Software System Testing Method Based on Formal Model

Weixiang Zhang, Wenhong Liu, Bo Wei
Beijing Institute of Tracking and Telecommunications Technology
Beijing, China
e-mail: wxchung@msn.com

***Abstract*—To do software testing of a large-scale and high-complexity security-critical software system is a difficult task in engineering practice. In this paper, a software system testing method based on formal model is proposed. Firstly, the software function, performance and interface were abstracted, and gave the formal definition of software system. Secondly, different strategies of cue analysis were proposed to identify the requirements of software system testing. Then, the process, the stage and the scene analysis of the software system were carried out, and the scene tree model which could describe the system level behavior of the software was established. Finally, the test coverage criterion based on the scene tree model was proposed, and formal static checking and dynamic test cases generation method were given to generate test case set. Experiments show that the method is effective and feasible.**

Keywords-system testing; formal model; test case generation; scenario modeling; software testing

I. INTRODUCTION

With the increasing popularity and deepening of software applications, software has become the main enabling component in more and more systems [1]. As the scale and complexity of the software system increases dramatically, the quality of software becomes more and more eye-catching.

Software testing is one of the most important means to guarantee the software quality. Software system testing (or system testing) and unit testing, integration testing, configuration testing (also known as software qualification testing or validation testing), acceptance testing is called five major different test categories [2]. System testing is a test behavior on a complete and integrated system that is used to evaluate the compliance of the system with defined requirements [3].

The test object of the system test is a complete software system, which contains dozens of software configuration items with different critical grades. The software system usually has many functions, complicated interfaces and performance indexes difficult to measure [4]. For this reason, system testing is more difficult than configuration item testing. But the existing technical methods and research results focused on configuration items testing, study on software system testing is not enough. It is lack of useful test strategies and methods on system testing in practice [5].

Formal modeling uses a language with strict mathematical definitions of semantics and syntax to characterize the software and its behavior and to describe its behavior patterns to ensure its correctness [6]. Formal verification based on formal modeling, through model checking and other means to analyze and verify whether the software has the desired properties, and in line with a given behavioral model. Vassev et al. [7] used a self-help system specification language, ASSL, to establish a formal model for NASA tasks and to generate functional prototypes. Zhu Yan et al. [8] established a comprehensive avionics system stochastic Petri net system model, and based on the model to carry out the performance analysis. Bieber et al. [9] Validated the reliability and safety of the Airbus A320 hydraulic subsystem using the Altarica model.

The model-based software testing method can improve the degree of automation of test case generation, partly solve the problem of test failure identification, and can easily analyze the test results and evaluate the test results, which is beneficial to the reuse of testing process [10]. By abstracting the behavior and structure of the software and describing it in an easy to understand way, the model of the software system can be obtained, which can be used to generate the test cases. Common software testing models are finite state machine [11], Markov chain [12], UML [13], grammar model [14]. Formal methods have always been controversial [15], mainly due to the relatively high cost of software formalization, which makes it impractical to implement full formalization in large software systems. However, it is an effective compromise to introduce formal methods in key areas or critical services.

In this paper, a test method of measurement and control software system based on formal model is proposed. First, formal definitions of software functions, performance, interfaces, and software systems are given. Secondly, the paper presents the test strategy of using the clue analysis to identify the function and scene of the software system, and further establish the formal model of the scene tree of the software system. The behavior of the model can represent the system-level behavior of the software system. At last, the test coverage criterion of software system based on formal model is proposed, and formal static checking and dynamic test case generation method are given. In this paper, an instance analysis and method description are given based on a security decision software system. At the end of this paper, the example verification and analysis results are present.

II. FORMAL MODEL OF SOFTWARE SYSTEM

A. Formal Basis

Software systems are generally composed of multiple software configuration items, there are data exchange or interoperability between them, they complement each other to achieve system-level functions, and to meet the required performance and other quality requirements.

Definition 1 $FUNC = (func\_Name, IO_{in}, IO_{out}, RS_{in}, RS_{out})$ is called a software function, where $func\_Name$ is the

function name, $IO_{in}=\{pi_1,pi_2,\cdots,pi_m\}\ (m\geq 0)$ is the function input, $IO_{out}=\{po_1,po_2,\cdots,po_n\}\ (n\geq 0)$ is the function output, pi_i is the input parameter, po_j is the output parameter, RS_{in} is the input constraint set, RS_{out} is the output constraint set.

Definition 2 $PERF=(perf\_Name,\{(c,cv)\})$ is called a software performance, where $perf\_Name$ is the performance name, $(c,cv)\in C\times CV$, $C=\{c_1,c_2,\cdots,c_n\}$ is the performance index set, $CV=\{cv_1,cv_2,\cdots,cv_n\}\ (n\geq 0)$ is the performance value set, cv_i can be expressed as $[\min,\max]$, where min is the lower limit, and max is the upper limit.

Definition 3 $\mathrm{INTF}=(\mathrm{intf\_Name,sour,dest,pt,DT})$ is called a software interface, where intf_Name is the interface name, sour is the interface source, dest is the interface destination, pt is the protocol type of the interface, $DT=\{dt_1,dt_2,\cdots,dt_n\}\ (n\geq 0)$ is the collection of data elements of the interface.

Definition 4 $CONF=(conf\_Name,\{(a,av)\})$ is called a software configuration, where $conf\_Name$ is the configuration name, $(a,av)\in A\times AV$, $A=\{a_1,a_2,\cdots,a_n\}\ (n\geq 0)$ is the collection of attribute names of the configuration, $AV=\{av_1,av_2,\cdots,av_n\}\ (n\geq 0)$ is the collection of attribute values of the configuration.

Definition 5 $CSCI=(\mathrm{csc}i\_Name,FUNC^{\mathrm{csc}i},PERF^{\mathrm{csc}i},INTF^{\mathrm{csc}i},CONF^{\mathrm{csc}i})$ is called a software configuration item, where $\mathrm{csc}i\_Name$ is the name of the software configuration item, $FUNC^{\mathrm{csc}i},PERF^{\mathrm{csc}i},INTF^{\mathrm{csc}i},CONF^{\mathrm{csc}i}$ is the collection of functions, performances, interfaces, and configurations of the software configuration item respectively.

Definition 6 $SS=(ss\_Name,FUNC^{ss},PERF^{ss},INTF^{ss},CONF^{ss},CSCI)$ is called a software system, where ss_Name is the name of the software system, $FUNC^{ss},PERF^{ss},INTF^{ss},CONF^{ss}$ is the collection of functions, performances, interfaces, and configurations of the software system respectively, $CSCI$ is the collection of software configuration items contained in the software system. It should be noted that the software system contains a collection of software configuration items, but also has its own independent function, performance, interface and configuration.

B. Clue Analysis

Compared to software configuration items, the functions, performances, interfaces, and configurations of software systems are often not obvious, especially in the absence of system-level specifications. They are implicitly distributed in the specifications of the various software configuration items or their interface documents.

The difficulty of requirement analysis of software system testing is to identify the function, performance, interface and configuration in the software system level. Clue analysis method is helpful in solving this problem [16].

Definition 7 ST is called a system clue, which refers to an instance of a mapping that is observable at the software system level from the system input to the system output. In a broad sense, the software system itself is a mapping from input to output, although the content of this mapping may not be known to testers.

There are four strategies to carry out clue analysis to a software system in usual:

1) Clue analysis based on inputs. The so-called input here includes input data and user operation of the system interface or input port. Design some input coverage metric, such as traversing the various input combinations of each interface (it is hard to implement), and then design system inputs to identify system functions and scenes.

2) Clue analysis based on outputs. The so-called output here includes output data and display information of the system interface or output port. Design some output coverage metric, such as traversing the output of each system, and then according to the metric design the corresponding system input to identify the system functions and scenes.

3) Clue analysis based on interface or port. Determine what operations （or data） will occur in each interface (or port), and then look for clues from the list of operations (or data). In a sense, input-based clue analysis covers one-to-many relationships from input to user interface, and interface-based clue analysis covers one-to-many relationships from interface to input, which can be seen as complementary to the former.

4) Clue analysis based on data. Design the coverage metrics for the data, and then perform clue analysis based on these metrics. Data-based clue analysis focuses on transformational (rather than reactive) systems, which support database-based transaction processing. Using these data and their relationships, data-based clue analysis can yield many interesting clues.

Clue analysis can effectively help to identify system-level functions, performances, interfaces and configurations, and then establish the scene tree.

C. Scene Tree Model

Software system is in a known state at a given time, it calls one or more system functions, and through them to connect one or more system interface, that is, opens a scene of the software system.

Existing studies focus on the use of UML to establish the scene. Ryser [17] used the annotated UML state diagram for scene modeling, introduced relational tables to represent the dependencies between scenarios, and found the required set of test cases by traversing. Lettrari [18] extended the UML timing diagram to provide the Rhapsody UML model prototyping tool to support the modeling of the scene. Tasi [19] used the sequence of events, activities and associated pre / postconditions to model the scene and generated test cases in a random way. Marchetti [20] studied two examples of medical systems and human resources systems, used UML time series diagram for modeling, gave the test case

generation algorithm and found some defects in the software systems.

The scene tree is the core of our formal model. Before the scene tree, the definitions of scenes and events are given first.

Definition 8 $A=(FUNC_t)$ is called a scene when the software system is running, where $FUNC_t$ denotes the set of $FUNC$ that can be called when the software system runs to time t, the set can contain 0, 1 or more $FUNC$.

Definition 9 $E=(Event\_Name, PC, FUNC, PF)$ is called an event that triggered the scene switch, where $Event\_Name$ is the name of the event, PC is the set of preconditions of the event, and when the precondition is true, the event is triggered. $FUNC$ is the function set triggered by the event, PF is the corresponding parameter set.

The execution of a software system can be seen as an orderly transition of a large number of scenes, one scene switch to another scene via event triggering. In the process of scene switching, due to different parameters of the event would call different functions and thus have different flow direction, it is true that using the scene tree to describe the scene and its switching relationship. When the initial scene is known, the execution of the software system can also be seen as an ordered sequence of events.

Definition 10 $AT=(A, a_0, E, R_{A,E})$ is called a scene tree that describes each scene and its switching relationship, where A the collection of scenes, $a \in A$ is a scene of the software system, called a node of the scene tree, $a_0 \in A$ is the initial scene, E is a collection of events that contain events that trigger a scene switch, $R_{A,E} \subseteq A \times A \times E$ is the switching relation set of the scene. If there is a switching relationship $(a_p, a_c, e) \in R_{A,E}$ to switch from scene a_p to scene a_c, then a_p is the parent of a_c, a_c is the child of a_p, and e is the event that triggers the scene switch.

Below will take the geographical security control software system (GSCS) as an example, introduces the scene tree establishment process. Firstly, introduce briefly the software system functions and operation flow.

The main purpose of GSCS is to provide geographical information such as terrain, landform, city, population, transportation hub and important facilities, and display the flight status, space situation, security situation, attitude change and movement trend of the aircraft flight process. Its role is to provide decision-making basis for program development, security control implementation, situation control and so on. GSCS includes six software configuration items, including geographic information storage and management (GSM), experiment information storage and management (ESM), simulation and analysis (SAA), security control decision and processing (CDP), real-time data transceiver (RDT) and integrated situation display (ISD).

The main operation flow of GSCS is: Firstly, GSM provides geographical information, ESM provides experiment information, SAA generates the theoretical safety pipeline data, and accomplish the initial configuration of the software system. Then, after the configuration is complete, GSCS enters the real-time processing state, RDT begins to receive and transmit real-time data. Once receiving real-time data, the data is processed by CDP, and then the processing result is sent to ISD for display and ESM for storage. Finally, after the end of the mission, SAA carries out post-test data analysis.

According to the operation flow, the operational flow chart of GSCS is drawn as shown in Figure 1, and then the stage analysis is carried out to establish the stage analysis chart, as shown in Figure 1.

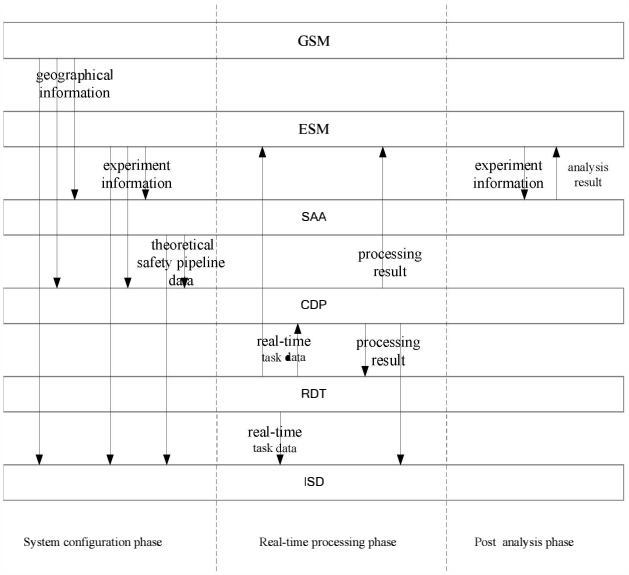

Figure 1. The stage analysis chart of GSCS

Combined with clue analysis, based on the operational flow chart and stage analysis chart, draw the scene tree chart of GSCS, as shown in Figure 2.

III. MODEL-BASED SOFTWARE SYSTEM TESTING

A. Static Configuration Consistency Inspection

The configuration consistency inspection is a means of confirming the configuration information of the software system. The purpose is to ensure that the current configuration information is consistent with the task or software requirements. The configuration consistency inspection is usually carried out before the dynamic test or can be carried out separately [6].

The configuration consistency inspection is performed by comparing the configuration information of the system with the given standard configuration to check whether it conforms to the configuration consistency requirements.

For two configuration $conf_1 = (confName_1, \{(a_i, av_i)\})$ and $conf_2 = (confName_2, \{(a_j, av_j)\})$, where $(a_i, av_i) \in A_1 \times AV_1$ and $(a_j, av_j) \in A_2 \times AV_2$, if they meet the following four conditions simultaneously, we call $conf_1$ and $conf_1$ are consistent (abbreviated as $conf_1 == conf_2$): (1) $confName_1 = confName_2$; (2) $A_1 = A_2$; (3) $AV_1 = AV_2$; (4) for any

$(a_i, av_i) \in A_1 \times AV_1$, there is $(a_j, av_j) \in A_2 \times AV_2$, satisfy $a_i = a_j, av_i = av_j$. Otherwise, $conf_1$ and $conf_1$ are called unconsistent, abbreviated as $conf_1 != conf_2$.

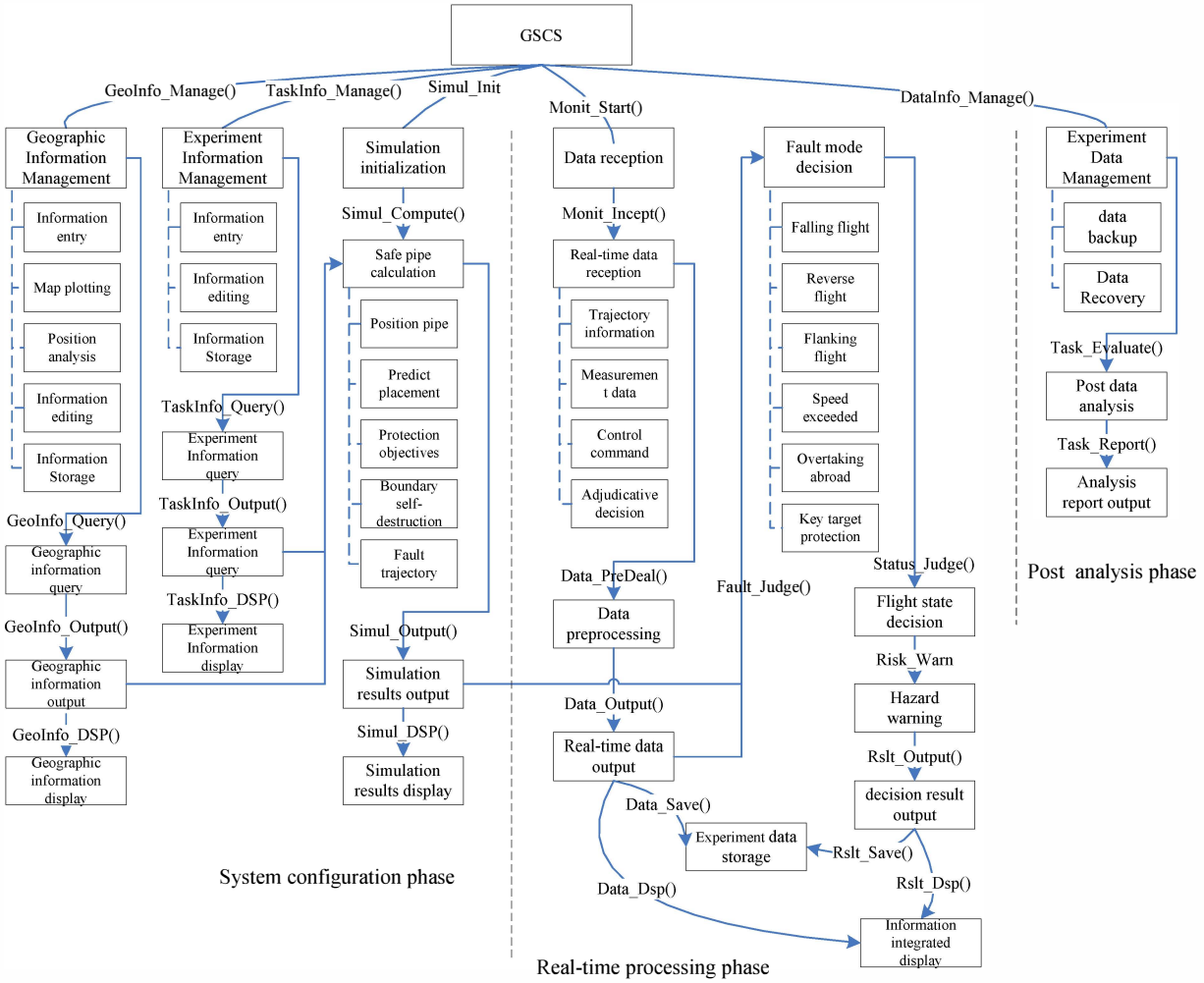

Figure 2. The scene tree chart of GSCS

The process of configuration consistency inspection is shown in Figure 3, where the standard configuration set is a pre-stored configuration data set containing complete and standard software configuration information.

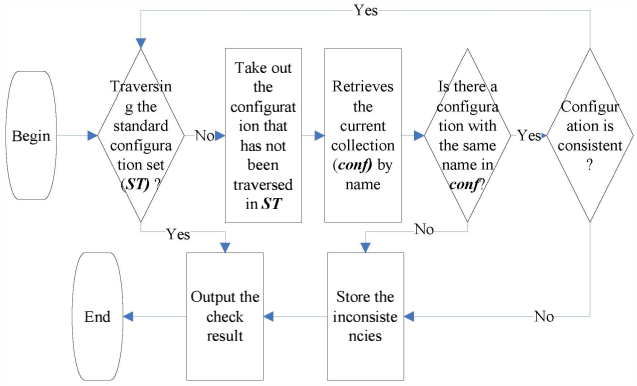

Figure 3. The flow chart of static configuration consistency inspection

Configuration consistency inspection is a static test method, which can provide a reliable static environment for the software system and guarantee the validity and correctness of the dynamic test results.

B. *Dynamic Test Case Generation*

Based on the path-based coverage test criterion, the coverage criterion of software system testing based on formal model is proposed. First, give the definition of coverage.

Definition 11 Scene a is covered by test case set T , if for any function contained by a , there is a subset $T' \subseteq T$, satisfy the test sufficiency requirement for the function.

Definition 12 Event e is covered by test case set T , if for any scene cutover that triggered by e , there is a subset $T'' \subseteq T$, satisfy the test sufficiency requirement for the scene cutover.

For a software system SS, the set of test cases T generated based on its scene tree AT should satisfy the following criteria:

1) (Node Coverage Criterion) T satisfies the node coverage criterion, if and only if for any scene a (ie, nodes, leaves) in AT, there is a subset $T' \subseteq T$, make it is true that a is covered by T'.

2) (Edge Coverage Criterion) T satisfies the edge coverage criterion, if and only if for any event e (ie, the edge, the branch between each node) in AT, there is a subset $T' \subseteq T$, make it is true that e is covered by T'.

It is easy to prove that the test case set which satisfied node coverage and edge coverage criterion satisfies the following path coverage criterion:

The test case set T satisfies the path coverage criterion if and only if for any path *route* (including the edges between any two nodes and all nodes) from the root node (i.e., the starting scene) to the leaf node in the scene tree, all the scenes and events in *route* are coved by T.

As mentioned earlier, the implementation of the software system can be seen as an ordered sequence of scene, and can also be seen as an ordered sequence of events. Therefore, the depth-first traversal method is used to generate test cases. Its main idea is: for any node (i.e., scene a)in the scene tree, design a set of test case that cover the corresponding path of the node. If the node is a leaf node, add this set of test cases to the total test case set, and set the state of the node to be traversed; otherwise, select a child node a_1 of a that has not been traversed, proceed with depth-first traversal from a_1.

IV. EXAMPLE VALIDATION

The method proposed in this paper is verified by the example of Geographic Security Control System (GSCS). The main functions of GSCS have been introduced, its operational flow chart and phase analysis chart and scene tree chart have been drawn, so here will not repeat them.

Here give the software system testing data. Based on the scene tree chart of GSCS, 252 test cases were designed according to the node coverage and edge coverage criteria and 57 software defects were found. The distribution of test cases and defects are shown in Table I, the defect classification statistics shown in Table 2.

TABLE I. TEST CASE DISTRIBUTION LIST

Test Case Concerns	Test Case Count	Test Case Proportion	Defect Count	Defect Proportion
GSM	48	19.05%	16	28.07%
ESM	21	8.33%	8	14.04%
SAA	27	10.71%	16	28.07%
CDP	32	12.70%	0	0.00%
RDT	22	8.73%	7	12.28%
ISD	36	14.29%	4	7.02%
Other (integrated process, system interface, performance, etc.)	66	26.19%	6	10.53%
Total	252	100%	57	100%

TABLE II. SOFTWARE DEFECT CLASSIFICATION STATISTICS TABLE

Grade / category	1	2	3	4	5	Total
Program error	0	0	6	25	4	35
Document error	0	0	0	2	0	2
Design error	0	0	1	11	4	16
Else	0	0	0	2	2	4
Total	0	0	7	40	10	57

From Table I, we can see that: (1) Except GSM, the distribution of test cases is roughly equal to the proportion of each function of GSCS. (2) The proportion of the test cases of GSM is too large, because there is more input, plotting, editing and other human-computer interaction, and according to the strategy of clue analysis, these human-computer interactions should be tested. (3) The proportion of test cases in other aspect (integrated process, system interface, performance, etc.) is close to three percent, slightly lower than the expected value, mainly due to a relatively large proportion of independent functions of GSCS, such as system configuration functions of GSM, ESM and SAA.

From Table I and Table II, it can be seen that: (1) The number and distribution of defects are in line with expectations. Most of the functions of CDP and ISD is reformed from existing software's, so there are few defects because these are relatively mature. In contrast, most of the functions of GSM and SAA are new developed, so there are more defects. (2) Defects in other aspect (integrated process, system interface, performance, etc.) are less, because GSCS exchange information through the network data frame, and part of the defects has been included in the defects of RDT. (3) The defect grade of GSCS is generally low, and there are no serious defects of grade 1 / grade 2, because the core function of the system is inherited from the existing software. (4) From the category of defects, the proportion of program error is more than 60%, and the design error is close to 30%, which indicates that the system should be improved from the aspects of software design and code realization. These is agreed with testers' intuitive feel on configuration management, process control and other aspects of GSCS.

Example validation results show that the formal technique proposed in this paper can establish the scene tree model of the software system. The model can help the tester to know the system behavior, identify the system function, performance and interface, and then help them to generate the software test suite which satisfies the test coverage criterion. Test case execution results and software defect distribution were in line with expectations. The effect of GSCS in engineering tasks also confirmed the feasibility and effectiveness of the software system testing method proposed by this paper.

V. CONCLUSION

In this paper, a test method is proposed based on a security decision system based on geographic information. The method builds the scene tree model of the software system by formal modeling and cue analysis method, and

then uses the scene tree model to automatically generate the test case set, which can satisfy the requirement of software system testing in a certain range.

The use of formal description and formal model to carry out software system testing has the advantages of clear testing process, easy to determine test adequacy criteria, applicable static detection and easy automation. At the same time, the software formalization method will bring a relatively large test cost in the test requirements analysis and test design. Therefore, the introduction of formal methods for the key property or key business of the system is a reasonable and effective test method. The next step, we will continue to carry out research on formal modeling of software systems and automated model validation.

REFERENCES

[1] Alessandro Orso, Gregg R. Software testing: a research travelogue (2000-2014). FOSE'14, Hyderabad, India, 2014, 117-132.

[2] China National Standardization Administration Committee. GB/T 15532−2008, Computer Software Testing Specification. 2008.

[3] China National Standardization Administration Committee. GB/T 11457 − 2006, Information technology - Software engineering terminology. 2006.

[4] JIA Yongnian. The Application of Computer in Measurement and Control Network. Beijing: National Defense Industry Press, 2000.4, 127-140.

[5] ZHANG Weixiang, LIU Wenhong. Application of Grey-Box Testing Method. Journal of Spacecraft TT&C Technology, 2010, 29(6):86-89.

[6] Li R, Lian H, Ma SL, Li T. Avionics system testing based on formal methods. Journal of Software, 2015, 26(2): 181-201.

[7] Vassev E, Hinchey M. Developing experimental models for NASA missions with ASSL. In: Proceedings of the Workshop on Formal Methods for Aerospace(FMA). EPTCS 20, 2010: 88-94.

[8] Zhu Y, Geng XT, Gao XG. The modelling and analysis of integrated avionics system based on stochastic Petri net. Fire Control and Command Control, 2006, 31(1): 41-44.

[9] Bieber P, Castel C, Seguin C. Combination of fault tree analysis and model checking for safety assessment of complex system. In: Proceedings of the 4th Europe Dependable Computing Conference. LNCS 2485, 2002: 19-31.

[10] Dsilva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(7): 1165-1178.

[11] Rosaria S. Robinson H. Applying models in your testing process. Information and Software Technology, 2000, 42(12):815-824.

[12] Whittaket J A, Thomason M G. A Markov chain model for statistical software testing. IEEE Transactions on Software Engineering, 1994. 20(10): 812-824.

[13] Zhang Y, Chi ZX. System test based on UML[J]. Computer Engineering and Design, 2006,27(9) :1634-1636.

[14] Maurer P M. The design and implementation of a grammar-based data generator. Software Practice & Experience, 1992. 23(3): 223-244.

[15] Woodock J, Larsen PG, Bicarregui J, Fitzgerald J. Formal methods: Practice and experience. ACM Computing Surveys(CSUR), 2009, 41(4): 1-40.

[16] Paul C J. Software testing: a craftsman's approach. Third edition. Beijing: Posts & Telecom Press. 2011.3. 184-208.

[17] Ryser J, Glinz M. Using dependency charts to improve scenario-based testing. In: Proceedings of the 17th International Conference on Testing Computer Software. 2000. 46-57.

[18] Lettrari M, Klose J. Scenario-Based monitoring ans testing of real-time UML models. In: Proceedings of the 4th International Conference on the Unified Modeling Languages, Concepts, and Tools. Berlin: Springer-Verlag, 2001, 317-328.

[19] Tsai WT, Yu L, Saimi A, Paul R. Scenario-Based object-oriented test frameworks for testing distributed system. In: Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems. IEEE Press, 2003, 288-294.

[20] Marchetti E, Schilders L, Winfield S. Scenario-Based testing applied in two real contexts: Healthcare and Employability. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validatio. IEEE Press, 2011, 89-98.