Integrating Software Traceability for Change Impact Analysis

Suhaimi Ibrahim[1], Norbik Bashah Idris[1], Malcolm Munro[2], and Aziz Deraman[3]
[1] Centre for Advanced Software Engineering, University of Technology Malaysia, Malaysia
[2] Department of Computer Science, University of Durham, United Kingdom
[3] Faculty of Technology and Information System, University Kebangsaan Malaysia, Malaysia

Abstract: *Software maintenance is recognized as the most costly activity in software engineering with typical estimates of more than half of the software development cost. The main problem to a maintainer is that seemingly small changes can ripple throughout the system to cause substantial impact elsewhere. Software traceability and its subsequent impact analysis help relate the consequences or ripple-effects of a proposed change across different levels of software models. In this paper, we present a software traceability approach to support change impact analysis of object oriented software. The significant contribution in our traceability approach can be observed in its ability to integrate the high level with the low level software models that involve the requirements, test cases, design and code. Our approach allows a direct link between a component at one level to other components at any levels. It supports the top down and bottom up traceability in response to tracing for the ripple-effects. We developed a software prototype called Catia to support C++ software, applied it to a case study of an embedded system and discuss the results.*

Keywords: *Requirement traceability, impact analysis, concept location, change request*.

Received June 17, 2004; accepted October 30, 2004

1. Introduction

Software maintenance is recognized as the most expensive phase of the software lifecycle, with typical estimates ranging from 60% to 80% of the total cost [16]. Many software practitioners generally agree that making software changes without visibility into their effects can lead to poor effort estimates, delay in release schedules, degradation in software design, unreliable software products, and premature retirement of the software system [14, 20]. The Year 2000 date-change phenomenon is a good example of poor insight into the impacts of change [1].

Software change impact analysis [3], 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 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.

A maintainer generally accomplishes impact analysis by analyzing the existing dependencies or relationships among the software components composing the software system. Two main lines of approach have been described in the literature for executing such analyses. The first approach addresses the problem at the code level, focusing on the analysis of various dependencies in code, like data and control dependencies, or function dependencies. The second approach takes into account some of the software models in the software life-cycle, such as from the design model to code model. This approach addresses impact analysis from a broader perspective.

To implement impact analysis at a broader perspective is considerably hard to manage as it involves traceability within and across different models. 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 [18]. Such kind of traceability is called *requirement traceability* [18]. Pursuant to this, Turner and Munro [21] assume that a system traceability implies that all models of the software are consistently updated.

Research on requirement traceability has been widely explored since the last two decades that supports many areas such as reverse engineering, visualization, reuse, etc. Traceability is fundamental to the software development and maintenance of large system. It shows the ability to trace from high level abstracts to low level abstracts e. g., from a requirement to its implementation code. The fact about this traceability model is that if the component relationships are too coarse, they must be decomposed to understand complex relationships. On the other hand, if they are too granular, it is difficult to

reconstruct them into more recognized, easily understood software work products [4].

We would like to explore change impact analysis from which we would be able to capture the impacts of a proposed change. What we mean by a 'proposed change' is a target component that needs to be modified as a result of change request. Change request is initiated by the client or internal development staff due to the need to make a change in the software system. It should be translated into some explicit and more understandable items before a change impact analysis can be implemented.

This paper is organized as follows. Section 2 presents an overview of our traceability and impact analysis model. Section 3 discusses our approach to capture the artefacts change impact and its traceability. Section 4 discusses a case study and followed by some results in section 5. Section 6 presents some related work. Finally, section 7 presents a conclusions and some glimpses into future work.

2. Traceability Model

Figure 1 reflects the notion of our model to establish the relationships between artefacts; requirements, design, test cases and code. The thick arrows represent direct relationships while thin arrows represent indirect relationships. Both direct and indirect relationships can be derived from static and dynamic analysis of component relationships.

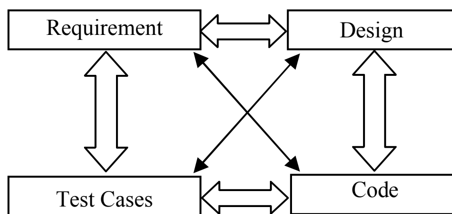


Figure 1. Meta-model of system traceability.

Direct relationships apply actual values of two components, while indirect relationships apply intermediate values of relationship e. g., using a transitive closure. Static relationships are software traces between components resulting from a study of static analysis on the source code and other related models. Dynamic analysis on the other hand, results from execution of software to find traces such as executing test cases to find the impacted codes. 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 [7].

2.1. Horizontal Traceability Analysis

We regard a horizontal traceability as a traceability model of inter-artefacts such that each component (we call it as an artefact) in one level provides links to other components at different levels. Figure 2 shows a traceability from the point of view of requirements. For example, R1 is a requirement that has direct impacts on test cases T1 and T2. R1 also has direct impacts on the design D1, D2, D3 and on the code component C1, C3, C4. Meanwhile T1 has its own direct impact on D1 and D1 on C4, C6, etc which reflect the indirect impacts to R1. The same principle also applies to R2. R1 and R2 might have an impact on the same artefacts e. g., on T2, D3, C4, etc. Thus, the system impact can be interpreted as follows:

$$S = (G, E)$$
$$G = GR \cup GD \cup GC \cup GT$$
$$E = ER \cup ED \cup EC \cup ET$$

Where

S: Represents a total impact in the system

G: Represents an artefact of type requirements (GR), Design (GD), Code (GC) or Test cases (GT).

E: Represents the relationships between artefacts from the point of view of an artefact of interest. This is identified by ER, ED, EC, and ET.

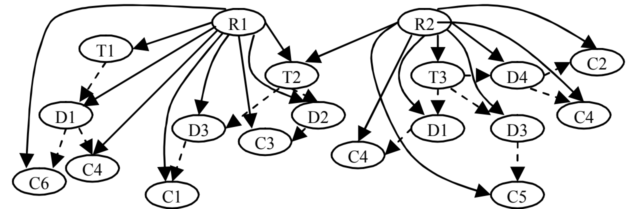


Figure 2. Traceability from the requirement traceability perspective.

Each level of horizontal relationship can be derived in the following perspectives.

1. Requirement traceability

$$ER \subseteq GR \times SGR$$
$$SGR = GD \cup GC \cup GT$$

A requirement component relationship (ER) is defined as a relationship between requirement (GR) with other artefacts (SGR) at different levels.

2. Design traceability

$$ED \subseteq GD \times SGD$$
$$SGD = GR \cup GC \cup GT$$

A design component relationship (ED) is defined as a relationship between a design component (GD) with other artefacts (SGD) at different levels.

3. Test case traceability

$$ET \subseteq GT \times SGT$$
$$SGT = GR \cup GD \cup GC$$

A test case component relationship (ET) is defined as a relationship between a test case (GT) with other artefacts (SGT) at different levels.

4. Code traceability

$$EC \subseteq GC \times SGC$$
$$SGC = GR \cup GD \cup GT$$

A code component relationship (EC) is defined as a relationship between a code component (GC) with other artefacts (SGC) at different levels.

2.2. Vertical Traceability Analysis

We relate a vertical traceability model as intra-artefacts of which an artefact provides links to other artefacts within the same level. In principle, we consider the following as vertical platforms:

- Requirement level.
- Test case level.
- Design level.
- Code level.

Requirement level here refers to the functional requirements. While the test case level refers to the test descriptions that describes all possible situations that need to be tested to fulfil a functional requirement. In some systems, there might exist some requirements or test cases being further decomposed into their sub components. However, to comply with our model, each is uniquely identified. To illustrate this phenomenon, let us consider the following example.

Req #: 5
Code: SRS_REQ-02-05

Description: The driver presses an "Activation" button to activate the AutoCruise function.
The test cases involved:

1. *Test case #: 1*
   Code: TCASE-12-01
   Description: Launch the Auto Cruise with speed is > 80 km/hr.
   1.1. Test case #: 1.1
   Code: TCASE-12-01-01
   Description: Launch the Auto Cruise while not on fifth gear.
   1.2. Test case #: 1.2
   Code: TCASE-12-01-02
   Description: Launch the Auto Cruise while on fifth gear.
2. *Test case #: 2*
   Code: TCASE-12-02
   Description: Display the LED with a warning message "In Danger" while on auto cruise if the speed is >= 150 km/h.

We can say that *Req # 5* requires three test cases instead of two as we need to split the group of test *case# 1* into its individual test *case # 1.1* and test *case # 1.2*.

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. In our implementation, we apply the low level design abstracts that contain the software packages and classes with their interactions while the code level is to include all the methods and their interactions.

3. Approach

3.1. Hypothesize Traces

We believe that some relationships exist among the software artefacts in a system. We need to trace and capture their relationships somehow not only within the same level but also across different levels before a change impact analysis can be implemented. The process of tracing and capturing these artefacts is called hypothesizing traces.

Hypothesized traces can often be elicited from system documentation or corresponding models. It is not important in our approach whether the hypotheses should be performed manually through the available documentations and software models or automatically with the help of a tool. Figure 3 reflects one way of hypothesizing traces. It can be explained in the following steps:

1. For each requirement, identify some selected test cases (RxT).
2. Clarify this knowledge with the available documentation, if necessary.
3. Run a test scenario (dynamic analysis) for each test case based on the available test descriptions and procedures, and capture the ripple effect in terms of the methods involved (TxM). We developed a tool support, called *CodeMentor* to identify the impacted code by instrumenting the source code prior to its execution [11].
4. Perform a static analysis on the code to capture the call graphs of call-invocations (MxM) and other structural relationships as shown in Table 1. These structural relationships are explicitly available in codes that need to be captured because they represent the program dependencies affected by the change impact. More discussion on this is found in the following section.

We experimented using tool supports such as *McCabe* [23] and *Code Surfer* [22] to help capture the program dependencies. Other manual works as well as the need for other types of information saw us developing our own code parser called *TokenAnalyzer* [10].

3.2. Impact Analysis

Techniques are available to address program dependencies in code such as call graphs, data flows and dependence graphs of program slicing [9]. However, the way these techniques are used may vary depending on the problem being addressed. In our case, we use the call graphs and dependence graphs to

capture the program dependencies of method-to-method and class-to-class relationships. For example, in method-to-method relationships, we identify the method interactions derived from the call relationships. Say, if

$$M1 \rightarrow M2$$
$$\rightarrow M4$$

M1 calls two other methods; M2 and M4. This means any change made to M2 or M4 would have a potential effect on M1. So, in our context of impact analysis, we have to work the other way around by picking up a callee and finding its corresponding callers.

In another example, if class A is inherited from class B, then any change made in class B may affect class A and all its lower subclasses, but not its upper classes. Table 1 provides information on what type of relationship, with descriptions and examples of all possible dependencies in C++ that can contribute to change impact. Our code parser, *TokenAnalyzer* was specially designed and developed to capture all these dependencies.

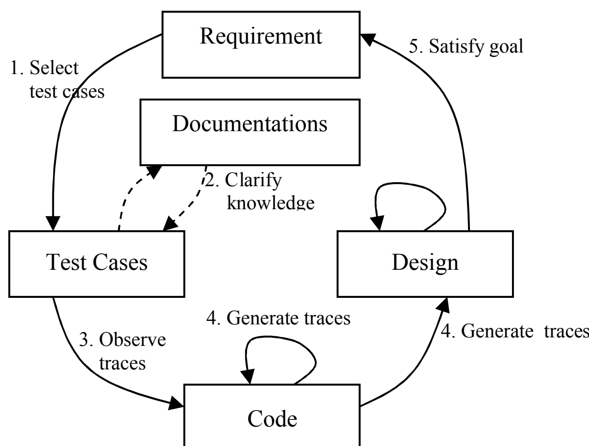


Figure 3. Hypothesized and observed traces.

3.3. Traceability Approach

Intrinsically, traceability provides a platform to implement change impact analysis. We can classify three techniques of traceability.

1. Traceability via *explicit links*: *Explicit links* provide a technical means of explicit traceability e. g., traceability associated with the basic inter-class relationships in a class diagram modelled using UML [5].
2. Traceability via *name tracing*: *Name tracing* assumes a consistent naming strategy and is used when building models. It is performed by searching items with names similar to the ones in the starting model [15].
3. Traceability via *domain knowledge* and *concept location: Domain knowledge* and *concept location* are normally used by experienced software developer tracing concepts using his knowledge about how different items are interrelated [17].

We apply 1 and 3 in our traceability approach. We obtain the *explicit links* of component relationships from the hypothesized traces and establish a set of matrices to implement the traceability between components in the system. We use *concept location* to establish links between requirements and test cases with the implementation code. This process requires a maintainer to understand the *domain knowledge* of the system he wants to modify. With this prior knowledge of a requirement, a maintainer should be able to decompose it into more explicit items in terms of classes, methods or variables. These explicit items represent a requirement or a concept that are more traceable in the code [17]. With the help of test cases in hand, our approach via *codeMentor* should be able to support a maintainer tracing and locating the ripple-effects of the defined items in terms of the impacted methods and classes.

Name tracing is another technique for implementing traceability. It can be used to locate the corresponding items of a model with another model e. g., to locate the occurrences of an item of similar name in a requirement with the ones that exist in the implementation code in an effort to establish some links between requirements and code. However, this strategy is not practical in our context of study. The reason is that *name tracing* cannot be used to search for structural relationships of program dependencies.

Figure 4 describes the implementation of our total traceability approach. The horizontal relationships can occur at the cross boundaries as shown by the thin solid arrows. The crossed boundary relationship for the requirements-test cases is shown by RxT, test case-code by TxM, and so forth. The vertical relationships can occur at the code Method interactions (MxM) and design Class interactions (CxC), Package interactions (PxP) level respectively. The method interactions can simply be transformed into class interactions and package interactions by the use of mapping mechanism based on the fact that a package is made up of one or more classes and a class is made up of one or more methods. The thick doted lines represent the total traceability we need to implement in either top down or bottom up tracing. By top-down tracing, we mean we can identify the traceability from the higher level artefacts down to its lower levels e. g., from a test case we can identify its associated implementation code. For bottom-up tracing, it allows us to identify the impacted artefacts from a lower to a higher level of artefacts e. g., from a method we can identify its impacted test cases and requirements.

Table 1. Structural relationships in C++.

Relationships	Definitions	Examples

Aggregation	A class contains a data members of pointer (reference) to some other class.
	-----Impact -------
	A ← B
		class B {};
		class A
		{
		B * test
		};

Call	An operation (method) of the first class calls an operation of the second one. -----Impact ------- A ← B	class B {void f ()}; class A { void method () { B test; test.f ();} }

Composition	A class contains a data member of some other class type.
	-----Impact -------
	A ← B
		class B {};
		class A
		{
		B test;
		}

Create	Some operation of the first class creates an object of the second class (instantiation) .
	-----Impact -------
	A ← B
		class B {};
		class A
		{
		B *s = new B ();
		}

Inheritance	Inheritance relation among classes. -----Impact ------- A ← B	class B {}; class A : public B

Association	A class contains an operation with formal parameters that have some class type.
	-----Impact -------
	A ← B
		class B{};
		class A
		{
		void m1 (B* par = 0);
		}

Friendship	Dependency from two classes.
	-----Impact -------
	B ← A
		class B{};
		class A
		{
		friend class B;
		}

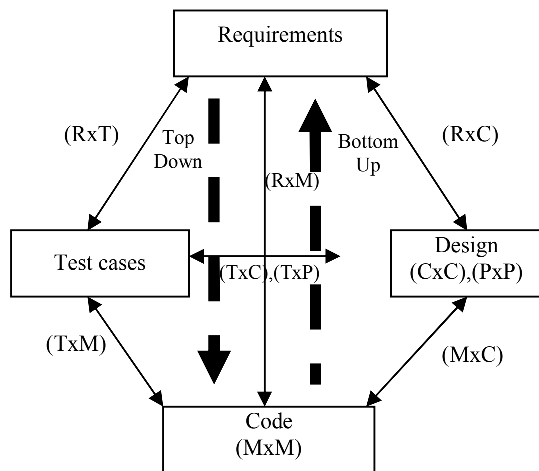


Figure 4. System artefacts and their links.

4. CASE Study: OBA

To implement our model, we applied it to a case study of software project, called the On-Board Auto Cruise (OBA). OBA is actually an embedded software system with LOC of about 4K, developed as a term project in group-based post-graduate students of computer science at the Centre for Advanced Software Engineering, University of Technology Malaysia. It was built with a complete project management and documentations adhering to DOD standards, MIL 498 [8]. The software project was built based on the UML specification and design standards [5] with software written in C++. The objective of our case study is to realize a traceability and impact analysis between the software components that include the requirements, design, code and test cases.

5. Results

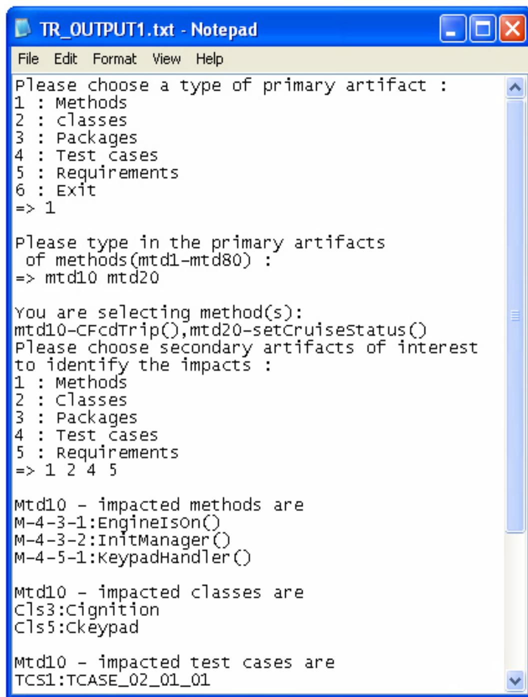
We identified from the OBA project, 46 requirements, 34 test cases, 12 packages, 23 classes and 80 methods. Our prototype, Configuration artifact traceability for impact analysis (*Catia*) assumes that a user change request has already been translated and expressed in terms of the acceptable components i. e., requirements, classes, methods or test cases. *Catia* was designed to manage the impact of one type of artifacts at a time. The system works such that given an artifact as a primary impact, *Catia* can determine its impacts on other artifacts (secondary artifacts) in either top-down or bottom-up tracing.

Listing 1 and Listing 2 show the output of our prototype. In Listing 1, the user chose a requirement by inputting *req12* as a primary impact and chose *classes* and *test cases* as his secondary impact of interest. *Catia* then produced a list of impacted *classes* and *test cases* for the *req12*. This snapshot output reflects the top-down traceability as the result was derived from a requirement down to its low level software components.

```
TR_OUTPUT.txt - Notepad
File  Edit  Format  View  Help
Please choose a type of primary artifact :
1 : Methods
2 : classes
3 : Packages
4 : Test cases
5 : Requirements
6 : Exit
=> 5

Please type in the primary artifacts
of requirements (req1-req46) :
=> req12

You are selecting requirement(s):
req12 SRS_REQ_02_07

Please choose secondary artifacts of interest
to identify the impacts :
1 : Methods
2 : Classes
3 : Packages
4 : Test cases
5 : Requirements
=> 2 4

req12 - impacted classes are
Cls1:CDrivingStation
Cls2:Cpedal
Cls3:Cignition
Cls5:Ckeypad
Cls6:CInitManager
req12 - impacted test cases are
TCS5:TCASE-03-02-03
TCS7:TCASE-03-02-06
```

Listing 1. Snapshot of top-down traceability.

Listing 2 shows a snapshot of a bottom-up traceability when the user presented *mtd10* and *mtd20* as his primary impact and would like to view the impacted code in terms of methods, classes, test cases and requirements. *Catia* produced a list of the impacted artefacts for each *mtd10* and *mtd20*. Taking *mtd10* for example, it causes impact to methods *EngineIsOn(), InitManager()* and *KeypadHandler()*. In terms of the impacted classes, it affected the classes *Cignition* and *Ckeypad*, and so forth.



```
TR_OUTPUT1.txt - Notepad
File Edit Format View Help
Please choose a type of primary artifact :
1 : Methods
2 : Classes
3 : Packages
4 : Test Cases
5 : Requirements
6 : Exit
=> 1

Please type in the primary artifacts
 of methods(mtd1-mtd80) :
=> mtd10 mtd20

You are selecting method(s):
mtd10-CFcdTrip(),mtd20-setCruiseStatus()
Please choose secondary artifacts of interest
to identify the impacts :
1 : Methods
2 : Classes
3 : Packages
4 : Test Cases
5 : Requirements
=> 1 2 4 5

Mtd10 - impacted methods are
M-4-3-1:EngineIsOn()
M-4-3-2:InitManager()
M-4-5-1:KeypadHandler()

Mtd10 - impacted classes are
Cls3:Cignition
Cls5:Ckeypad

Mtd10 - impacted test cases are
TCS1:TCASE_02_01_01
```

Listing 2. Snapshot of bottom-up traceability.

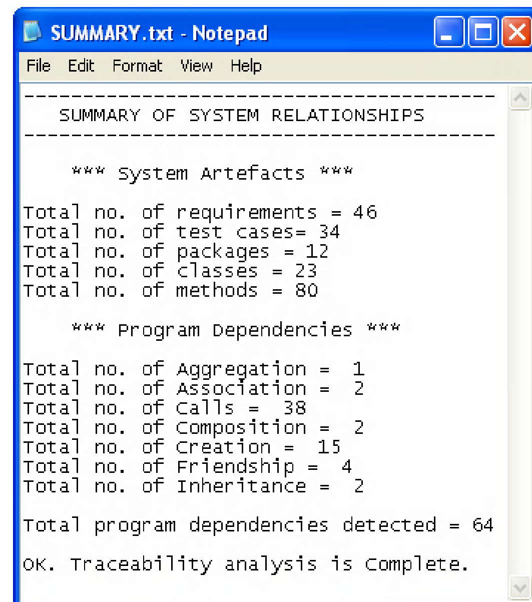
Listing 3 shows the summary of the component relationships of OBA system. In the first part of the summary, *Catia* shows the number of artefacts of each component type, while in the second part, it shows a total count of each type of program dependencies. The total count of all program dependencies was detected to be 64.

6. Related Work

We need to make clear that a software traceability and change impact are two different issues in literature and research undertaking, although both are related to one another. In change impact analysis, efforts and tools are more focused on code rather than software system. These include OOTME [12], CHAT [13] and OMEGA [6]. Object-Oriented Test Model Model Environment (OOTME) provides a graphical representation of object oriented system that supports program relationships such as inheritance, control structures, uses, aggregation and object state behavior. OOTME is suitable to support regression testing across functions and objects.

Change impact Analysis Tool (CHAT), an algorithmic approach to measure the ripple-effects of proposed changes is based on object oriented data dependence graph that integrates both intra-methods and inter-methods. OMEGA, an integrated environment tool for C++ program maintenance was developed to handle the message passing, class and declaration dependencies in a model called C++DG. The use of program slicing leads to recursive analysis of the ripple effects caused by code modification. McCabe [23] supports impacts at testing scenarios using call graphs of method-calls-method relationships, while, Code Surfer [22] provides an impressive impact analysis at the code level based on static analysis. The latter also allows a user to manipulate artefacts at any statements.

As the above mentioned approaches and tools are only limited to code model, we are not able to appreciate the real change impact as viewed from the system perspective. To manage a change impact analysis at a broader perspective, we have to associate them with traceability approach that requires a rich set of coarse and fine grained granularity relationships within and across different level of software models. Sneed's work [12] relates to a traceability approach by constructing a repository to handle maintenance tasks that links the code to testing and concept models. His concept model seems to be too generalized that includes the requirements, business rules, reports, use cases, service functions and data objects. He developed a model and a tool called GEOS to integrate all three software entities. The tool is used to select the impacted entities and pick up their sizes and complexities for effort estimation.



```
SUMMARY.txt - Notepad
File Edit Format View Help
----------------------------------------
   SUMMARY OF SYSTEM RELATIONSHIPS
----------------------------------------

    *** System Artefacts ***

Total no. of requirements = 46
Total no. of test cases= 34
Total no. of packages = 12
Total no. of classes = 23
Total no. of methods = 80

    *** Program Dependencies ***

Total no. of Aggregation =  1
Total no. of Association =  2
Total no. of Calls =  38
Total no. of Composition =  2
Total no. of Creation =  15
Total no. of Friendship =  4
Total no. of Inheritance =  2

Total program dependencies detected = 64

OK. Traceability analysis is Complete.
```

Listing 3. Summary of system relationships.

Bianchi *et al.* [2] introduces a traceability model to support impact analysis in object-oriented

environment. However, both [2, 19] do not involve a direct link between requirements and test cases to the code. Yet their work considers classes as the smallest artefacts of software components. Lindvall and Sandahl [15] present a traceability approach based on domain knowledge to collect and analyse software change metrics related to impact analysis for resource estimates. However, their work does not consider automated *concept location*. They relate some change requests to the impacted code in terms of classes but no requirements and test cases involved.

Our work differs from the above in that we attempt to integrate the software components that include the requirements, test cases, design and code. Our model and approach allow a component at one level to directly link to other components of any levels. Another significant achievement can be seen in its ability to support top down and bottom up tracing from a component perspective. This allows a maintainer to identify all the potential effects before a decision can be made. Our traceability integration manages to link the high level software components down to the implementation code with methods being considered as our smallest artefacts. This allows potential effects to become more focused.

7. Conclusions and Future Work

We apply the combination of both dynamic and static analysis techniques to integrate requirements to the low level components. Dynamic analysis is used to link the requirements and test cases to the implementation code, while static analysis is used to establish relationships between components within the code and design models. Our approach of traceability and impact analysis contributes some knowledge to the integration of both top-down and bottom-up impacts of system artefacts. This strategy allows provision for efficiency as the impacted artefacts can be directly accessed from an artefact perspective.

It seems that our approach would be more impressive if we could extend our traceability approach to include the detailed statements such as variables as our smallest artefacts. However, we have to bear in mind that considering those options would create large relationships among the software artefacts that may degrade the system performance. In large system, the maintainers are normally interested to know which classes or methods that need to be modified rather than the detailed statements of the code [15]. They would then intuitively recognize those detailed parts as they explore further.

Currently, our prototype uses text based exploration to describe the impacts. The application can be turned into GUI, which will be our future work. Last but not least, with all these basic infrastructures we provide, it can be used as a basis to measure impacts as measurement is another important issue that needs to be addressed in software maintenance. We reserve this objective for future work.

References

[1] Arnold R. S., "The Year 2000 Problem: Impact, Strategies and Tools," *Technical Report*, Software Evolution Technology, 1996.

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

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

[4] Bohner S. A. and Arnold R. S., "Software Change Impact Analysis for Design Evolution," *in Proceedings of the 8th International Conference on Software Maintenance and Reengineering*, IEEE CS Press, Los Alamitos, CA, pp. 292-301, 1991.

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

[6] Chen X., Tsai W. T., and Huang H., "Omega: An Integrated Environment for C++ Program Maintenance," *in Proceedings of International Conference on Software Maintenance*, IEEE Computer Society, pp. 114-123, 1996.

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

[8] Harmonization Working Group (HWG), incorporation with USA Department of Defense (DOD), *Overview and Tailoring Guidebook on MIL-STD-498*, Arlington, 1996.

[9] Horwitz S., Reps T., and Binkley D., "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[10] Ibrahim S. and Mohamad R. N., "Code Parser for C++," *Technical Report*, *Technical Report of Software Engineering*, Malaysian University of Technology, August 2004.

[11] Ibrahim S., Idris N. B., and Deraman A., "Case Study: Reconnaissance Techniques to Support Feature Location Using RECON2," *Asia-Pacific Software Engineering Conference*, *IEEE*, pp. 371-378, December 2003.

[12] Kung D., Gao J., Hsia P., and Wen F., "Change Impact Identification in Object-Oriented Software Maintenance," *in Proceedings of the International Conference on Software Maintenance*, pp. 202-211, 1994.

[13] Lee M., "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," *in Proceedings of the 34th International Conference on Object-Oriented Languages and Systems*, pp. 61-70, 2000.

[14] Lehman M. M., "Software Evolution," *Encyclopedia of Software Engineering*, pp. 1202-1208, 1994.

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

[16] Pressman R. S., *Software Engineering*: A Practitioner's Approach, 4th Edition, McGraw Hill, New York, 1997.

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

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

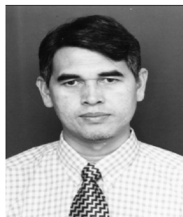
[19] Sneed H. M., "Impact Analysis of Maintenance Tasks for a Distributed Object-Oriented System," *in Proceedings of Software Maintenance*, IEEE, pp. 180-189, 2001.

[20] Swanson E. R. and Beath C., *Maintaining Information Systems Organizations*, John Wiley & Sons, New York, 1987.

[21] Turver R. J. and Munro M., "An Early Impact Analysis Technique for Software Maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 1, pp. 35-52, 1994.

[22] www.gramatech.com/products/codesurfer/index.html, 2004.

[23] www.mccabe.com, 2004.



Suhaimi Ibrahim is a senior lecturer in software engineering at the Centre for Advanced Software Engineering, University of Technology Malaysia. He is involved in several short term and National IRPA research projects on ICT and now pursuing his study for a PhD programme working on research related to software impact analysis. His research interests include reverse engineering, change impact analysis, requirement traceability, and software testing and software quality.

Norbik Bashah Idris is a professor and director of the Centre for Advanced Software Engineering at University of Technology Malaysia (KL Campus). He has helped to spearhead and establish real-time software engineering in Malaysia, working with multinationals and local industry especially on propagating software process quality. His research interests include real time systems, software engineering, and software maintenance.

Malcolm Munro is a professor of software engineering at the Durham University, UK. He has led a number of EPSRC funded projects including Release (Reconstruction of Legacy Systems), Visualising Software in a Virtual Reality Environment (VVSRE), Guided Slicing and Targeted Transformation (GUSTT), and distributed and dynamic visualisation generation (Jigsaw). He is also involved in research in Software as a Service (SaaS) and the application of Bayesian networks to software testing and program comprehension. His research interests include software visualisation, software maintenance and evolution, and program comprehension.

Aziz Deraman is a professor of software engineering and currently he is the dean of the Faculty of Information Science and Technology (FTSM), University Kebangsaan Malaysia. He also affiliated is various organizations and industries as an adviser, panel as well as a resident consultant in information technology. He maintains a diverse research interest including IT strategic planning, software process, and management and community computing.