A SOFTWARE TRACEABILITY VALIDATION FOR CHANGE IMPACT ANALYSIS OF OBJECT ORIENTED SOFTWARE

Suhaimi Ibrahim,
Norbik Bashah Idris
Centre For Advanced Software Engineering,
Universiti Teknologi Malaysia,
Kuala Lumpur, Malaysia

Malcolm Munro
Department of Computer Science,
University of Durham,
United Kingdom

Aziz Deraman
Fac. of Technology & Infor. System,
Universiti Kebangsaan Malaysia,
Selangor, Malaysia

Abstract - *Software traceability and its subsequent impact analysis help relate the consequences or ripple-effects of a proposed change across different levels of software system. Our software traceability approach can be observed at its ability to integrate the high level with the low level software models of object-oriented software that include the requirements, test cases, design and code. It supports the top down and bottom up traceability in response to tracing for the potential effects. The objective of this paper is to present our validation experiment on a case study of software embedded system. It determines the effectiveness of our approach via a prototype tool, called CATIA. The results reveal that the nature of the components at different traceability levels affect various aspects of effectiveness metrics.*

Keywords: *Requirements traceability, impact analysis, static analysis, dynamic analysis.*

1. Introduction

Software change impact analysis [1], or *impact analysis* for short, offers considerable leverage in understanding and implementing change in the system because it provides a detailed examination of the consequences of changes in software. Impact analysis provides visibility into the potential effects of the proposed changes before the actual changes are implemented. The ability to identify the change impact or potential effect will greatly help a maintainer or management to determine appropriate actions to take with respect to change decision, schedule plans, cost and resource estimates. Many works of impact analysis are centered around the code level such as [2,3,4].

To manage impact analysis at a broader perspective is considerably hard as it involves traceability within and across different models in the software lifecycle, such as from requirements to design, design to code, etc. Ramesh relates traceability as the ability to trace the dependent items within a model and the ability to trace the corresponding items in other models [5]. Such kind of traceability is called *requirements traceability* [5]. Pursuant to this, Turner and Munro [6] assume that a system traceability implies that all models of the software are consistently updated.

Research on requirements traceability has been widely explored since the last two decades that focus up to now on the analysis of dependencies between classes, either at the design or code level [7,8,9]. When traceability is used to execute impact analysis, one of the factors to be considered is the effectiveness of impact analysis. The effectiveness has been defined and measured in different ways in the literature. Arnold and Bohner provide a general framework for expressing the effectiveness of an impact analysis approaches [10]. Lindval and Sandahl apply and measure the impact analysis effectiveness based on the comparison between set of objects predicted as changed/unchanged and actually changed/unchanged components [7]. Bianchi *et al.* define and apply some metrics based on the granularity and traceability models [8]. Bianchi's nature of work and assessment

mainly contribute to our validation, although it is different in objective.

The aim of our study in this paper is to present our software traceability validation based on a controlled experiment. In the experiment, the subjects evaluated our traceability model via a prototype tool, called CATIA (Configuration Artifact Traceability For Impact Analysis) that supports C++ software and applied it to a software development project of an embedded system.

This paper is organized as follows: Section 2 presents an overview of our traceability model. Section 3 discusses some traceability techniques used. Section 4 presents an empirical study of a case study and controlled experiment. Section 5 presents some results and discussions. Lastly, section 6 gives a conclusion and future work.

2. Our Traceability Model

Figure 1 reflects the meta model of our traceability system to establish the relationships between software components that include the requirements, design, test cases and code. These components represent the workproducts of the software development phases which can be extracted from the requirements and specification, design, testing and code documents respectively. The arrows represent the direct relationships derived from a static and dynamic analysis of component relationships [11]. Static analysis is obtained from a study on source code and other related models. Dynamic analysis on the other hand, results from the execution of software to find traces such as executing some test cases of a requirement to find its impacted codes.

Requirement level may include other high level requirements such as user requirements, business requirements or requirement modeling. Meanwhile, design level can be classified into high level design abstracts (e.g. collaboration design models) and low level design abstracts (e.g. class diagrams) or a combination of both. For the sake of research and implementation, we focus on functional requirements that can be derived from the high level requirements and low level design abstracts to represent design that contain the software packages and class interactions of UML class diagrams [14]. While test cases represent all possible conditions to satisfy each requirement, and code is to include all the methods, their contents and interactions.

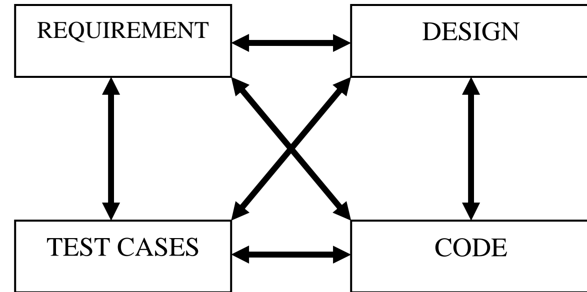

Figure 1: Meta model of traceability system

We classify our model into two categories; vertical and horizontal traceability. Vertical traceability refers to the association of dependent items within a model and horizontal traceability refers to the association of corresponding items between different models [12].
In general, the impact analysis of a system can be interpreted as follows.

$S = (G, E)$

$G = GR \cup GD \cup GC \cup GT$

$E = ER \cup ED \cup EC \cup ET$

The system impact, S is a set of inter-artifact relationships. G represents the artifact models or levels i.e. requirements (GR), design (GD), test cases (GT) and code (GC) and E represents a relationship between artifacts of different levels. Each level can be defined in the following perspectives.

i) Requirement Traceability

$ER \subseteq GR \times SGR$

$SGR = GD \cup GC \cup GT$

$GR = \{R_1, R_2, \ldots\ldots, R_n\}$

ER is defined as a relationship between requirements (GR) and other artifacts of different levels (SGR). SGR can be a design, code or test cases. GR is a set of requirements.

ii) Design Traceability

$ED \subseteq GD \times SGD$

$SGD = GR \cup GC \cup GT$

$GD = \{D_1, D_2, \ldots\ldots, D_n\}$

ED is defined as a relationship between design and other artifacts of different levels (SGD). SGD can be requirements, code or test cases. GD is a set of design components.

iii) Test case Traceability

$ET \subseteq GT \times SGT$

$SGT = GR \cup GD \cup GC$

$GT = \{T_1, T_2, \ldots\ldots, T_n\}$

ET is defined as a relationship between test cases and other artifacts of different levels (SGT). SGT can be requirements, design or code. GT is a set of test cases.

iv) Code Traceability

$EC \subseteq GC \times SGC$

$SGC = GR \cup GD \cup GT$

$GC = \{V_1, V2, \ldots, V_t\}$

EC is defined as a relationship between code components and other artifacts of different levels (SGC). SGC can be requirements, design or test cases. GC is a set of code components. The relationships at code level can be further constructed between variables, methods and classes based on OO dependencies [13].

Our model provides the ability to support top down and bottom up tracing for potential impacts e.g. from requirements to design, requirements to test cases, requirements to code, design to test cases, design to code, test cases to code or vice versa with methods being considered as our smallest artifacts. The detailed implementation of our model was presented in [13].

3. Traceability Techniques

Traceability provides some infrastructures to support change impact analysis. Basically, traceability can be established via the following techniques.

i. Explicit links
ii. Cognitive links
iii. Name tracing
iv. Concept location

We apply a combination of the techniques i), ii) and iv) into our implementation. *Explicit links* [14] are required at the low level design as the class diagrams can be recovered from code using our own code parser, called *CodeMentor* [15]. *Cognitive links* [7] are applied to allow developers or experts decide on the true impacts between requirements and implementation code. *Name tracing* [16] is not appropriate in our context as it ignores program dependencies of which this information is very useful to account for change impact.

We use *concept location* [17] to establish potential impacts between each requirement and its implementation code via automated test scenarios [18]. The test scenarios apply some test cases (available from the system documentation) to generate the impacts in terms of the classes and methods. We incorporate this technique with some cognitive elements in ii). It allows developers or experts to select the only true impacts from the available potential impacts. We captured and collected all these components and their relationships into a database repository for CATIA implementation.

4. Empirical Study

4.1 Case Study

We applied our traceability approach to a case study of software project, called the Automobile Board Auto Cruise (OBA). The OBA project is an embedded software system of size 4k LOC with 480 pages of documentation. The project was built with a complete project management and documentation standard adhering to DoD standard, MIL-STD-498 [19]. The software design was built based on the UML specification and design standards [14] with code written in C++.

We performed static analysis on code to obtain the program dependencies of classes and methods. We had also spent a substantial time and effort on dynamic analysis by executing each requirement into its implementation code via test scenarios mentioned earlier. Our objective here was to establish all the impacted artifacts and relationships prior to a controlled experiment.

4.2 Controlled Experiment

Subjects and Materials

Twenty participants were involved in a one day experiment with the above case study used as a maintenance project. Participants were the post-graduate students of software engineering at the Centre For Advanced Software Engineering, Universiti Teknologi Malaysia. The subjects were familiar with the above case study in which OBA was used as common development teamwork for their final year project. A completed OBA system was then selected from the best teamwork of the year.

The subjects represented the software practitioners as all of them had at least one year working experience and had some knowledge on software maintenance they learnt during the post graduate course. The subjects were divided into five groups of fairly distributed expertise and working experience. They were provided with a set of system documentations, CATIA tool and source code of OBA project to start the experiment.

Experimental Procedures

Each group was given a set of problem change requests (PCRs) that consist of i) a request to identify the bottom-up traceability of

potential impacts ii) a request to identify the top-down traceability of potential impacts. In i), group had to transform a PCR into some understandable software components, called PIS (primary impact set). PIS is a set of initial or first target components that need to be changed. PIS can be set at any artifact levels. In this experiment, groups were asked to decompose PIS down to its variables and used CATIA to produce the potential impacts (SIS-secondary impact set) in terms of methods, classes, packages, test cases and requirements.

CATIA produced some useful information on the SIS including its LOC and complexities. With SIS, groups had to draw the actual impacts (AIS-actual impact set) by manually searching through the right paths in the code and other components for validation. They were also required to state the time duration in minutes to complete the search for PIS and AIS. Lastly, they had to modify, compile and submit the compiled code together with the data they had obtained. We double checked their work by examining through all the paths and steps they had undertaken to make sure that no careless mistakes occurred. In ii), groups were asked to choose any one of the available requirements and used CATIA to produce the impacted test cases, packages, classes and methods.

4.3 Effectiveness Metrics

We need to determine some metrics of effectiveness as defined by [8,10].

Inclusiveness

Inclusiveness is used for assessing the adequacy of an impact analysis approach. It is defined as

Inclusiveness = 1 if AIS $\subseteq$ SIS

Inclusiveness = 0 otherwise.

This metric indicates that if *inclusiveness* = 1, then it is worthwhile considering further metrics *S-Ratio, Amplification* and *Change Rate*.

S-Ratio

S-Ratio is defined as an indicator of the adherence between AIS and SIS. This metric indicates how far the SIS corresponds to the AIS that can be expressed as

S-Ratio = AIS#/SIS#

The smaller the ratio, the fewer will be the number of impacted components that actually need to be changed, and less modification effort is required. The desired trend is to see that the *S-Ratio* = 1 i.e. both AIS# and SIS# coincides.

Amplification

Amplification metric is to observe the ripple-sensitivity i.e. the effect caused by making a small other parts of a system.

Amplification = PIS#/SIS#

The desired trend is not to have the SIS with a bigger different than PIS, so that the *amplification* tends toward 1.

Change Rate

Change Rate is to assess the sharpness of impact analysis approach. *Change Rate* is defined as

Change Rate = SIS#/System

The smaller the estimated impact in a system, the better will be the result of impact analysis approach. The desired trend is to have the SIS# relatively small subset of the system such that the *Change Rate* is <<1.

5. Results and Discussion

We identified from the OBA project, 46 requirements, 34 test cases, 12 packages, 23 classes and 80 methods. Table 1 reveals the results of bottom-up traceability collected and derived from the experiment. In each artifact level, the *Inclusiveness* was 1, meaning that the AIS was always part of the SIS. This indicates that the actual impacts is within the estimated impacts, thus further metrics such as *S-Ratio*, *Amplification* and *Change Rate* were meaningful.

The *S-Ratio* relates to how far the AIS is adherent to the SIS. The data show that adherence gets worse (i.e. the *S-Ratio* decreases) as the degree of granularity gets finer. This means that the maintainer will have to devote a substantial maintenance effort to search for the components that will actually be modified. *Amplification* expresses how much the SIS extends the PIS. The result shows that the *amplification* increases as the granularity gets finer. This is due to the fact that more component dependencies exist at the lower level, thus the SIS tends to increase considerably.

Finally, the *Change Rate* expresses how large is the estimated impacts can be detected out of a system. The data shows that this metric gets worse (i.e. increases) as the granularity gets coarser. We calculated the *S-Ratio* average of method, class and package levels to be 0.59, 0.75 and 0.87 respectively. These results show that as

the degree of granularity gets finer (e.g. methods) the SIS# tends to be less accurate i.e. 0.59. But as the degree of granularity gets coarser, the AIS# and SIS# become more generalized and get closer to 1.0.

Table 1: The results of bottom-up change impacts

Grp	Levels	PIS#	SIS#	AIS#	Incl	S-Ratio	Ampl	Change Rate	Analy. Effort (PIS)	Spec. Effort (AIS)
A	Method	2	5	3	1	0.60	2.5	0.06	22	20
	Class	-	2	1	1	0.50	-	0.09		
	Package	-	1	1	1	1.0	-	0.08		
	T. Case	-	30	16	1	0.53	-	0.88		
	Reqt.	-	44	19	1	0.43	-	0.96		
B	Method	2	13	8	1	0.62	6.5	0.16	27	35
	Class	-	4	3	1	0.75	-	0.17		
	Package	-	3	2	1	0.67	1.5	0.25		
	T. Case	-	32	13	1	0.41	-	0.94		
	Reqt.	-	44	17	1	0.38	-	0.96		
C	Method	3	10	6	1	0.60	3.33	0.13	30	27
	Class	-	4	3	1	0.75	-	0.17		
	Package	-	3	2	1	0.67	-	0.25		
	T. Case	-	34	16	1	0.47	-	1.0		
	Reqt.	-	46	20	1	0.44	-	1.0		
D	Method	3	11	6	1	0.55	3.67	0.14	35	25
	Class	-	4	3	1	0.75	-	0.17		
	Package	-	2	2	1	1.0	-	0.17		
	T. Case	-	34	10	1	0.29	-	1.0		
	Reqt.	-	46	15	1	0.33	-	1.0		
E	Method	2	5	3	1	0.60	2.5	0.06	25	22
	Class	-	2	2	1	1.00	-	0.09		
	Package	-	2	2	1	1.0	-	0.17		
	T. Case	-	31	8	1	0.26	-	0.91		
	Reqt.	-	45	22	1	0.49	-	0.98		

At the test cases and requirement levels, the SIS produced were too generalized as almost all the test cases and requirements were affected by the PIS. This is due to the fact that some common program modules such as the start/end module and variable initialization module were always called to execute any test cases or requirements no matter whether we like it or not. To reduce this issue, we had allocated a substantial time prior to actual experiment capturing and selecting the true impacts by employing developers to implement test scenarios on each test case and requirement. They captured all the potential impacts and derived from it some useful or true impacts with the help of our supporting tool, *TestAnalyzer*

[13]. Thus, we treated the user decisions as AIS and the results produced by the *TestAnalyzer* as SIS. We kept these results into a CATIA database for implementation.

Table 2: The results of top-down change impacts

Grp	Pri. Artifacts	Sec. Artifacts	Cnt	LOC	VG
A	req9	Mtd	22	348	116
		Cls	13	473	154
		Pkg	9	491	160
		Tcs	3	348	116
		Reqt	-	-	-
B	req46	Mtd	26	389	106
		Cls	11	477	123
		Pkg	8	473	133
		Tcs	3	389	106
		Reqt	-	-	-
C	req22	Mtd	23	289	99
		Class	14	473	154
		Pkg	10	491	160
		Tcs	2	289	99
		Reqt	-	-	-
D	req36	Mtd	23	229	64
		Cls	12	352	109
		Pkg	8	485	138
		Tcs	2	229	64
		Reqt	-	-	-
E	req32	Mtd	19	244	72
		Cls	10	399	114
		Pkg	7	485	138
		Tcs	1	244	72
		Reqt	-	-	-

We also assessed two *efficiency* metrics of the maintenance process: *Analysis Effort* that the subjects spent time to search for the PIS, and the *specification Effort* expressed the time taken to search for the AIS. These efforts were expressed in minutes. Please note that, duration was applied to method, class and package levels only. The data shows that the *Analysis Effort* required is greater in group C and D. The increased effort could be explained by an increase in PIS# and SIS#. Meanwhile, the group B recorded the highest *Specification Effort* due to an increase in AIS#.

Table 2 depicts the results of top-down traceability where groups were asked to choose a requirement and use CATIA to generate the impacts on other low level components. These impacts represent the true impacts we had collected earlier. If compared to potential impacts, the correlative result was unpredictable. The table shows that the req9 (group A) had caused impacts on 22 methods, 13 classes, 9 packages and 3 test cases. The impacts took up LOC (348) and VG (116) on methods, LOC (473) and VG (154) on classes, LOC (491) and VG (160) on packages, LOC (348) and VG (116) on test cases. Please note that no requirements were allowed to cause impact on other requirements as each requirement is unique.

6. Conclusion and Future Work

Our traceability approach and tools provide some leverage and support for change impact analysis. The experiment revealed that in bottom up traceability, our approach tends to provide less accuracy (i.e. S-Ratio decreases) at the finer granularity in code. This means more maintenance effort is required as the degree of granularity get finer to search for the actual impacted components. Searching impacts of classes and packages seems to be more efficient than methods but it is too generalized to appreciate the actual effects. Therefore, the maintainers or management should decide on whether to establish the efficiency or accuracy of the maintenance task as the main objective to satisfy.

In top down traceability such as from requirements or test cases to code, the results of the potential impact were less useful. This is due to the fact that a requirement itself is very subjective. We could capture the estimated impacts on code via test scenarios but the actual impacted components (AIS) were arbitrary that could only be decided by the users. We foresee the need to cover other high level components in software lifecycle such as architectural design and specification. The artifact dependencies at these levels are more complicated to manage as it contains more semantics rather than syntactic structures.

References

[1] S.A. Bohner and R.S. Arnold, "An Introduction to Software Change Impact Analysis", IEEE CS Press, pp. 1-26, 1996.

[2] D. Kung, J. Gao, P. Hsia, and F. Wen, "Change Impact Identification in object-oriented software maintenance", Proceedings of International Conference on Software Maintenance, pp. 202-211, 1994.

[3] M. Lee, "Algorithmic analysis of the impacts of changes to object-oriented software", Proceedings of 34th International Conference on and Systems, pp. 61-70, 2000.

[4] X. Chen, W.T. Tsai and H. Huang, "Omega – An integrated environment for C++ program maintenance", IEEE, pp. 114-123, 1996.

[5] B. Ramesh, "Requirements traceability: Theory and Practice", Annuals of Software Engineering, vol. 3, pp. 397-415, 1997.

[6] R.J. Turver and M. Munro, "An Early impact analysis technique for software maintenance", Journal of Software Maintenance: Research and Practice, Vol. 6 (1), pp. 35-52, 1994.

[7] M. Lindvall and K. Sandahl, "Traceability Aspects of Impacts Analysis in Object-Oriented System", Journal of Software Maintenance Research And Practice, vol. 10, pp. 37-57, 1998.

[8] A. Bianchi, A.R. Fasolino, A.R. and G. Visaggio, "An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models", IWPC, pp. 149-158, 2000.

[9] H.M. Sneed, "Impact Analysis of maintenance tasks for a distributed object-oriented system", Proceedings of Software Maintenance, IEEE, pp. 180-189, 2001.

[10] R.S. Arnold and S.A Bohner, "Impact Analysis – Towards A Framework for comparison", Proceedings of the Conference On Software Maintenance, pp. 292-301, 1993.

[11] S. Ibrahim, N.B. Idris, M. Munro and A. Deraman, "Implementing A Document-Based Requirements Traceability: A Case Study", Proceedings of IASTED Int'l Conference on Software Engineering, pp. 124-131, 2005.

[12] O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem", in Proceedings of the First International Conference on Requirements Engineering, Colorado, pp. 94-101, 1994.

[13] S. Ibrahim, N.B. Idris, M. Munro and A. Deraman, "*A* Requirements Traceability to Support Change Impact Analysis", Asean Journal of Information Technology, Pakistan, vol. 4(4), pp. 345-355, 2005.

[14] G. Booch, I. Jacobson and J. Rumbaugh, "UML Distilled Applying the Standard Object Modeling Language", Addison-Wesley, 1997.

[15] S. Ibrahim and R.N. Mohamad, "Code Parser for C++", Technical report of Software Engineering, CASE/August 2004/LT2, 2004.

[16] Y. Maarek, D. Berry and G. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries", IEEE Trans. Software Engineering, Vol. 17(8), pp. 800-813, 1991.

[17] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension", Proceedings of 10th International Workshop on Program Comprehension, IEEE, pp. 271-278, 2002.

[18] S. Ibrahim and M.A. Al-Busaidi, "A Dynamic Analysis Approach to Support Concept Location". Universiti Teknologi Malaysia: Technical report of Software Engineering, CASE/Mei 2004/LT2, 2004.

[19] MIL-STD-498. "*Roadmap for MIL-STD-498*".http://www2.umassd.edu/SWPI/DoD/MIL-STD-498/