DOI : 10.3745/JIPS.2008.4.1.033

Journal of Information Processing Systems, Vol.4, No.1, March 2008 **33**

An Experiment of Traceability-Driven System Testing

Eun Man Choi* and Kwang-Ik Seo*

Abstract: Traceability has been held as an important factor in testing activities as well as model-driven development. Vertical traceability affords us opportunities to improve manageability from models and test cases to a code in testing and debugging phase. This paper represents a vertical test method which connects a system test level and an integration test level in testing stage by using UML. An experiment how traceability works to effectively focus on error spots has been included by using concrete examples of tracing from models to the code.

Keywords: *Software Engineering, Software Testing, Traceability, UML-based Testing*

1. Introduction

We often observe the cases in which paying heavy cost to testing and maintenance due to the small errors induced by mistakes. If traceability between different views and levels of abstraction are not provided then their testing or maintenance requires a great effort. Those cases suggest that great expenses should be produced after completion of a software system as well as under construction. In the context of testing and maintenance activities, we often look over model to the code back and forth to figure out error spots.

At present the requirement or design specification is used broadly for a software system test[1]. In requirements and design stage, we analyze these artifacts to check errors because the artifacts have faults. In testing stage, we test a system by validation and verification.

However, the need of short time-to-market requires that staged testing activities are more tightly connected by traceability. If testers found faults in system level testing, sometimes they need to look down the code level behalf of just noticing faults to unit authors. Developers have to modify source code bearing errors to make correctness after finding out faults in testing stage. It is difficult to inspect a logical structure and algorithm of source code because most of test methods based on UML are black box testing style. Therefore, it is hard to grasp the error spot and trace a code line. Moreover, both test team and development team need to spend lots of cost and time to communicate about errors in case that test team or quality assurance team and development team work separately. When test team executes testing based on UML specification and finds any faults they should explain not only those faults but also various data and process used to find faults to developers. That means that it needs lots of effort to communicate each other. In cases of this, if it is possible to associate the relation between test cases and source code they can decrease costs and effort of communicating to correct errors although they work separately.

Testing and verification of a system based on UML specification are classified in two types, before and after implementation. The first one is UML verification methods to judge the correctness of UML specification itself in analysis and design phase before implementation. The second one is a method of testing a system based on UML specification after implementation. UML design test is contributed to save money due to the early detection of errors. But only testing design specification is not sufficient to verify and confirm the software's detail functions and implementations.

Model based system testing also has the limitation that can not cover the detail in syntactic level and microscope functions of a system under test. It is useful to show the correctness of a system in user requirements level. But it has difficulties to locate error spots in code level and trace from design to implementation. Error types of mismatching with models and code would be the most critical weakness of model-based system testing.

System testing needs to be more rigorous by vertical trace from system to unit level. Detection of faults in system testing should be flowed fast by tracing location of errors in more detail level down. This paper proposes a rigorous approach that designs test cases about system functions by applying the proposed test method and studies test methods to trace the relation between source code and test cases. The proposed test method in this paper supports to find error spots by tracing between test cases, UML artifacts, and source code.

2. System Testing and Traceability

System testing is concerned with testing an entire system based on its specification. The work presented in this

Manuscript received October 23, 2007; revised January 10, 2008; accepted February 28, 2008.

Corresponding Author: Eun Man Choi

* Dept. of Computer Engineering, Dongguk University, Seoul, Korea (emchoi@dgu.ac.kr, bradseo@dgu.ac.kr)

section addresses system testing based on test cases derived from UML models and traceability research up to the present.

2.1 Specification Based System Testing

Braind and Labiche described the TOTEM methodology that derives test requirements from UML use case diagrams, interaction diagrams for each use case, class diagram with OCL constraints, and data dictionary describing classes, attributes, and methods[2][3]. Abdurazic and Offutt described a set of test requirements based on collaboration diagrams in which all the messages must be sent at least more than one times[4]. Also they proposed a technique for generating test cases from a restricted constraints of UML state diagrams[5]. Briand et al. enhanced this approach to support call and signal events, and various types of actions. A restricted form of UML class diagrams is used by Sheetz et al. to generate system test inputs. This approach converts a set of test objectives derived from class diagrams into test input sets. This approach ignores the details, such as generalization-specialization relationship between classes, that are present in class diagrams[6]. Hartmann et al. described an approach for testing distributed components[7].

Most UML based testing methods mentioned above have focus on functions or system tests. In other words, those test only system functions or interactions of modules and they have insufficient information about the relation between testing targets and source code. It makes difficulty to trace source code including errors although we find out defects. Therefore the information and technique between test cases and source code need to offer effective communication between testers and developers.

2.2 Traceability

Traceability means the ability to trace the life of an artefact from its inception to its use. Artefacts could be requirements, code, models, reports and test cases, etc. Generated traces can be used for several purposes: documenting links from implementation to models in order to show domain; managing changes to models; managing changes to code; performing impacts analysis. Although, to date, much of the research work in the literature has focused on requirements traceability or change effect analysis[8][9]. Also a lot of the research has carried out to improve connection between artifacts. In general, traceability is mentioned to trace from requirements to source code to maintain and understand artefacts[10][11][12][13]. Also there are the research of automation and algorithm to improve performance[14][15]. Some research to testing is proposed but they study convenience and efficiency to trace test documentations not to trace error spots[16][17]. So this paper supports to find errors and trace error spots in source code. Rigorous system testing really needs a full traceability. If “trace” is in place from requirements and test cases through models to code, tester can see what parts of the model and code are possibly defect after system testing.

Rigorous system testing also needs maximum domain based on the specification. K. Seo and E. M. Choi presented empirical comparison of major black-box testing methods based on UML and demonstrated the different domain results obtained from an experiment of testing example software system[18]. They compared five test methods: simple use case driven testing[19], collaboration diagram driven testing[4], Object-Z driven testing[20], OCL driven testing[21] and extended use case driven testing[22]. The experiments found that the extended use case driven testing method and the OCL driven method have relatively broad testing domain. An extended use case test method is a kind of black-box test based on system functions by using a scenario which contains logical flow test of the internal of program unit. Otherwise, the OCL test method doesn’t verify the logical flow but it tests the relationship with member variables or methods or objects. Therefore, if we find errors during testing the scenario instance which represents the logical flow by extended use case test method and then inspect those errors by OCL test method, testing work would be more efficient.

3. System Testing by Vertical Tracing

We define the meaning of vertical software testing in two viewpoints. The first viewpoint includes an abstraction level from unit testing level to system testing level through integration test level. It is not restricted within only one specific abstract test level. The second viewpoint is to separate system function domains as test targets and restrict testing scope on separated functions during testing vertically.

3.1 Abstraction Level Viewpoint

Fig. 1 shows testing process by vertical tracing. The process is based on V-model as the standard for carrying out IT-projects with the German government[23][23]. Left tail of the V development cycle represents the specification stream where system specifications are defined. Right tail of the V development cycle represents the testing stream where systems are being tested against the specifications defined on the left-tail. Usually unit test is executed by developers. Both integration and system testing are executed by testing or QA teams. If it is possible to identify requirements, design, and implementation artifact related to detected errors by system testing, more easily we can execute integration and system testing and modify source code. To realize this way when a tester finds errors during system testing, he goes down integration test level in detail to test the domain relying on error spot of system testing. Also by the error information detected in integration level they analyze unit test target and source code. This approach can help finding the error spot through entire test levels.

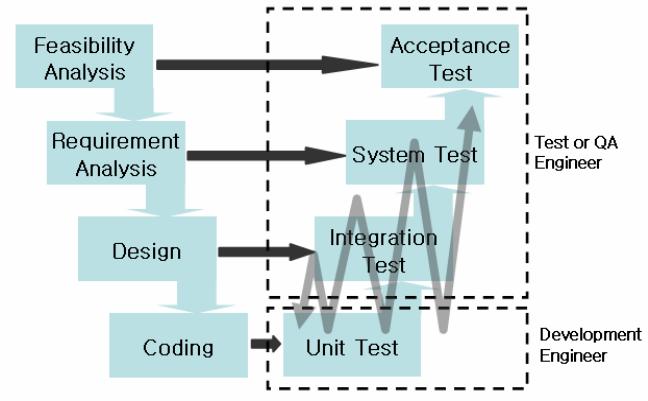

Fig. 1. Vertical tracing based on V-model[23][24]

3.2 Testing Domain Viewpoint

After coding phase, testers check system functions according to the user requirements. It is difficult for testers to check out all parts of source code like white-box style. Therefore functional test methods such as black-box test style are frequently used to check customer's needs.

Use case models system's behavior in the user point of view and describes how the system interacts with end-users. Use case slice means collection of all classes, messages, conditions that describe single function of the system. Generally speaking, a system provides many functions and many classes inside of the system are interacting for one of system's functions. No all classes work to implement one function and also no all methods and variables work in classes to implement a specific function. Therefore it is reasonable that testing only classes, methods, and variables which really work together is more efficient than testing all classes, methods, and variables to validate system functions.

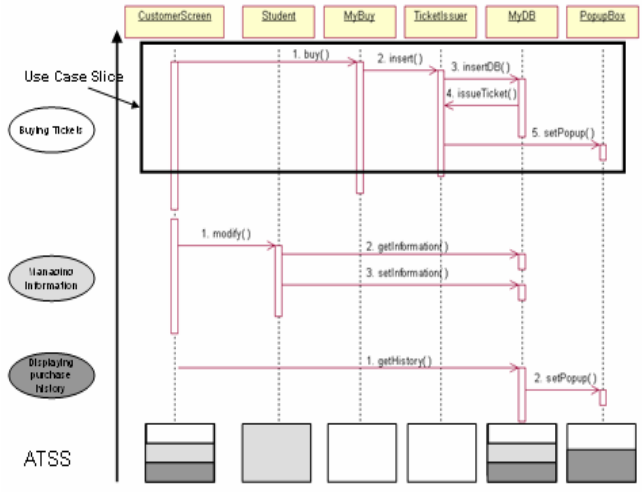

Fig. 2. Use Case slices

4. Traceability-Driven Testing

The big difference of rigorous system testing is that it has more detail process to figure out the cause of faults. Traceability level takes care in case of failures of system testing. In Fig. 3, the test process shows the possibility of vertical testing in abstract level and domain. To generate test cases with function tests, use cases and class diagrams defined in requirements analysis and design stage are referred. OCL and MM(Method Message)-Path[25] are extracted from class diagrams and sequence diagrams.

Pre-condition and post-condition expressed in OCL are used in designing test cases for system testing. We generate test data which is inputted into the first called class among classes realizing use cases by using pre-condition. And the results such as states of the last called class among classes realizing the same use case are compared with post-condition. In requirement analysis and design stage, OCL defines multiplicity, constraints, input/output conditions of a use case. Therefore post-condition of OCL can be used as expected results. In Figure 2, to buy ticket customer should input 0, 1, or 2 as a limited available date. It means the range of input data is equal or larger than 0 and equal or less than 2. MyBuy.buy() method receives one number from CustomerScreen by customer. So the input data range can be defined by OCL for MyBuy.buy() and the input data can be test data for the BuyingTicket function. In case the BuyingTicket function of ATSS is finished successfully, PopupBox.setPopup() should return 'true' boolean value to TicketIssuerer. So return data type can be defined by OCL for PopupBox.setPopup() and also the data can be expected results as test oracles.

MM-Path is a sequence of methods by messaging to realize a use case. We can make a function level test case based on the first and last called classes within MM-Path and test the result from finishing the MM-Path execution by OCL. Also we can catch error information occurred within MM-Path during function testing.

We can concern the correctness of integration testing when errors break out in function testing. So we need the way to check entire integration testing with error possible functions. Also although the function testing is also success,

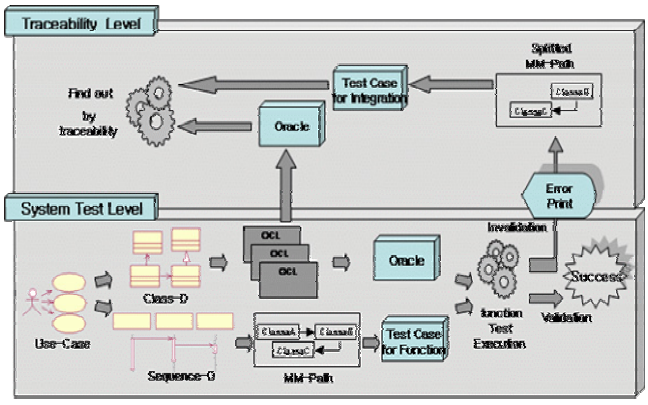

Fig. 3. Rigorous testing process

tester can want to inspect interaction of classes. For this case, repeatedly we split the MM-Path into a smaller partition and execute integration testing. This approach makes vertical testing that executes test from at first function testing to integration testing of classes in a function.

4.1 Traceability Support

In order to support traceability among models, source code, and test cases the structure of link relations among various artefacts are necessary. Each artefact has its own structure and purpose respectively but semantic associations are rarely exposed to representation. Most engineers just guess tracing from software design to code or vice versa and locate source code bearing errors by experience.

This paper searches useful trace links to test and offers connection by marking useful relation on artefacts. Above all, we analyze association between analysis requirements in various abstract levels and offers links to obtain reference.

The rigorous system test method proposed in this paper affords us to trace requirement model, object model, and source code used in generating test cases as well as in developing requirements models. The links to support traceability can be composed as follows.

- *Source code link* – mark a tag processed by javadoc. For instance, a tag such as {@link ReservationUML.CustomerStaff} supports to trace object model.
- *UML model link* – mark an extended tag to link each element such as use case, class, state, interface, etc. in UML model. For instance, if a class has {implementedBy = java.appl.hotel.ReservationAppl. java}, this supports traceability to source code.
- *Test case link* – generally test cases are represented in table style as document like in Fig. 4. Hypertext link offers traceability between documents.

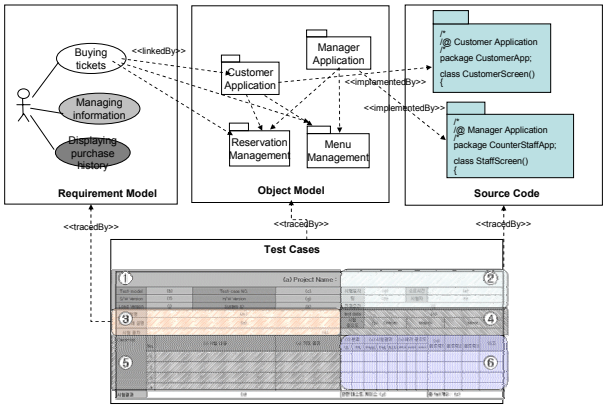

Fig. 4. Links for traceability support

	Room	Reservation	Payment
Reserve Room	checkAvailability()	create()	
Check In Customer	assignCustomer()	consume()	createBill()
Check Out Customer	removeCustomer()		payBill()

Fig. 5. Classes realizing Use Case

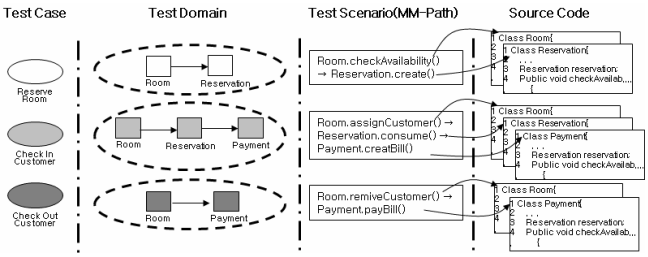

Fig. 6. Traceability support from test cases to source code

Developing a complex software system has many factors in analysis and design phase. Volume of source code and test cases are also very huge. Accordingly since the number of links to offer traceability can be extracted, introduction of CASE tool is required. This tool will provide facility of link change according to software modification. There are two approaches of implementing traceability. The first one is to create links for traceability and store link information in repository. The other is to implement trace links within tools or documents. The case of former needs to link various CASE tools to repository built by XML.

To construct a vertical testing environment, we need information for tracing between test cases and source code. Fig. 5 shows use case slices of a hotel management system. This has Reserve Room, Check-In Customer, and Check-Out Customer functions. Each function is realized by methods in three classes.

A unit of test case can be considered to be it of one use case. So in Fig. 6 a test case is designed as a use case unit. Fig. 6 shows the transformation from three test cases into classes under test, scenario and source code.

Test cases have test domain composed of classes realizing use cases. We can recognize that Room and Reservation classes are belonging to Reserve Room test domain. MM-Paths are used for generating test scenarios. For instance, the test scenario to test Reserve Room function is "Room.checkAvailability() → Reservation.create()" and we can find errors during executing this scenario. Also we can trace error spots in source code from the information about classes, methods, and attributes within the MM-Path. To construct this testing environment, we need several links data about test cases, classes, scenarios, methods and so on.

Fig. 7 shows the information of traceability links. In Fig. 7 we can see the tree structure which composed test cases, test domain, scenarios, and source code. One test case has relationship with several classes and also each class has several methods and attributes. We can also

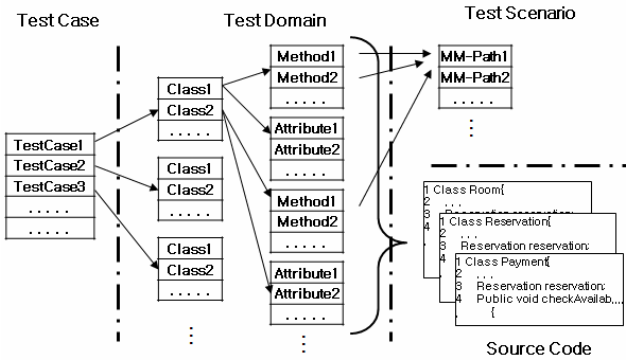

Fig. 7. Traceability links data

compose scenarios from set of classes and methods. Attributes are the testing target which is included in both scenario and source code. Therefore we can have traceability from functions to source code.

If source code is changed according to modify requirements, also the link information should be changed. In Fig. 5, suppose that 'Check Out Customer' add a function to show the room service list before the payment customer. It means that 'printList( )' method should be added in 'Payment.' It is not change { implementedBy = java.appl.hotel.Payment.java} as UML link information but add {@link ReservationUML.CustomerStaff.printList()} as javadoc expression.

4.2 Experiment and result

An experiment is designed for 1) investigating what kinds of information play important role in rigorous testing via tracing and 2) comparing effectiveness of proposed approach to recognize error spot. That includes rigorous system testing ATSS(Automatic Ticket Sales System) as a sample system by following procedure explained in Fig. 3. ATSS supports to buy a meal ticket for customers after checking date and browsing meal menu.

Table 1 shows that each combination of traceable information covers. Use case, sequence, component, and deployment diagram are used in system testing level because they are mainly applied to function testing. We

Table 1. Accessible domain of combinatorial information

		UC	SQ	UC/SQ	SQ/ST	SQ/CL	UC/SQ/ST	UC/SQ/CL
Test Level	System	○	○	○	○		○	○
	Integration		○	○	○	○	○	○
	Unit					○	○	○
Meta information	S-I						○	○
	I-U						○	○
	U-S/C							○
Linkage process or mechanism			○	○			○	○

(UC: use case, SQ: sequence, UC/SQ: UC and SQ, SQ/CL: SQ and class, SQ/ST: SQ and state chart, UC/SQ/ST: UC, SQ, and ST, UC/SQ/CL: UC, SQ, and CL, S-I: System and Integration, I-U: Integration and Unit, U-S/C: Unit and Source Code)

compared several meaningful combinations of UML diagram information and source code to find out what are the optimal combinations to provide traceability. There are 6 combination of traceable information shown at Table 1. UC/SQ/CL combination has the largest domain, meta-information to represent traceable linkage, and linkage mechanism**.**

To figure out reasonable combination of traceable information and show effectiveness of the proposed rigorous method by supporting traceability we compared 6 combinations mentioned above. We implanted various types of errors in developed sample system. Test object domain is composed of three use cases of ATSS; BuyingTicket, InformationManagement, and SearchingBuyingHistory. Each use cases has 5 ~ 7 events which can be elements of MM-path. Table 2 shows BuyingTicket use case description in brief.

Table 2. BuyingTicket Use case

Use Case : BuyingTicket	
Event	1. Customer inputs ID and Password to login. 2. Customer selects a date. 3. Customer searches a menu list. 4. Customer selects the menu. 5. Customer pays the money. 6. Customer receives a receipt. 7. Customer logouts.
Constraints	Available date is only within 3 days from the present.

Table 3. Implanted error examples

Abstraction Level	Error Type	Original	Intended error
Integration Level	Message Passing Error	db.insertBuy( )	Db.insertPurchase( )
	Method Parameter Type Error	db.insert(String code)	Db.insert(int code)
	Method Return Type Error	return true;	return true;

Unit Level	Method Algorithm Error	if(ob == btnBuy){
		if(Check()){
		Buy();
		}
		}
		else if (ob == btnFood){
		if(Check()){
		detailFoodInfor();
		}
		}
		else if(ob == btnSearch){
		getBuy();
		
		db.initBuyList(ccDate);
		}
			if(ob == btnBuy){
			if(Check()){
			Buy();
			}
			}
			else if (ob == btnFood){
			if(Check()){
			detailFoodInfor();
			}
			else if(ob == btnSearch){
			getBuy();
			
			db.initBuyList(ccDate);
			}

	Member Data Type Error	String id; id = ftID.getText();	ind id; id = ftID.getText();
	Member Data Missing	Int openCount = 0; openCount++;	No declaration openCount
	Member Data Range Error	0 <= intDate <= 2	if(intDate == 1) {ccDate = "08-Mar-11"} else if(intDate == 2) {ccDate = "08-Mar-12"} else if(intDate == 3) {ccDate = "08-Mar-13"}

Table 4. Error detection rate of combinatorial traceable information

Abstraction Level	Error Type (Implanted Error Number)	Pure System Testing					Rigorous Testing	
		UC	SQ	UC/SQ	SQ/ST	SQ/CL	UC/SQ/ST	UC/SQ/CL
Integration Level	Message Passing Error (4)	0/4	4/4	4/4	0/4	0/4	0/4	4/4
	Method Parameter Type Error (5)	0/5	5/5	5/5	0/5	0/5	0/5	5/5
	Method Return Type Error (5)	0/5	5/5	5/5	0/5	0/5	0/5	5/5
Unit Level	Method Algorithm Error (3)	0/3	0/3	0/3	3/3	0/3	3/3	3/3
	Member Data Type Error (5)	0/5	0/5	0/5	0/5	0/5	0/5	5/5
	Member Data Missing (5)	0/5	0/5	0/5	0/5	0/5	0/5	5/5
	Member Data Range Error (1)	0/1	0/1	0/1	1/1	0/1	1/1	1/1

(error location information against source code/detected errors)

To compare efficiency of various text methods, some errors are implanted intentionally in source code as test target. Table 3 shows the examples with error types. The errors are two kinds according to integration and unit abstract level. Integration level is focused on interface between methods and unit level is focused on elements like variables or sequences in methods.

Table 4 shows the error detection rate of test execution in experiment. From this result we conclude that UC/SQ/CL combination has higher error detection and rate location than the other combinations. The reason is that the meaning represented in UML diagram and implemented in source code can be traversed in detail during system level testing. In addition we can find and zoom in events or functions that cause failure by tracing links provided in rigorous testing.

Looking the shadows zone in integration abstraction level, they don't have error location information even though those test methods find failures. Usually black box testing style is used for integration level so the information to trace errors is scarce against source code. Testing method of use case succeeds in finding the system bears fault but this doesn't give developers any clue to fix errors. Developers just have to guess or suppose error spots by their own experience.

In the cases of SQ/ST, SQ/CL, and UC/SQ/ST, they also don't have information to detect errors. In SQ/ST and UC/SQ/ST, test methods don't offer the mechanism to trace error spots. Only sequence diagram is used as test scenario to construct key test cases based on state diagram. In case of SQ/CL, this method offers a test frame by using use case, sequence diagram, and class diagram but not gives specific method to design test case and domain. Especially after using sequence diagram as integration test case, the way applying class diagram is vague to support the frame from sequence diagram. Accordingly it doesn't show test method based on class diagram in detail and the error location information to trace is not enough. On the other hand, proposed method in this paper has specific method applying OCL to support class diagram based test. The employment effects of OCL are explained later because it becomes clearer in unit level test.

Table 3 shows also shadow zone in unit level. The difference between error location information and detected error numbers larger than integration level's. The reason is that system or integration test cases have description to find errors in integration or unit level, but after finding this they don't offer relation between test case and error spots. SQ and UC/SQ not only detect errors but also support traceability in integration level. But they don't offer error information in unit level because those methods don't have the way to descript unit level information. UC/SQ/ST method offers detection information to source code from state diagram in the cases of finding a method algorithm error and a member data range error. State diagram supports to check dynamical class states on specific condition from outside event or inside method. It can describe history of class attributes by algorithm and checks states of attribute at specific moment. However it doesn't have means to support information of type or missing of attributes.

We can find that the gap between error location information and detected error numbers gets higher into low abstraction level. That shows shortcoming of black box test style. Black box test style offer developer or test engineer to economic test way, but after finding errors it is not easy to trace error spots in source code. If test method has plenty of test information in each abstraction level, it supports traceability and offers the way to zoom in probable error spots. Most UML based test methods have limited information in use case or model abstraction level. That result makes hard to support traceability. However the proposed method in this paper ensures traceability and the way to zoom in because it extracts test information about each abstraction level and metadata during building test case in process.

5. Conclusion

This paper has presented detail procedure for handling traceability in system testing. Through the presentation of example traceability links and experiment of error detection/location it has been shown that the approach is viable. As the example shows, the instances of traceability links become quite large even for the small example used in experiment. This will probably not be a practical problem as the link information in them will be interpreted by tools rather than humans.

In the context of effectiveness of error detection supporting UC/SQ/CL combination for traceability link were the best in our experiment. However inserted error types should be extended to pick up the optimal combination of link in general. The idea of system testing based on traceability information illustrates more than one tool may contribute to the traceability with needs to be able to use this information in a consistent manner.

Reference

[1] E. Dustine, Effective Software Testing: 50 specific ways to improve your testing (Addison-Wesley, 2003).

[2] Lionel Briand and Yvan Labiche, A UML-based approach to system testing, Proc. 4th International Conf. on UML - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Toronto, CA, 2001, pp.194-208.

[3] Lionel Brian and Yvan Labiche, A UML-based approach to system testing, Software and System Modeling, 1(1), 2002, pp.10-42.

[4] Aynur Abdurazik and Jeff Offutt, Using UML collaboration diagrams for static checking and test generation, Proc. 3rd International Conf. on UML -- The Unified Modelling Language, Advancing the Standard, York, UK, Vol. 1939 of LNCS, 2000, pp.383-395.

[5] Jeff Offutt and Anyur Abdurazik, Generating tests from UML specifications, Proc. 2nd International Conf. on UML, 1999, pp.416-429.

[6] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe, Generating test cases from an OO model with an AI planning system, Proc. 10th International Symposium on Software Reliability Engineering, Boca Raton, Florida, 1999, pp.250-259.

[7] J. Hartmann, C. Imoberdorf and M. Meisinger, UML-Based integration testing, Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis, Portland, 2002, pp.60-70.

[8] O. Gotel and A. W. Finkelstein, An analysis of the requirements traceability problem, Proc. of the International Conf. on Requirements Engineering, Colorado Springs, CO, 1994, pp.94-102.

[9] B. Ramesh, Factors influencing requirements traceability in practice, Communications of the ACM, 41(12), 1998, pp.34-44.

[10] Marcus, A and Maletic, J. I, Recovering documentation-to-source-code traceability links using latent semantic indexing, Proc. 25th International Conference Software Engineering, 2003, pp.125-135.

[11] G. Antonio and G. Canfora, G. Casazza, Recovering traceability links between code and documentation, IEEE Transaction, Vol 28, 2002, pp.970-983.

[12] L. Naslavsky, T. Alspaugh, D. Richardson, and H. Ziv, Using scenarios to support traceability, Proc. TEFSE 2005, California, pp.25-31.

[13] T. Kastren, Towards Trace Based Model Synthesis for Program Understanding and Test Automation, Proc. International Conference on Software Engineering Advances, 2007, pp.46-56.

[14] J. Hayes, A. Dekhtyar, and J. Osborne, Improving requirements tracing via information retrieval, Proc. 11th IEEE Interantioanl Requirements Engineering Conference, 2003, pp.138-147.

[15] J. Richardson and J Green, Automating traceability for generated software artefacts, Proc, 19th International Conference on Automated Software Engineering, 2004, pp.356-366.

[16] M. Lormans and A. Van Deursen, Can LSI help reconstructing requirements traceability in design and test?, Proc. Conference on Software Maintenance and Reengineering, 2006, pp.47-56.

[17] M. Deng and B. Cheng, Retrieval By Construction: A traceability technique to support verification and validation of UML formalizations, Proc. International Jounal of Software Engineering and Knowledge Engineering, Vol. 15, 2005, pp.837-872.

[18] K. Seo and E. M. Choi, Comparison of five black-box testing methods for object-oriented software," Proc. 4th ACIS International Conference on Software Engineering Research, Management & Applications, Seattle, WA, 2006, pp.213-220.

[19] D. Wood, J. Reis, Use case derived test cases, Proc. on Conf. on Software Quality Engineering STAREAST, 1999, http://www.stickyminds.com/s.asp?F=S2021_ ART_2.

[20] Chun-Yu Chen, Constructing usage-based testing on Object-Z formal specification based specification, Ph.D. Dissertation, Auburn University, 1999.

[21] E. M. Choi, Generating test cases for object-oriented design specification described by OCL," Journal of Korean Information Processing Society, 8-D(6), 2001, pp.843-852.

[22] E. M. Choi, Use-case driven test for object-oriented system, Proc. the IASTED International Conference, ACTA Press, 2001, pp.164-169.

[23] Droschedl. W and Wiemers. M, Das V-Modell 97, (German, Oldenbourg, 1999)

[24] Roger S. Pressman, Software Engineering A Practitioner's Approach 6th, (McGraw-Hill, 2005).

[25] Paul C. Jorgensen and Carl Erickson, Object-Oriented Integration Testing, Communications of the ACM, 37(9), 1994, pp.30-37.

Eun Man Choi

He received the BS in Computer Science from Dongguk Univ. in 1982 and MS degree in Computer Science from KAIST in 1985. During 1985~1989, he stayed in Korea Research Institute of Standards and Science and DACOM Inc. to develop Korean Information Standards and National Administrative Information System. He received a Ph.D. degree in Computer Science from Illinois Institute of Technology in 1993. He has been a professor at Dongguk Univ. since 1993. His research interests include Software Design, Software Testing, Measurement, Aspect-Oriented Programming.

Kwang-Ik Seo
He received the BS and MS degrees in Computer Engineering from Dongguk Univ. in 2002 and 2004, respectively. And now he is undertaking a doctorate course as a member of the software engineering lab at Dongguk Univ. His research interests include Software Testing, Software Quality, and Process.