A Baseline-based BIST Design Model for Software Testability

Junhua SUI, Jianxin WANG, Huina LIU, Liquan LIU
Research Institute of Security Printing of the People's Bank of China, Beijing 100070, China

ABSTRACT: Software testability design is an important approach of improving software testability. Based on the baseline model embodying the observability of data processing, a new effective software testability design model BBDM is proposed through using hardware BIST for reference. This model can improve software testability, reduce the whole cost of software life cycle, and make for the automation of data analysis and testing case generation.
KEYWORD: baseline; BIST; observability; software testability design

1 PREFACE

Increasing software testability can not only reduce software-testing cost but also ensure software quality [1-3]. To introduce testability mechanism at the design stage of software development can improve software system testability and the efficiency of fault tracking and detection, shorten testing cycle, reduce testing cost, and increase system reliability. Software testability design, one of important approaches to the improvement of software testability, has been gradually becoming a focus of software testing research.

According to the definition made by IEEE and Freedman [4], observability, which is one of essential features software testability has, denotes the difficulty degree of acquiring the testing information. The decrease of the difficulty degree means the increasing of software testability. How to perform software testability design in the visual angle of observability is a new research point. This is the motivation of this paper.

2 BASELINE MODEL

In view of the essential features (observability and controllability) of software testability, a directed input-BPS-output graph (DIBOG)[5] is used to describe and analyze the detailed relationships of inputs, basic processing slices (BPS)[5], intermediate results and outputs. Different layers of DIBOG describe different relationships. A DIBOG is very helpful for code developers and testers to get the information about processing procedure.

Baseline model[5] is a reduced DIBOG based on OPB[5] (output path baseline) and EOPB[5] (equivalent output path baseline)[5]. The goal of baseline model is to further describe and analyze the procedure of data processing, and lay a foundation for testability measurement. Baseline model can embody the observability and controllability of data processing. Therefore, it is reasonable to introduce the model into software testability design.

To learn about more information about DIBOG and baseline model, refer to [5], which contains their detailed definition and description.

3 BIST

BIST (Built-In Self-Testing) is one of most important hardware testability design technologies. A BIST circuit can generate testing codes (a code sequence) and detect the corresponding output and judge whether there exists a fault. Fig 1 describes the general structure of BIST, where, TPG and CUT respectively represent a testing pattern generator and a circuit under testing. The core of BIST is to do some additional work to decrease test complexity and improve testability [6][7].

Software testing is a procedure of comparing procedure temporary results or final outputs, which are generated through inputting assumed values into the execution procedure of the objects under testing, with corresponding expected values. Obviously, the essentials of both software testing and hardware BIST are consistent. Moreover, Liu Mujin[8] has discussed the consistency of software and hardware testing. So, when testability design is performed at the design stage, BIST can be used for reference.

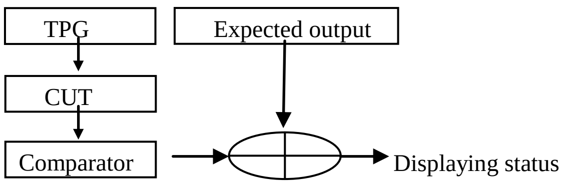

Figure 1. the general structure of BIST

4 CONSTRUCTION OF A BASELINE-BASED BIST DESIGN MODEL FOR SOFTWARE TESTABILITY

Software testability design is a procedure of introducing the testability mechanism into software development. This way can not only make it convenient for testers to perform software testing, but also provide a way to get internal testing information of the tested object. So, reasonable and effective testability design is the foundation of improving software testability successfully.

Based on the analysis of the sections above, a baseline-based BIST design model for software testability (abbreviated to BBDM) is put forward.

4.1 *Foundational Components and Logical Relationships*

Shown as figure 2, BBDM is composed of five primary parts: RTT (Reference Template for Testing), detecting-point design, detector construction, execution clue case-base (including the definition of comparing codes) and system comparator. The realization of the model must be based on the joint participation of system analyzers, code developers, and software testers.

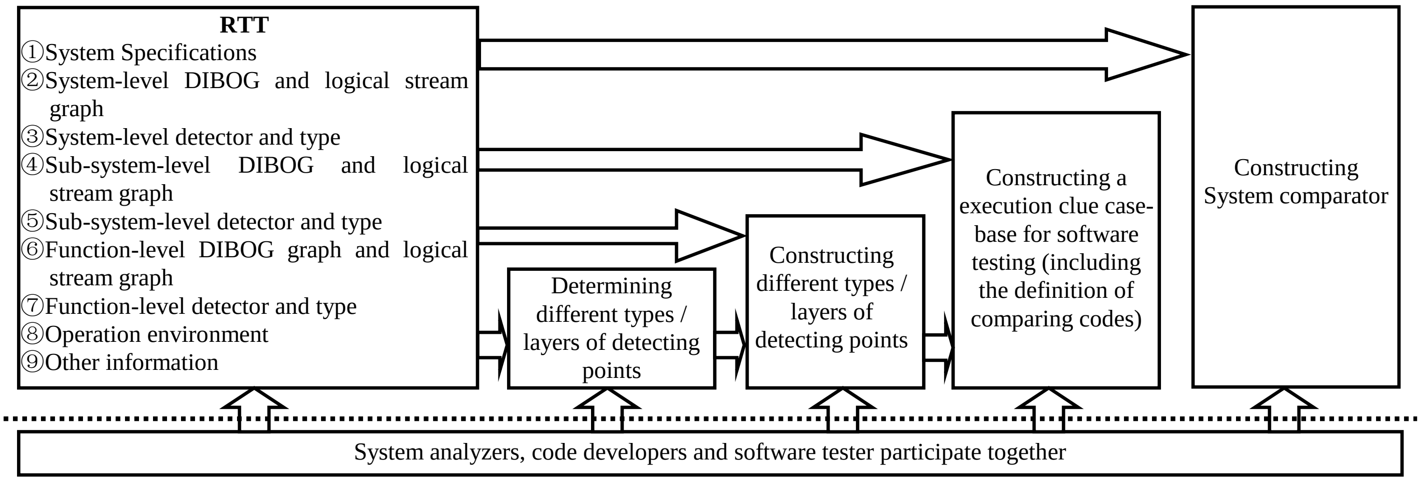
RTT
①System Specifications
②System-level DIBOG and logical stream graph
③System-level detector and type
④Sub-system-level DIBOG and logical stream graph
⑤Sub-system-level detector and type
⑥Function-level DIBOG graph and logical stream graph
⑦Function-level detector and type
⑧Operation environment
⑨Other information
Determining different types / layers of detecting points
Constructing different types / layers of detecting points
Constructing a execution clue case-base for software testing (including the definition of comparing codes)
Constructing System comparator
System analyzers, code developers and software tester participate together

Figure 2. the baseline-based BIST design model for software testability (BBDM)

Before the interior structure of each part is expatiated, their relationships should be first described. RTT is the kernel of the design model and can provide the foundation for designing detecting-points, constructing detectors, case-bases and system comparator. Figure 3 shows their logical relationships

Firstly, the reduction and equivalence of the baseline model is adopted to construct a system-level DIBOG graph and corresponding function-level DIBOG graphs, which can present the detailed relationships of inputs, BPS, intermediate results and outputs, and baseline metrics (such as BIAR (Baseline Input-element Annihilation Ratio)) is adopted to measure the testability of baselines.

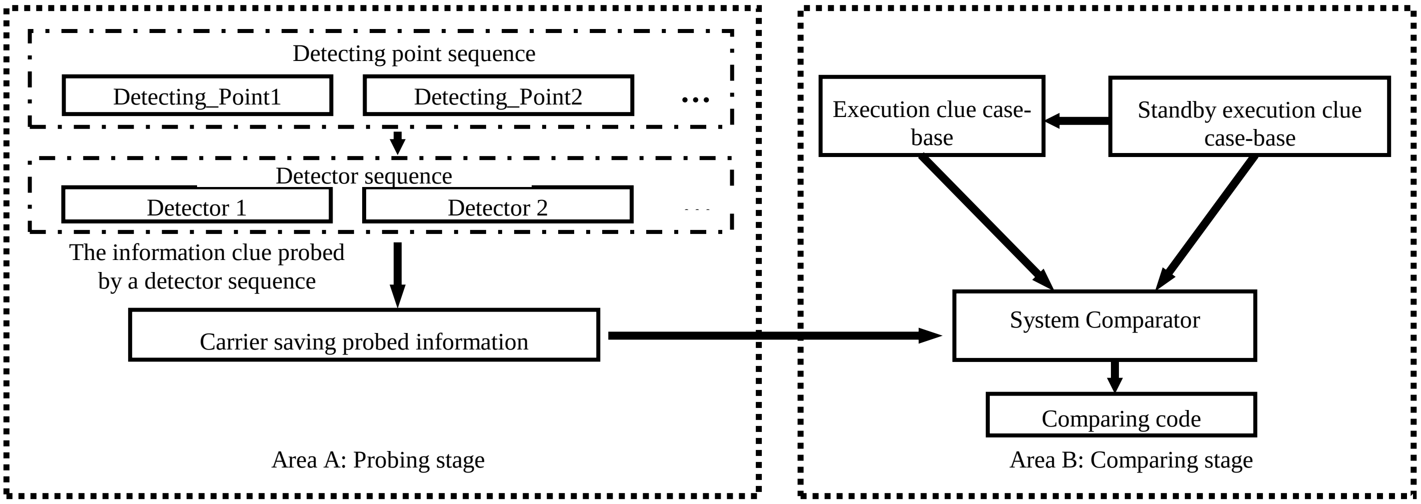

Figure 3. the internal logical structure of BBDM

Secondly, different layers of numerical-value detecting points are constructed on those baselines with highest BIAR. And at the same time, some logical-type detecting points are also constructed on backbone logical streams. Corresponding types of detectors are established according to different layers or types of detecting points on baselines or backbone logical streams. During system code development, these detectors, as components of the whole software system, are buried into the system. On the ground of designed testing cases, relevant input elements are entered into some baseline and the baseline is executed. Those detectors covered by the executed baseline compose a detector sequence. Each detector on the detector sequence detects relevant information near its detecting point and writes the information orderly into a carrier saving probed information. All the ordered information in the carrier composes baseline execution detection information (short in baseline detection information). All procedures above are described detailedly in area A of Figure 3. These procedures are full buried into the software system, which can not only make for system automation testing, but also realize dynamic information detection and collection during future system use and maintenance.

After the baseline detection information is wrote into the carrier, the system comparator loads the detection information from the carrier and the corresponding case execution clue from the Execution Clue Case-base, and compare them. If there exists a case execution clue, which is matched with the detection information, in the Execution Clue Case-base, a comparing code can be generated; or else, the detection information is recorded into the standby execution clue case-base. If necessary, the detection information can serve as a new case execution clue and be moved from the standby execution clue case-base to the execution clue case-base. All procedures above are also described in detail in area B of Figure 3. The comparing procedure can not only be buried into the software system, but also be independently developed as an analyzing tool. This, like the function of Area A, can make for automation testing and dynamic analysis.

In the model above, an execution clue case-base serves as the same function as the expected input part of hardware BIST structure, a system comparator serves as the same functions as the comparator of hardware BIST structure, a detecting point is equivalent to a hardware Checkpoint, and a detector is equivalent to a Check circuit that is added on Checkpoint. If, during software development, the functions of Areas A and B are buried into the software system, the software system, like a hardware circuit, has the BIST property, which is very helpful to improve testability and reduce testing cost. Therefore, the model brought forward in this paper is regarded reasonable.

4.2 *Determination of Detecting Layers and Detecting-Points*

How to determine detecting layers and detecting points is very important for code developers and testers to trace, collect and analyze related testing information.

4.2.1 *Detecting Layer*

BBDM can, on the basis of baseline layers of different DIBOG graphs, adopts a hierarchical structure for the construction of detecting layers. This can enhance software testability and make it convenient for software testers to get hierarchical detection information of the software system adopting the method.

Generally, four layers of detecting layers can be established according to functional areas. These layers have the similar functions----to make detectors on them respectively trace, collect and analyze the corresponding information. The information on a system detecting layer (SDL) covers the interfaces among different sub-systems, the information on a sub-system detecting layer (SSDL) covers those functions that together realize a common service, the information on a function detecting layer (FDL) covers the interfaces among those modules that together realize a common function, and the information on a module detecting layer (MDL) covers data processing in a model. Moreover, for the object-oriented design, two additional layers might be added: inter-class detecting layer (Inter-CDL) and inner-class detecting layer (Inner-CDL). The former contains the interface information among classes, and the latter contains the interface information among methods in a class.

In practice, there exists no strict rule for layer classification. A layer classification can be made according to concrete functions/services and realization modes, which is determined by practical requirements.

4.2.2 *Detecting Point*

The goal of setting a detecting point is to make it convenient for a detector to trace, collect and save some related information near the detecting point. By virtue of key properties of software testability, a detecting point can be set at the following positions: (1) a baseline with high FIAR[5]; (2) around a BPS with many inputs; (3) around a control of stream direction. Additionally, a detecting point can be set according to fault distribution of different types of software systems.

4.3 *Foundational Construction of Primary Components in BBDM*

After the mutual relationships of five components in BBDM are described, foundational structure of each component is constructed in brief as follows. And the detailed construction can refer to [9].

4.3.1 *RTT*

It is system analyzers, code developers and software testers who together participate in the construction of RTT and perfect it increasingly. The RTT is composed of 9 parts listed in figure 2. System specifications serve as the basis of system design and software testing, and the basic function of the other parts can refer to section 4.3.2.

4.3.2 *Classification and Construction of Detectors*

According to the return value probed by a detector, there exist two kinds of detectors: logical-value detector (LD) and numerical-value detector (ND). And according to the hierarchical structure of detecting layers, there exist at least two kinds of detectors: system-level detector (SD) and function-level detector (FD). If necessary, a BPS interior-detector (BID) could be added.

The foundational function of a detector is to probe and transport the relevant value of an object under test. So, the basic structure of a detector should include two parts: detecting part and transporting part, and a detector can be constructed easily no matter structural language or object-orient language is adopted. In view of information security, a detector could contain an on-off that can be used to enable or disable its detecting function.

4.3.3 *Construction of a System Comparator*

A comparator is mainly used to compare the detection information from the carrier with the corresponding case execution clue from the Execution Clue Case-base, and generate a comparing code, which can make preparations for auto-tracing and auto-testing.

Obviously, the comparator should support the following functions:

(1) Records a detecting clue, which can help to perform the tracing and comparison of sensitive information on detecting points, locate faults, and make for execution playback and on-spit data analysis.

(2) Discloses some latent faults.

(3) Corrects and enlarges a case base.

5 MODEL VERIFICATION

To verify that BBDM can increase software testability and reduce development cost, [9] made a contrast experiment on a simple bank ATM system. Only experiment results are listed in table 1, and the other detailed information of the experiment can refer to [9].

Table 1 experiment results of a contrast experiment

Number	Item	Without BBDM and detectors	Adopting BBDM and adding detectors
1	Code lines	3700	4100
2	Workload (days)	14	13
3	Number of disclosed faults (errors)	12	15
4	Testing cases	27	31
5	Occupied memory (K)	7174	*7301*
6	Time delay (ms)	—	*4*
7	Observability	Poor	Good
8	Tracability	Don't support	Support
9	Auto-testing	Don't support	Support
10	Testability	Poor	Good

The experiment results listed in Table 1 represent: after BBDM is adopted at the stage of software system design and some detectors are buried at the stage of code development, the ability of disclosing faults (errors) and the automatization of testing case generation and functional testing can be improved, testing case base can be enlarged, testing coverage can be increased, and the final progress is ahead of that without using the model. These results profit from the improvement of software system testability, but this is at the cost of more memory and time delay.

The contrast experiment indicates that the BBDM is reasonable and helpful for some software systems that are not sensitive to realtime and memory source.

6 CONCLUSIONS

BBDM is an effective software testability design method. It can improve software testability, reduce the whole cost of software life cycle, and make for data tracing & analysis and the automation of testing case generation and software testing.

The addition of detectors can result in the increasing of time delay and occupied memory. This might make an influence on those software systems sensitive to realtime and hardware resources. Thereby, once BBDM is adopted for these systems, it is indispensable to compute and evaluate the system influence resulting from the operation of added detectors. The quantitative analysis of the influence should be the research focus for the future.

REFERENCES

[1] Jeffrey Voas,Keith W Miller. Software Testability: The New Verification. IEEE Software, 1995; 12.

[2] S. C. Gupta, Impact of software testability considerations on software development life cycle. 1995.

[3] Pu-lin Yeh, Jin-Cherng Lin. Software Testability measurements derived from data flow analysis. Proceedings of Software Maintenance and Reengineering, 1998: 96~102.

[4] Roy S. Freedman, Testability of Software Components. IEEE Transactions on Software Engineering, Volume 17, Issue 6 (June 1991): 553 ~ 564.

[5] Sui Junhua, Research on Software Testability Methods& Technologies, Graduate University of Chinese Academy of Sciences, 2005. 3: 55~70.

[6] Savaria Y. A Pragmatic Approach to the Design of Self-testing Circuits [A].1989 Int Test Conf. Washington: IEEE Pub, 1989. 745~754.

[7] Venkataraman S. An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers. Proc of International Conference on Computer-Aided Design: [s.n.], 1993, 572~577.

[8] Liu Mu-Jin, Xu Shi-Yi, Unity between Hardware and Software Testing. Journal of TongJi University (Natural Science), Vol. 30 No. 10, Oct. 2002:1186~1189.

[9] Sui Junhua, Research on Software Testability Methods& Technologies, Graduate University of Chinese Academy of Sciences, 2005.3: 95~115.