on grey-box approach. However, in this research, we proposed a grey-box testing approach that uses the knowledge of design specification instead of source code. We utilized formal specifications that were documented using Parnas's Module Documentation (MD) method to generate test oracle and to execute the test. The MD provides the information of external and internal view of a module that is useful in our testing approach.

Key words: Specification-based testing, grey-box testing, testing tool, test oracle generator

INTRODUCTION

Software testing is a part of validation and verification process that is performed to verify software quality and reliability. The goal of software testing is to reveal software faults by executing the program on inputs and comparing the outputs of the execution with expected outputs. If perform rigorously, testing can be used to increase software engineer confidence towards software correctness. However, testing is one of most time-consuming and costly part of the software development. Earlier studies indicated that software testing can consume more than fifty percent of software development cost[1]. Besides, the National Institute of Standard and Technology reported that in the U.S alone the annual costs of an inadequate infrastructure for software testing is estimated to range from 22.2- 59.5$ billion[2].

The use of mathematical development techniques which is also so-called formal methods can provide high assurance of correctness as mathematics has the ability to give precise definition of problems. Thus, ambiguity and inconsistency can be eliminated early in software development process. Many researchers have put particular emphasis on developing an effective method to utilize mathematics for specifying and designing software[3-5]. The use of formal specifications provides significant opportunity to develop effective testing techniques[6,7].

This research addresses the problem of improving the effectiveness of fault detection where the focus of the work is on unit/module testing where each module may consist of several programs. The aim is at investigating the strategies and techniques to automate module testing. In particular, we investigate the use of Module Documentation (MD) that written using standard mathematical notation in automating the process of test oracle generation and test execution.

Theoretical background: This section discusses theoretical issues that form an important background of this research.

Software Testing: Testing is an important process in software development which is employed to ensure that the design and implementation of programs comply with the specified requirements. The goal of testing is failure detection whereby we observe differences between the behaviors of implementation and expected behavior as specified in the specification. IEEE[8] defined software testing as: The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.

Corresponding Author: Salmi Baharom, Faculty of Computer Science and Information Technology, University Putra Malaysia, 43400 UPM, Serdang, Selangor, Malaysia Tel: +6019-2728807 Fax: +603-89422534

In general, testing can be performed either by static or dynamic technique. Static techniques are based on documents examination either manually or automatically and most importantly without need to execute the Software Under Test (SUT). The examples of static technique are software inspection, software reviews, code reading and algorithm analysis and tracing. Dynamic technique refers to testing that requires the SUT to be executed.

The process of software testing involves selecting test cases, executing the software with the selected test cases and evaluating the results produced by those executions whether the results conform to its specification, so-called test oracle[1,9,10]. In literatures, software testing has mainly concentrated on the problem of selecting test cases[1,11-13] but it does not mean that test oracle problem is trivial. Instead all software testing methods rely on the availability of test oracle and test without able to differentiate success or failure is a useless test even with good test cases.

Generally, there are four types of test which are unit testing, module testing, integration testing and system testing. Unit testing is meant for checking individual component independently. Module testing checks the integration between all components in a module. Integration testing checks the integrations between collections of modules. System testing checks the integration sub-system integration.

The classical approaches to software testing are black-box and white-box testing. IEEE[8] defines black-box testing (functional testing) as testing that ignores the internal mechanism of a system or component and focuses solely on the output generated in response to selected inputs and execution conditions. The white-box testing (structural testing) is testing that takes into account the internal mechanism of a system or component i.e. it requires the internal structure of the SUT completely exposed to the tester. Due to pervasive web and internet application, grey-box testing approach has gain increasing popularity in software testing. Grey-box testing approach is testing with limited knowledge of the internal workings of the SUT.

Specification-based testing: Specification is statement of some of the properties required of a product, or a set of products and a product is considered faulty if the statements made in its specification are not true of that product[14]. While specification-based testing refers to testing that uses information obtained solely from specification. Testing from specification gives several advantages such as it allows test to be developed earlier and it can be ready before the program is finished. Besides, any inconsistencies and ambiguities in the specification can be detected and removed during the test development.

In earlier studies, specification-based testing looked at input/output relation which is seen as black-box testing approach. Recently there are studies that proposed the use of integrated approach in specification-based testing which is also known as grey-box testing approach. The grey-box approach differs from black-box approach in that it takes into consideration the internal structure of SUT. The internal structure of SUT can be obtained from design specification such as UML activity diagram[11].

Grey-box testing: Black box testing method generates tests from specification and purely in terms of observable input and output without the information of internal structure. It does not require information on how the system was implemented. Likewise, white-box testing method does require internal structure of unit being tested completely exposed to the tester.

Many program faults can be overlooked by black-box testing since a test primarily stresses the user interface and does not consider the inner structure of the test unit[15]. On the other hand, testing without specification knowledge (white-box approach) may not be effective to show if program have been properly implemented as stated in its specifications. Consequently, studies[11,13,16] suggested the use of integrated approach which is known as grey-box testing approach. Grey-box testing is a mixture of black-box and white-box testing techniques, which considers both the external view and the internal structure of SUT.

Test oracle: Test oracle is the important component in testing in order to determine whether SUT behaved as expected during the test execution. It may be done either manually or automatically. As in[17], an ideal test oracle should satisfy three characteristics which are complete desirable properties, avoid over-specification and efficiently checkable.

In literatures, various types of test oracles are found such as embedded assertion language and executable specification. Embedded assertion language is considered as explicit oracle whereby executable assertions are embedded within the implementation. For example Anna provides formal language for annotating Ada implementations with assertions. Executable specification language is language that provides two versions of code such Paisley, OBJ and TRIO. Alternatively for non-executable specification there are studies to develop Test Oracle Generator (TOG) from formal specification.

Formal Specification: The following definition of formal specification is obtained from[18]. A specification is formal if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax) rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics) and the rules for inferring useful information from specification (the proof theory). Testing can benefit from formal specification as it provides precise description of expected software behavior and most importantly, it is in a form that it can be manipulated easily for automation purpose.

The use of formal specifications provides significant opportunity to develop effective testing techniques. It has been reported that the use of formal methods in software testing can help to produce high integrity systems in a cost-effective way and it offers a simpler, structured and more rigorous approach to the development of functional tests than standard testing techniques[6,7]. Hence, efforts to develop effective formal specifications such as pre/post-condition approach, functional/relational approach and model-based approach provide significant opportunity towards software testing.

MATERIALS AND METHODS

Software projects are usually organized as a module. Parnas *et al.*[19,20] defines a module as work assignments. A module has a private data structure and one or more access-programs. As described by[20], for each module there should be a Module Interface Specification (MIS) that treat a module as a black-box. The MIS identifies those programs that can be invoked from outside the module, called access-programs and describing the externally-visible effects of using them. For each MIS implementation there should be an internal documentation that is known as Module Internal Design Document (MIDD) that gives information on how a module should be implemented. We named both MIS and MIDD as module documentation (MD).

The MIDD must be sufficiently precise that one can use it, together with the MIS, to verify the workability of the design. Both MIS and MIDD are described as representations of one or more mathematical relations i.e., functional/relational approach[20]. The MIS is used by anyone who either maintains or uses a component whereas the MIDD is used only by those who design, build, review or maintain the module. As described by[20], the MIDD should contain three types of information:

- A description of the module's data structure used
- A description of the effect of each access-program on the value of the variable in the data structure which is known as program function
- A description that describes the intended interpretation of that data structure. It is known as abstraction relation since it is a mapping from the more concrete data structure to a more abstract external view

Trace Function Method (TFM): MIS can be specified using TFM[21]. The TFM is being developed by the SQRL research group in the University of Limerick that is the enhancement of Trace Assertion Method (TAM). As in TAM, the TFM uses tabular expressions and the concept of traces of events to produce complete specifications and descriptions. TFM consists of:

- A list of the input and output variables, including shared global variables and their types
- A set of output function definitions, specifying the value of each of the output variables as a function of the trace of the components history
- A set of auxiliary function definitions used in the output function definitions

Relational specification: Parnas introduce the concept of Limited Domain Relation or LD-relation in short, to allow for non-deterministic program. In this approach, a relation is supplemented with an additional set, a subset of the relation's domain called the competence set. The competence set contains with the states in which termination is guaranteed. The definition of LD-relation and the meaning of domain and competence sets which are adopted from[19] are as follows:

Definition 1: Let U be a set. An LD-relation on U is ordered pair:

$L = (R_L, C_L)$
Where:
R_L = Relational component of L, is a relation on U, i.e. $R_L \subseteq U \text{ X } U$
C_L = Competence set of L, is a subset of the domain of R_L, i.e. CL Dom(R_L)

Definition 2: Let P be a program, let U be a set of states and let $L_p = (R_p, C_p)$ be an LD-relation on U such, that:
$(x,y) \in R_p \Leftrightarrow (x,..., y) \in \text{Exec}(P, U), C_p = S_p$
L_p, is called the LD-relation of P and the description of P.

Table 1: Meaning of domain and competence sets

Behavior of program P	Competence set C_{L_P}	Domain of R_{L_P}	R_{L_P}
P terminates when started in x	Include x	Include x	Includes (x,y) if P might terminate in y when started in x
P some terminates when started in x	Does not include x	Include x	Includes (x,y) if P might terminate in y when started in x
P never terminates when started in x	Does not include x	Does no include x	No pairs of the form (x,y)
P never terminates	Empty	Empty	Empty
P is never guaranteed to terminate but may	Empty	Empty	Includes (x,y) if P might terminate in y when started in x

If C_p = Dom(R_p), then (by convention) the competence set need not be given explicitly. In other words, if C_p is not given, then it is, by default, Dom(R_p).

Predicate logic: LD-relation uses predicate logic to express the specification. It differs from traditional logic in that it allows the use of partial functions, functions whose value is not defined for certain value of its input types. The use of partial function in writing program specifications is useful when we want to observe the behavior of software and only accept the definite answer either true or false. The well-known problem of partial function is usually illustrated using square root function as an example. Details on the mathematical concept of predicate logic can be found in[3].

Before and after value: The following convention is adopted from[19] to indicate before and after value.

Let P be a program and $x_i,\ldots,x_k$ be the program variables used in P. Then

- x_i (to be read x_i after) denotes the value of the programming variable x_i after execution of P
- x_i (to be read x_i before) denotes the value of the programming variable x_i before execution of P

MD example: The idea of module documentation method is illustrated using an example of Circular Queue class. A circular queue is designed using linear model but wrap around from end to beginning of an array. Let us consider a circular queue implemented using an array of length QSIZE, which contains at most QSIZE element of type integer. In the circular queue, data is entered from the rear of a queue and data is removed from front of a queue. Hence, two indexes are required, which are front and rear to keep track of the first and the last data in the queue, respectively. When a data is removed from the queue, the value of front is increased by 1. When the value of front reaches QSIZE then the next value of front is set to 0. The value of front is calculated using the following formula, where % is modular operator:

$$\text{front} = (\text{front}+1)\%\ \text{QSIZE}$$

Similarly, when data is inserted in the queue, the value of rear is increased by 1 and the value is set to 0 when it reaches QSIZE. The following formula is used to calculate the value of rear.

$$\text{rear} = (\text{rear}+1)\%\ \text{QSIZE}$$

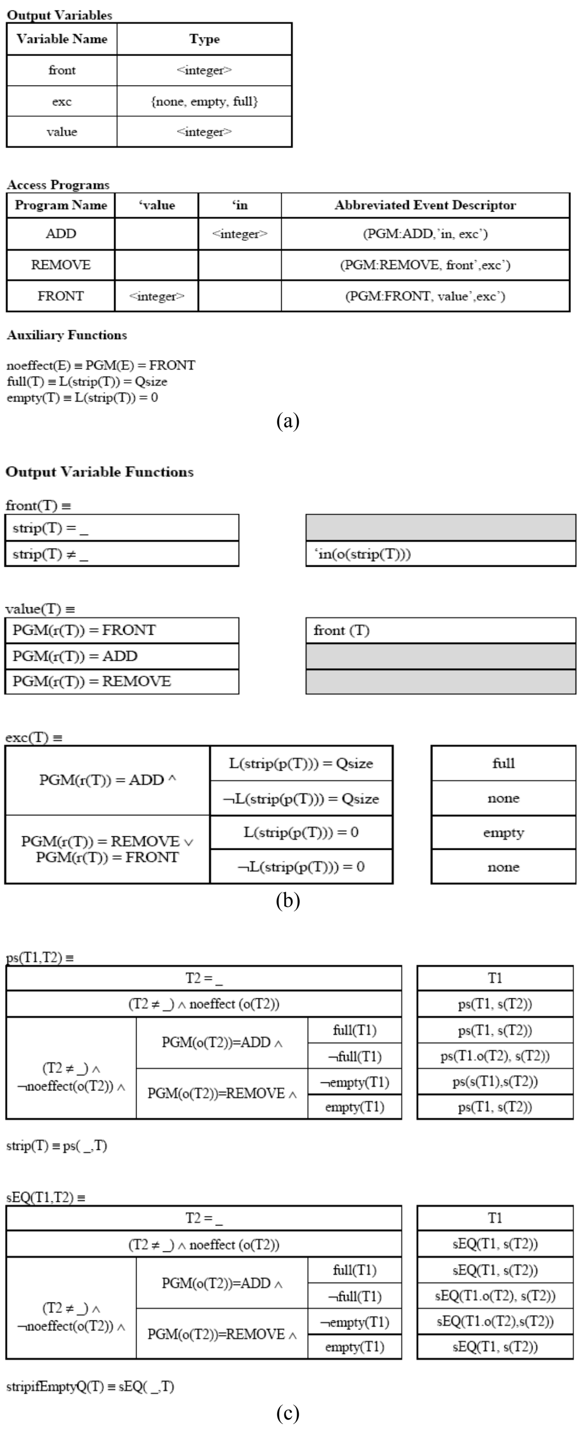
A counter variable named len is used to count the number of data inserted in the queue. Initially, len is set to 0 when the queue is empty and it increases each time a data is inserted in the queue. Likewise, it decreases each time a data is removed from the queue. Len variable is an indicator to determine whether a queue is full or empty. A queue is empty if len = 0 and full when len = QSIZE. The following Fig. 1 and 2 shows the MIS and MIDD respectively for a circular queue as described above.

The abstraction relation as shown in Fig. 2b specifies the relation between state and the external view of the circular queue. The external view is described in terms of sequence of events (trace). We explain the concept of abstraction relation based on the above example of circular queue. For simplicity, let us assume that the size of queue is 4. Based on the above assumption and the example of MD described above, a state DS where dataQ = [a,m,i], front = 1, rear = 3, len = 3 and QSIZE = 4 can correspond to many traces such as listed below.

- add(b).remove().add(a).add(m).add(i)
- add(a).add(a).remove().add(m).add(i)
- add(b).add(a).add(m).remove().add(i)
- add(a).add(a).add(m).add(i).remove()

RESULTS

Md-based testing tool: We propose module testing approach known as MD-based Testing Tool that can be done automatically by programmers. The MD-based Testing Tool consists of five components which are Test Oracle Generator, TFM Simulator, Test Harness, Tested Data State Storage and MUT Driver. The conceptual design of our testing tool is represented in Fig. 3.



Output Variables

Variable Name	Type
front	<integer>
exc	{none, empty, full}
value	<integer>

Access Programs

Program Name	‘value	‘in	Abbreviated Event Descriptor
ADD		<integer>	(PGM:ADD,’in, exc’)
REMOVE			(PGM:REMOVE, front’,exc’)
FRONT	<integer>		(PGM:FRONT, value’,exc’)

Auxiliary Functions

noeffect(E) ≡ PGM(E) = FRONT
full(T) ≡ L(strip(T)) = Qsize
empty(T) ≡ L(strip(T)) = 0

(a)

Output Variable Functions

front(T) ≡

strip(T) = _	
strip(T) ≠ _	‘in(o(strip(T)))

value(T) ≡

PGM(r(T)) = FRONT	front (T)
PGM(r(T)) = ADD	
PGM(r(T)) = REMOVE	

exc(T) ≡

PGM(r(T)) = ADD ^	L(strip(p(T))) = Qsize	full
	¬L(strip(p(T))) = Qsize	none
PGM(r(T)) = REMOVE ∨ PGM(r(T)) = FRONT	L(strip(p(T))) = 0	empty
	¬L(strip(p(T))) = 0	none

(b)

ps(T1,T2) ≡

T2 = _			T1
(T2 ≠ _) ∧ noeffect (o(T2))			ps(T1, s(T2))
(T2 ≠ _) ∧ ¬noeffect(o(T2)) ∧	PGM(o(T2))=ADD ∧	full(T1)	ps(T1, s(T2))
		¬full(T1)	ps(T1.o(T2), s(T2))
	PGM(o(T2))=REMOVE ∧	¬empty(T1)	ps(s(T1),s(T2))
		empty(T1)	ps(T1, s(T2))

strip(T) ≡ ps(_,T)

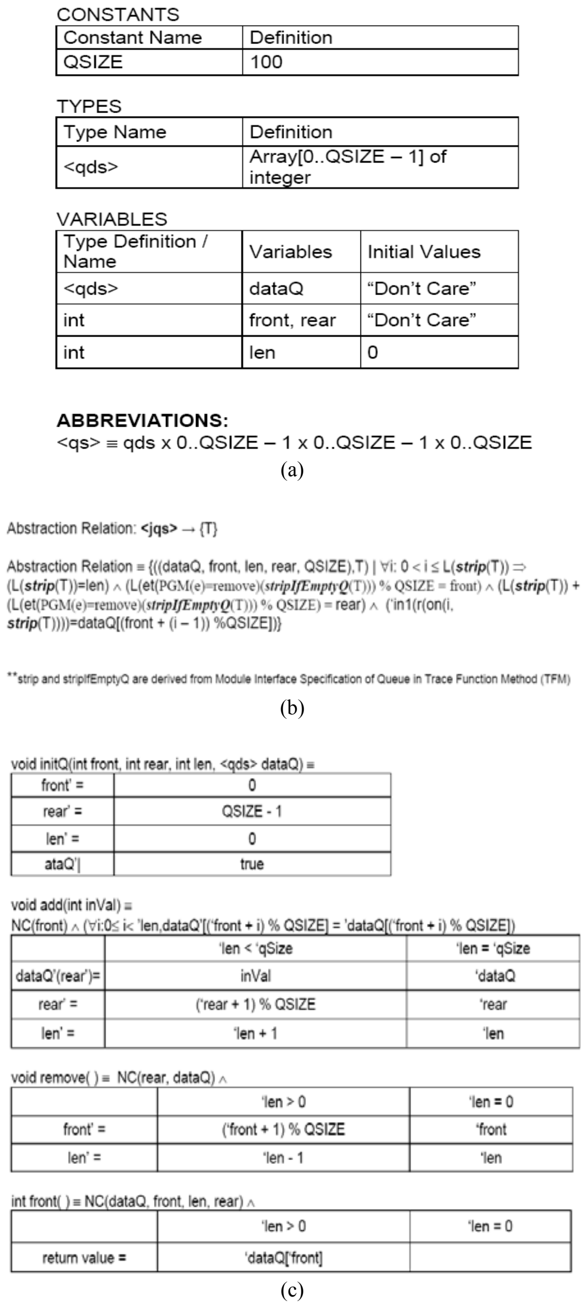
sEQ(T1,T2) ≡

T2 = _			T1
(T2 ≠ _) ∧ noeffect (o(T2))			sEQ(T1, s(T2))
(T2 ≠ _) ∧ ¬noeffect(o(T2)) ∧	PGM(o(T2))=ADD ∧	full(T1)	sEQ(T1, s(T2))
		¬full(T1)	sEQ(T1.o(T2), s(T2))
	PGM(o(T2))=REMOVE ∧	¬empty(T1)	sEQ(T1.o(T2),s(T2))
		empty(T1)	sEQ(T1, s(T2))

stripifEmptyQ(T) ≡ sEQ(_,T)

(c)

Fig. 1: MIS of a circular queue, (a) Input and output variables, (b) Output variable functions and (c) Auxiliary Functions



CONSTANTS

Constant Name	Definition
QSIZE	100

TYPES

Type Name	Definition
<qds>	Array[0..QSIZE – 1] of integer

VARIABLES

Type Definition / Name	Variables	Initial Values
<qds>	dataQ	“Don’t Care”
int	front, rear	“Don’t Care”
int	len	0

ABBREVIATIONS:
<qs> ≡ qds x 0..QSIZE – 1 x 0..QSIZE – 1 x 0..QSIZE

(a)

Abstraction Relation: **<jqs>** → {T}

Abstraction Relation ≡ {((dataQ, front, len, rear, QSIZE),T) | ∀i: 0 < i ≤ L(***strip***(T)) ⇒ (L(***strip***(T))=len) ∧ (L(et(PGM(e)=remove)(***stripIfEmptyQ***(T))) % QSIZE = front) ∧ (L(***strip***(T)) + (L(et(PGM(e)=remove)(***stripIfEmptyQ***(T))) % QSIZE) = rear) ∧ (‘in1(r(on(i, ***strip***(T))))=dataQ[(front + (i – 1)) %QSIZE])}

**strip and stripIfEmptyQ are derived from Module Interface Specification of Queue in Trace Function Method (TFM)

(b)

void initQ(int front, int rear, int len, <qds> dataQ) ≡

front’ =	0
rear’ =	QSIZE - 1
len’ =	0
ataQ’|	true

void add(int inVal) ≡
NC(front) ∧ (∀i:0≤ i< ’len,dataQ’[(‘front + i) % QSIZE] = ’dataQ[(‘front + i) % QSIZE])

	‘len < ‘qSize	‘len = ‘qSize
dataQ’(rear’)=	inVal	‘dataQ
rear’ =	(‘rear + 1) % QSIZE	‘rear
len’ =	‘len + 1	‘len

void remove( ) ≡ NC(rear, dataQ) ∧

	‘len > 0	‘len = 0
front’ =	(‘front + 1) % QSIZE	‘front
len’ =	‘len - 1	‘len

int front( ) ≡ NC(dataQ, front, len, rear) ∧

	‘len > 0	‘len = 0
return value =	‘dataQ[‘front]	

(c)

Fig. 2: MIDD of a circular queue, (a) A description of queue data structure (b) Abstraction relation and (c) Program functions

Test Oracle Generator (TOG): The role of TOG is to automatically generate oracle from MD. The task of generated oracle is to check expected output that is produced during the execution of Module under Test (MUT) with a test case. The expected output is not only the observable output of a module but also the value of internal data state after the execution. The test oracle generator will produce two test oracles based on two

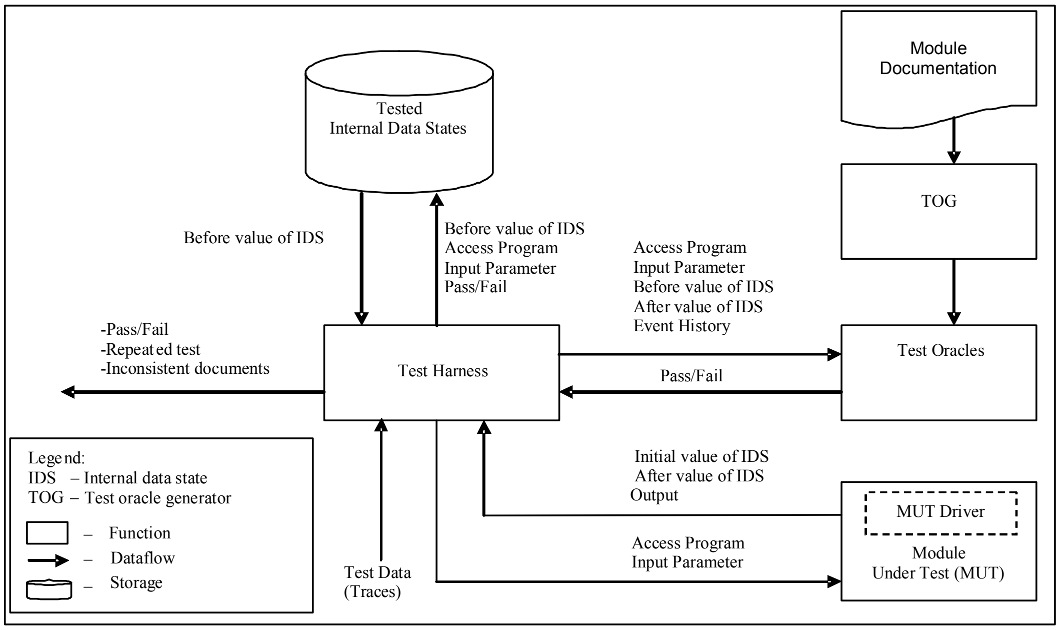


Fig. 3: The Conceptual design of MD-based testing tool

types of documents which are program function/relation (PF/R oracle) and abstraction relation (AR oracle).

TFM simulator: The TFM Simulator is developed based on MIS that is written in Trace Function Method (TFM). We utilize the TFM simulator that is developed by SQRL group. The TFM simulator acts as part of oracles for checking whether the trace correspond the output produced by the MUT.

Test harness: The test harness or test procedure serves as a middleware for providing test case for test oracles and MUT. First the test harness receives a test case in the form of sequence of events or trace. Each time before sending any trace to the test oracle, test harness checks if the data state of that particular trace has been tested before. If the data state has been tested, the test harness then checks the result. The testing process stops for if the result fails. Otherwise proceed with next event. However, if the trace has not been tested, test harness sends the trace for evaluation by the test oracles. Then the tested data states with the test result as well as the access program are kept in storage.

Tested data state storage: In order to detect previously tested data state, storage is required. The storage stores value of data state before execution, access program, input parameter and result of the test.

MUT driver: As MD-based testing tool is not only checking the observable output but also the value of internal data state. Therefore, it requires some mechanism to extract the value of internal data state of the module. Insertion of codes into the MUT by automated means is proposed.

Test data: The MD-based Testing Tool accepts test data in terms of sequence of events or traces, such as add(3). add(6). add(7). remove(). remove(). Each event of the test case will be executed one by one. For each event, it will be executed on the MUT and the internal data states and output produced from the execution will be checked. Finally the tested internal data states will be stored together with the test result. The detail of the algorithms is shown in Fig. 4 and 5 illustrates how the test data flows using the proposed approach.

DISCUSSIONS

Grey-box testing approach is usually based on the knowledge of specification and code. However, in specification-based testing where information is obtained solely from specification, a complete and precise design specification can replace the role of code.

The MIDD is the intermediate artifact between MIS and final code, which preserve the essential

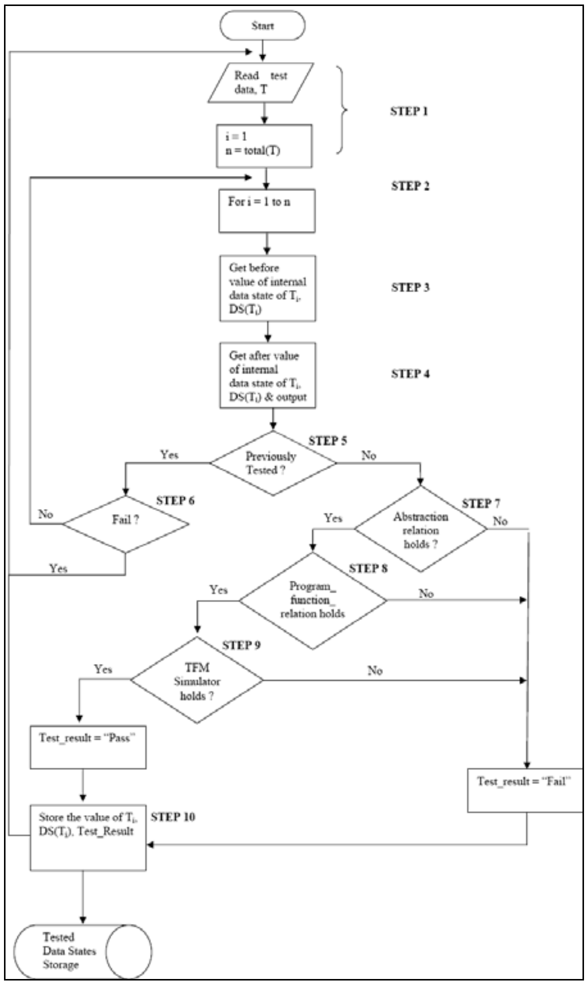

Start
Read test data, T
i = 1
n = total(T)
STEP 1
For i = 1 to n
STEP 2
Get before value of internal data state of T_i, $DS(T_i)$
STEP 3
Get after value of internal data state of T_i, $DS(T_i)$ & output
STEP 4
Previously Tested ?
STEP 5
Yes
No
Fail ?
STEP 6
Abstraction relation holds ?
STEP 7
Program_function_relation holds
STEP 8
TFM Simulator holds ?
STEP 9
Test_result = "Pass"
Test_result = "Fail"
Store the value of T_i, $DS(T_i)$, Test_Result
STEP 10
Tested Data States Storage

Fig. 4: An algorithm for MD-based testing tool

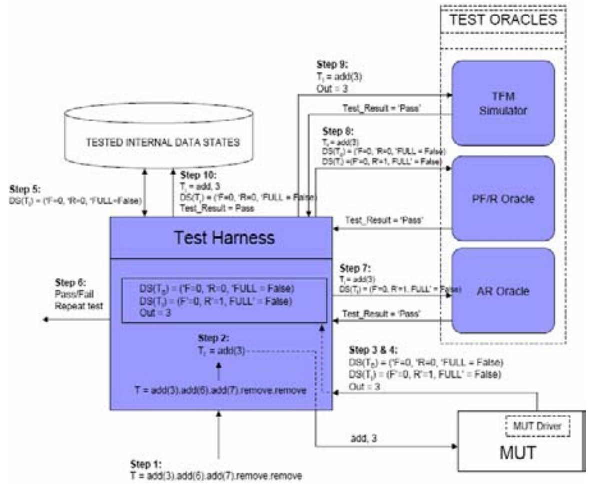

TEST ORACLES
TFM Simulator
PF/R Oracle
AR Oracle
TESTED INTERNAL DATA STATES
Test Harness
MUT Driver
MUT
Step 1:
T = add(3).add(6).add(7).remove.remove
Step 2:
T_i = add(3)
T = add(3).add(6).add(7).remove.remove
Step 3 & 4:
$DS(T_o)$ = ('F=0, 'R=0, 'FULL = False)
$DS(T_i)$ = (F'=0, R'=1, FULL' = False)
Out = 3
add, 3
Step 5:
$DS(T_i)$ = ('F=0, 'R=0, 'FULL=False)
Step 6:
Pass/Fail
Repeat test
$DS(T_o)$ = ('F=0, 'R=0, 'FULL = False)
$DS(T_i)$ = (F'=0, R'=1, FULL' = False)
Out = 3
Step 7:
T_i = add(3)
$DS(T_i)$ = (F'=0, R'=1, FULL' = False)
Test_Result = 'Pass'
Step 8:
T_i = add(3)
$DS(T_o)$ = ('F=0, 'R=0, 'FULL = False)
$DS(T_i)$ = (F'=0, R'=1, FULL' = False)
Test_Result = 'Pass'
Step 9:
T_i = add(3)
Out = 3
Test_Result = 'Pass'
Step 10:
T_i = add, 3
$DS(T_i)$ = ('F=0, 'R=0, 'FULL = False)
Test_Result = Pass

Fig. 5: An illustration of test case execution

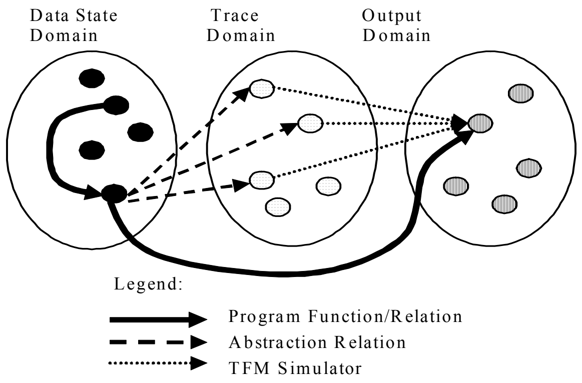


Fig. 6: Relation of traces, states and output

information from the MIS and are the basis of the code implementation. Apparently, the knowledge on how a module should be implemented by the programmer is observable. Thus, it gives tester the opportunity to test a module using grey-box testing approach. The knowledge of internal structure of the SUT that is obtained from MIDD allows us to test analogous to those used in white-box testing approach. Besides, it provides basis in terms of coverage measure in terms of data states.

The MD-based Testing Tool benefits from interface specification and design specification that is provided in the MD. Generally, the MIS gives relation between traces and output and the MIDD gives three types of relation which are (1) relation between before value and after value of data states, (2) relation between after value of data states and output and (3) relation between data state and traces. This information allows the design to be verified and therefore can be useful to test the implementation of module. We illustrate the relation between states and traces as in Fig. 6.

The essence of our approach is the use of combination formal interface specification that gives the input/output relation and formal design specification that describes the effect on some concrete data structure. Particularly, the use of abstraction relation offers significant opportunity toward the effectiveness of testing where it allows us to detect errors earlier in a lengthy test sequences. For example, a test case for a queue might include adding twenty items onto the queue; remove two items, adding fifty more, removing one and checking the result. Assume that the queue program only has one defect. The defect is that the element will be truncated when adding the 20th element. As long as we treat the component as a black-box, the problem will not be discovered until all elements on top of the truncated element are removed.

This makes testing expensive and relatively ineffective. However, this problem can be detected much more quickly because after each event, the abstraction relation is checked to determine either it holds or not. Besides, using MIDD allows for the detection of a return to a previously detected state.

CONCLUSIONS

We have presented the conceptual design of MD-based testing tool. This research contributes to the effectiveness of software testing by improving the effectiveness of test execution process by automated means. We proposed grey-box approach for testing a module with internal memory. The idea of this research is to use the knowledge of data structure instead of program structure. We believe our grey-box testing gives better coverage for black box testing with memory and can avoid some duplication in test cases by detecting return to the same state. Furthermore, the use of precise design documents proposed by Parnas, in particular the use of abstraction relation offers significant opportunity toward the effectiveness of fault detection.

ACKNOWLEDGEMENTS

Many thanks due to Prof. David L. Parnas and Dr. Xin Feng for their invaluable comments, questions and suggestions. We also greatly appreciate the helpful and invaluable discussions with Dr. Dennis K. Peters particularly in clarifying the MIDD example.

REFERENCES

1. Beizer, B., 1990. Software Testing Techniques. Ist Edn., Van Nostrand Reinhold Co, pp: 550. ISBN:0-442-20672-0.
2. NIST, 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. http://www.nist.gov/director/prog-ofc/report02-3.pdf.
3. Parnas, D.L., 1993. Predicate logic for software engineering. IEEE Trans. Software Eng., 19: 856-862. doi: 10.1109/32.241769.
4. Dijkstra, E.W., 1975. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM, 18: 453-457. doi: http://doi.acm.org/10.1145/360933.360975.
5. Hoare, C.A.R., 1969. An axiomatic basis for computer programming. Commun. ACM, 12: 576-580. doi: http://doi.acm.org/10.1145/363235.363259.
6. Bicarregui, J., J. Dick, B. Matthews and E. Woods, 1997. Making the most of formal specification through animation, testing and proof. Sci. Comput. Program., 29: 55-80. doi: 10.1016/S0167-6423(96)00029-9.
7. Bowen, J.P. and M.G. Hinchey, 2005. Ten commandments revisited: A ten-year perspective on the industrial application of formal methods. Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, Lisbon, Portugal. DOI: http://doi.acm.org/10.1145/1081180.1081183.
8. IEEE, 1994. Standard glossary of software engineering terminology. In IEEE Software Engineering Standards Collection. IEEE Std 610:12-190.
9. Tse, Francis C.M., Lau, W.K. Chan, Peter C.K. Liu and C.K.F. Luk, 2007. Testing object-oriented industrial software without precise oracles or results. Commun. ACM, 50: 78-85. doi: 10.1145/1278201.1278210.
10. Harold, M.J., 2000. Improving software testing by observing practice. In: Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, June 4-11. ACM Press, New York, pp: 61-72. doi: 10.1145/336512.336532.
11. Linzhang, W., Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong and Z. Guoliang, 2004. Generating test cases from UML activity diagram based on gray-box method. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), Nov. 30-Dec. 03 Busan, Korea. Pp: 284-291. DOI 10.1109/APSEC.2004.55.
12. Vilkomir, S.A. and J.P. Bowen, 2006. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. Formal Aspects Comput., 18: 42-62. Doi: 10.1007/s00165-005-0084-7.
13. Liu, S. and Y. Chen, 2008. A relation-based method combining functional and structural testing for test case generation. J. Syst. Software, 81: 234-248. Doi: 10.1016/j.jss.2007.05.036.
14. Hoffman, D.M. and D.M. Weiss, 2001. Software Fundamentals. 1st Edn. Addison-Wesley Longman Publishing Co., Inc. ISBN:0-201-70369-6.
15. Moller, K.H. and D.J. Paulish, 1993. An empirical investigation of software fault distribution. Proceeding of IEEE 1st International Software Metrics Symposium, May 21-22. Baltimore, Md., pp: 82-90. Doi: 10.1109/METRIC.1993.263798.

16. Chen, H.Y., T.H. Tse, F.T. Chan and T.Y. Chen, 1998. In black and white: An integrated approach to class-level testing of object-oriented programs. ACM Trans. Software Eng. Methodol., 7: 250-295. Doi: http://doi.acm.org/10.1145/287000.287004.
17. Baresi, L. and M. Young, 2001. Test oracles. Technical Report CIS-TR-01-02, Dept. of Computer and Info. Science, University of Oregon, http://ix.cs.uoregon.edu/~michal/pubs/oracles.html.
18. Lamsweerde, A.V., 2000. Formal specification: A roadmap. Proceedings of the Conference on the Future of Software Engineering, June 04-11. Limerick, Ireland. Doi: 10.1145/336512.336546.
19. Parnas, D.L., J. Madey and M. Iglewski, 1994. Precise documentation of well-structured programs. IEEE Trans. Software Eng., 20: 948-976. Doi: 10.1109/32.368133.
20. Parnas, D.L. and Madey, J., 1995. Functional documentation for computer systems engineering. Sci. Comput. Program., 25: 41-61. Doi: 10.1016/0167-6423(95)96871-J.
21. Baber, R., D.L. Parnas, S. Vilkomir, P. Harrison, and T. O'Connor, 2005. Disciplined methods of software specifications: A case study. Proceeding of the International Conference on Information Technology Coding and Computing, April 2005. Doi: 10.1109/ITCC.2005.132.