A REQUIREMENTS TRACEABILITY TO SUPPORT CHANGE IMPACT ANALYSIS

Suhaimi Ibrahim,
Norbik Bashah Idris
Centre For Advanced Software Engineering,
Universiti Teknologi Malaysia,
Kuala Lumpur, Malaysia
**suhaimi@case.utm.my*
norbik@case.utm.my

Malcolm Munro
Department of Computer Science,
University of Durham,
United Kingdom
malcolm.munro@durham.ac.uk

Aziz Deraman
Fac. of Technology & Infor. System,
Universiti Kebangsaan Malaysia,
Selangor, Malaysia
a.d@pkrisc.ukm.my

Abstract

It is inevitable that a software undergoes some change in its lifetime. With some change requests comes a need to estimate the scope (e.g. size and complexity) of the proposed changes and plan for their implementation. 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 of different levels. It supports the top down and bottom up traceability in response to tracing for the potential 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: *Software traceability, impact analysis, change request, concept location*

1. Introduction

It is inevitable that a software undergoes some change in its lifetime. With some change requests comes a need to estimate the scope (e.g. size and complexity) of the proposed changes and plan for their implementation. The main problem to a maintainer is that seemingly small changes can ripple throughout the system to cause substantial impact elsewhere. A maintainer generally accomplishes change by analyzing the existing dependencies or relationships among the software components composing the software system.

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.

To implement impact analysis at a broader perspective is considerably hard to manage as it involves traceability within and across different models in the software life-cycle, such as from the design model to code model. 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 [2]. Such kind of traceability is called *requirements traceability* [2]. Pursuant to this, Turner and Munro [3] 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

supports many applications such as redocumentation, 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 a requirements traceability for change impact analysis from which we should be able to capture the impacts of a proposed change. What we mean 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 model. Section 3 discusses our approach to handle the artifacts and change impact analysis followed by the traceability techniques. Section 4 discusses our total traceability approach. Section 5 presents our case study and followed by some results and discussions. Section 6 presents some related work. Lastly, section 7 gives a conclusion and future work.

2. A Traceability Model

Figure 1 reflects the notion of our model to establish the relationships between artifacts. The thick arrows represent the direct relationships while the thin arrows represent the indirect relationships. Both direct and indirect relationships can be derived from static or dynamic analysis of component relationships. 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 [5].

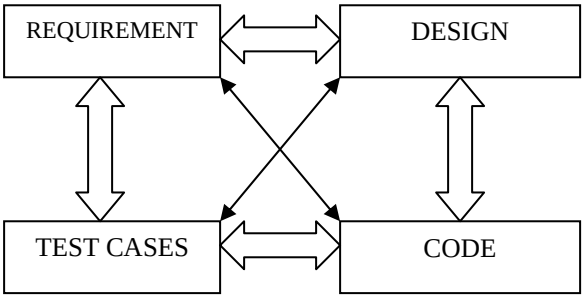

Figure 1: Meta-model of traceability system

2.1 Horizontal Traceability

We regard horizontal traceability as a traceability model of inter-artifacts such that each component (we call it as an artifact) in one level provides links to other components of 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 artifacts 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 artifact of type requirements (GR), design (GD), code (GC) or test cases (GT).

E - represents the relationships between artifacts from the point of view of an artifact of interest. This is identified by ER, ED, EC and ET.

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

i) Requirement Traceability

$ER \subseteq GR \times SGR$

SGR = GD ∪ GC ∪ GT

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

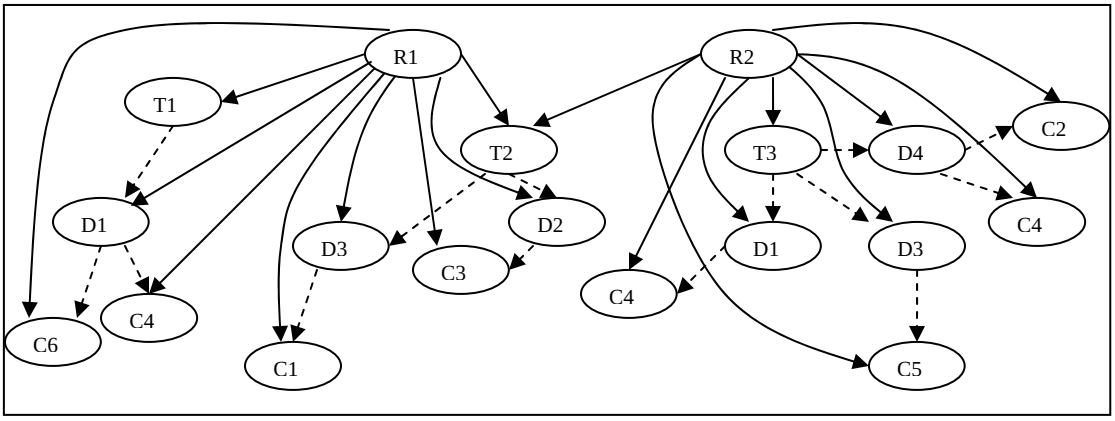

Figure 2: Traceability from the requirement perspective

ii) Design Traceability

ED ⊆ GD x SGD

SGD = GR ∪ GC ∪ GT

A design component relationship (ED) is defined as a relationship between a design component (GD) with other artifacts (SGD) of different levels. GD can be further decomposed into more detailed design components, if necessary.

iii) Test case Traceability

ET ⊆ GT x SGT

SGT = GR ∪ GD ∪ GC

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

iv) Code Traceability

EC ⊆ GC x SGC

SGC = GR ∪ GD ∪ GT

A code component relationship (EC) is defined as a relationship between a code component (GC) with other artifacts (SGC) of different levels. Code can be further decomposed into more detailed components.

2.2 Vertical Traceability

We regard a vertical traceability model for intra-artifacts of which an artifact provides links to other components within the same level of artifacts. In principle, we consider the following as our vertical platforms.

a) Requirement level
b) Test case level
c) Design level
d) 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 fulfill a 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 > 80 km/hr.
      i) Test case#: 1.1
      Code : TCASE-12-01-01
      Description: Launch the Auto Cruise while not on fifth gear.
      ii) 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*.

In the design and code, again there might exist some ambiguities of what artifacts should be represented as both may consist of some overlapping components e.g. should the classes be classified in the design or code ? To us, this is just a matter of development choice.

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 pay less attention on high level design abstracts to derive a traceability as this needs more research and would complicate our works. We apply the low level design abstracts that contain the software packages and class interactions. While, the code is to include all the methods, their contents and interactions.

3. Approach

3.1 Hypothesize Traces

We believe that there exists some relationships among the software artifacts in a system. We need to trace and capture their relationships somehow not only within the same level but also across different levels of artifacts before a impact analysis can be implemented. The process of tracing and capturing these artifacts 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 by manually through the available documentations and software models or by 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 potential effects in terms of the methods involved (TxM). Methods are the member functions in C++. We developed a tool support, called *CodeMentor* to identify the impacted code by instrumenting the source code prior to its execution [6].
4. Perform a static analysis on the code to capture the class-class (CxC), method-class (MxC), class-method (CxM) and method-method (MxM) dependencies.

We experimented using tool supports such as *McCabe* [7] and *Code Surfer* [8] to help capture the above program dependencies. However, other manual works as well as the need for other types of information saw us developing our own code parser called *TokenAnalyzer* [9].

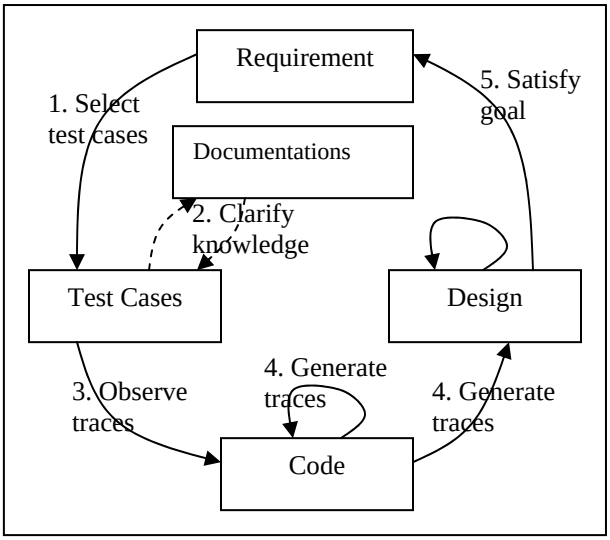

Figure 3 : Hypothesized and observed traces

3.2 Impact analysis

Some techniques are available to address impact analysis in code such as call graphs, data flows, ripple-effects and dependence graphs of program slicing [10]. However, the way these techniques are used may vary depending on the problem being addressed. In our case, we use the ripple-effects of call graphs and dependence graphs to manage impact analysis. We need to analyze from the program dependencies which artifacts cause effect to which artifacts. For example, in a method-to-method relationship of call invocations

M1 → M2
→ 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 on the other way around by picking up a callee and finding its corresponding callers for its potential effects. In other word, a change made to a callee may have a potential effect or ripple effect on its callers.

Table I: Structural relationships in C++

Relationships	Definitions	Examples
Call (MxM)	An operation (method) of the first class calls an operation of the second one. -----Impact ----- a1() ← b1()	class B {void b1();} class A { void a1() { B test; test.b1();} }
Composition (CxC)	A class contains a data member of some other class type. -----Impact ----- A ← B	class B {}; class A { B test; }
Create (CxM)	Some operation of the first class creates an object of the second class (instantiation) . -----Impact ----- a1() ← B	class B {}; class A { void a1() { B t; } }
Friendship i) (CxC) ii) (MxC)	Dependency from two classes. -----Impact ----- i) A ← B ii) A ← c1()	class B{}; class A { friend class B; friend int C:: c1(); }
Uses (CxM)	Data uses another data of different classes. -----Impact ----- a1() ← B a1() ← C	class B {int k;…} class C {int m;…} class A { B b; C c; void a1() {int m=b.k + c.m; } }
Inheritance (CxC)	Inheritance relation among classes. -----Impact ----- A ← B	class B {}; class A : public B { }
Association (CxM)	A class contains an operation with formal parameters that have some class type. -----Impact ----- a1() ← B	class B{}; class A { void a1 (B* par =0); }
Aggregation (CxM)	A class contains data members of pointer (reference) to some other class. -----Impact ----- a1() ← B	class B {}; class A { void a1() { B* test; } }
Define i) (MxC) ii) (CxM)	A class contains data members and member functions. -----Impact ----- i) A ← a1() ii) a1() ← A	Class A { Void a1(); }

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, not to its upper classes. Table I presents the types of relationship, with descriptions and examples of all possible dependencies in C++ that can contribute to change impact. In call relationship, there exists a method-method relationship as the called method *b1()* may affect the calling method *a1()*. In composition relationship, *test* is a data member of class B, would imply a change in B may affect class A, in a class-class relationship.

In create relationship, a change made in class B would affect the creation of objects in method *a1()*. Thus, we can say that class B may affect method *a1()* in a method-class relationship. In friend relationship, two types of friendship can occur, namely class friendship and method friendship. Class friendship results in a class-class relationship as it allows other class to access its class private attributes. While, method friendship results in a method-class relationship as it allows a method of other class to access its class private attributes.

In use relationship, we consider the use of data (i.e. variables or data members in C++) in the data assignment. Our objective here is to apply the use relationships that provide links between components of different methods and classes. In Table I, the use relationship involves data from other classes to implement a data assignment i.e. *k* and *m* from class B and C respectively. So, the class B and C would give impact to method *a1()* in a class-method relationship.

In Association relationship, a class contains an operation with formal parameters that have some class type. A change of class type in B would

affect method *a1()* in method-class relationship. In aggregation relationship, a class contains data members of pointer (reference) to some other class. So, a change in class B may affect class A in method-class relationship. Lastly, in define relationship observes i) a method-class relationship when one or more methods are defined in a class, so any change in a method simply affects its class ii) a class-method relationship when a change in a class implies an impact to its methods.

Our code parser, *TokenAnalyzer* was specially designed and developed to capture all these dependencies and form the designated tables of method-method, method-class, class-method and class-class relationships.

From our analysis on program dependencies, we can conclude that the artifact relationships and type relationships can be classified into several categories as appeared in Table II. Please note that for each artifact relationship, the types of relationship may appear explicitly and implicitly. Explicit relationships (shown in no brackets) mean the direct relationships we captured and obtained from the hypothesized traces. While, implicit relationships (shown in brackets) denote the indirect relationships we need to compute from the lower level artifact relationships.

Table II: Classifications of artifact and type relationships.

Artifact relationships	Types of relationship
CxC	composition, class friendship, inheritance, [create, association, aggregation, method friendship, uses, call, define]
CxM	create, association, aggregation, uses, define [call]
MxC	method friendship, define [call]
MxM	Call

The reason behind these indirect relationships is if there is an impact to a data would imply an impact to its method, and an impact to a method would imply an impact to its class it belongs to. Thus, there is a need to make them explicitly defined by transforming the lower level matrices into the higher level matrices e.g. to transform the MxM into MxC, CxM, CxC. With these new formations, we need to add into or update the existing designated tables we captured earlier. This gives us the broader potential effects as we move on to higher levels. Our point here is to allow users to visualize the impact at any level of relationships. The computation on this transformation is discussed in section 4.1.

3.3 Traceability Techniques

Intrinsically, traceability provides a platform for 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 modeled using UML [11].
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 [12].
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 [13].

We apply 1) and 3) in our traceability approach. We obtain the *explicit links* of artifacts including the transformation matrices. 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 [13]. 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 as appeared 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.

4. Total Traceability Approach

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 e.g. the requirements-test (RxT), test case-code (TxM), and so forth. The vertical relationships can occur at the code level (MxM - method interactions) and design level (CxC - class interactions, PxP - package interactions) respectively.

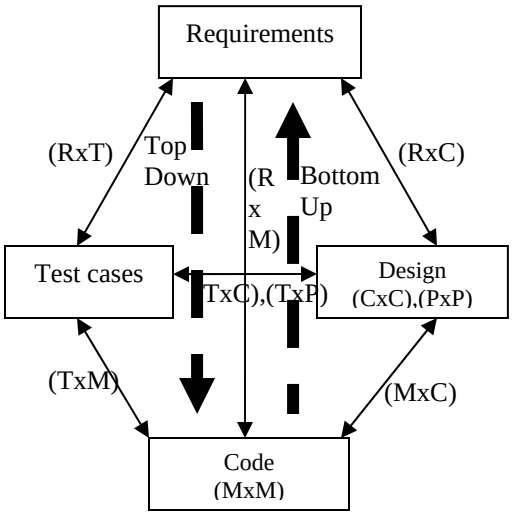

Figure 4 : System artifacts and their links

As we had the RxT and TxM from the hypothesized traces earlier, we can compute the RxM using a transitive closure,
(RxT) and (TxM) → (RxM)
such that if R impacts T and T impacts M, then R impacts M. The rest of the matrices can be computed or created as discussed in Section 4.1.

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 artifacts 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 artifacts from a lower to a higher level of artifacts e.g. from a method we can identify its impacted test cases and requirements. As the system goes along the way tracing for potential effects of either top-down or bottom-up traceability, it collects and sums up the size of metrics of the impacted artifacts. The metric sizes are measured in terms of the LOC (lines of code) and VG (value of program complexity). We need to assign each method and class with LOC and VG before hand with the help of a code parser, e.g. using *McCabe* tool.

4.1 Computing Matrices

Some matrix tables were made available from our previous hypothesized traces. In each table of binary relationships, the row parts represent the artifacts of interest while the column parts are the potential effects. For example, in MxM each method (in rows) produces some potential effects on some other methods (in columns). These potential effects are called footprints [14]. We apply a mapping table to create or transform a lower level artifact relationship into its higher level.

For example, to create a CxM table we first look into the existing MxM matrix that for each column, we use the mapping table to upgrade the rowed methods into the rowed classes. This means, the method footprints are automatically upgraded into the class footprints. If no method footprints exist during this transformation, means no corresponding class footprints take place. Similarly, we can establish the MxC relationships such that for each row of MxM, we upgrade the columned methods into the columned classes carrying the method footprints along to become the class footprints.

To create the CxC relationships, we can work on the basis of either CxM or MxC. Taking the CxM as an example, for each rowed class of CxM we upgrade the columned methods into the columned classes carrying the columned method footprints along to become the columned class footprints. On MxC, we can upgrade the rowed methods of MxC into the rowed classes carrying the rowed method footprints along to become the

rowed class footprints. We can apply the same concept to create other matrices such as CxP, PxC and PxP.

It is interesting to note that in our context of study, MxC is not the same as CxM. The reason is that the MxC is to see the potential effect of a method over other parts of the code in terms of classes. Whereas, the CxM is to see the potential effect of a class over other parts of the code in terms of methods. This is the reason why we cannot apply a transitive closure to some matrices. Similarly, We use the underlying RxM to transform it into RxC and RxP by upgrading the methods into classes and classes into packages. The same principle applies to TxM to transform it into TxC and TxP.

5. Case Study

To implement our model, we applied it to a case study of software project, called the Automobile Board Auto Cruise (OBA). OBA is an embedded software system of 4k LOC with 480 pages of documentation developed by the M.Sc group-based students of computer science at the Centre For Advanced Software Engineering, university of Technology Malaysia. OBA was built as an interface to allow a driver to interact with his car while on auto cruise mode such as accelerating speed, suspending speed, resuming speed, braking a car, mileage maintenance, and changing modes between the auto cruise and non-auto cruise.

The project was built with complete project management and documentations adhering to DoD standards, MIL-STD-498[15]. The software project was built based on the UML specification and design standards [16] with a software written in C++.

5.1 Results

We identified from the OBA project, 46 requirements, 34 test cases, 12 packages, 23 classes and 80 methods. Our system, *Catia* assumes that a change request has already been translated and expressed in terms of the acceptable artifacts i.e. requirements, classes, methods or test cases. *Catia* was designed to manage the potential effect of one type of artifacts at a time.

The system works such that given an artifact as a primary impact, *Catia* can determine its effects on other artifacts (secondary artifacts) in either

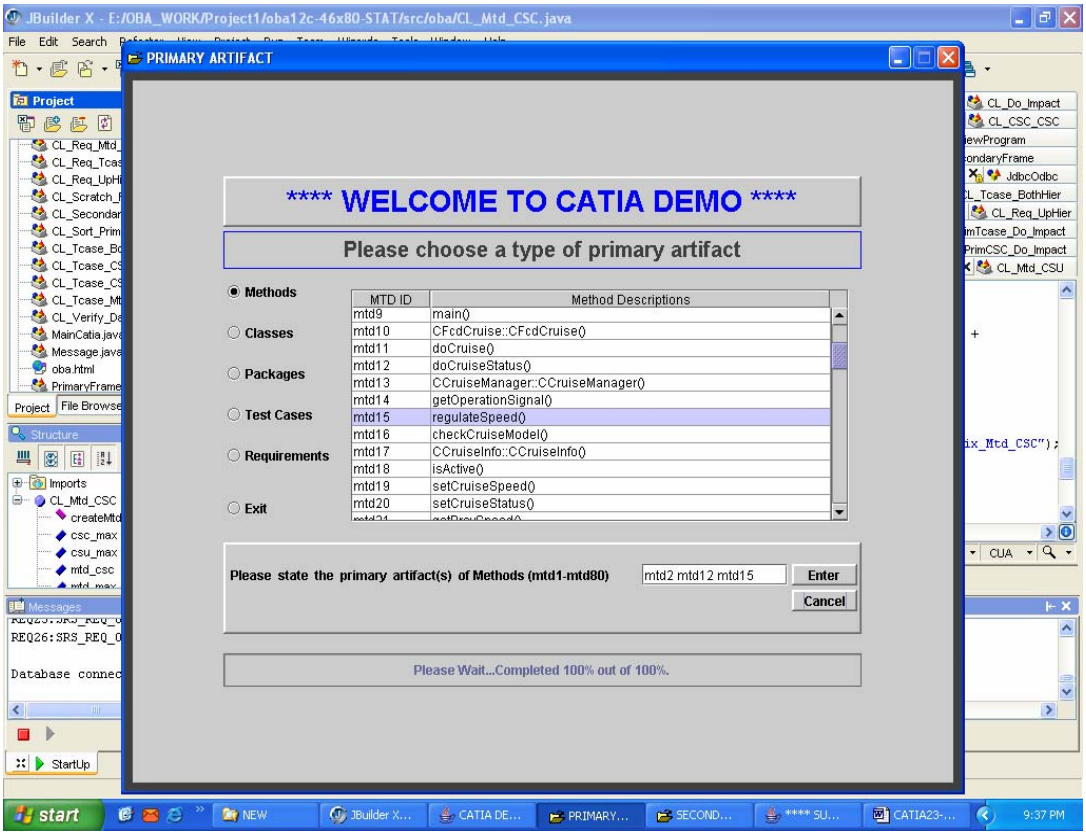

Figure 5 : First user interface of CATIA

top-down or bottom-up tracing. Figure 5 shows an initial user entry into the *Catia* system by selecting a type of primary artifacts followed by the detailed artifacts. The user had selected

In terms of the classes, mtd2 had caused 3 classes with their LOC (190) and VG (71). In packages, mtd2 had caused 2 packages of size LOC (279) and VG (100). Mtd2 involved in 4

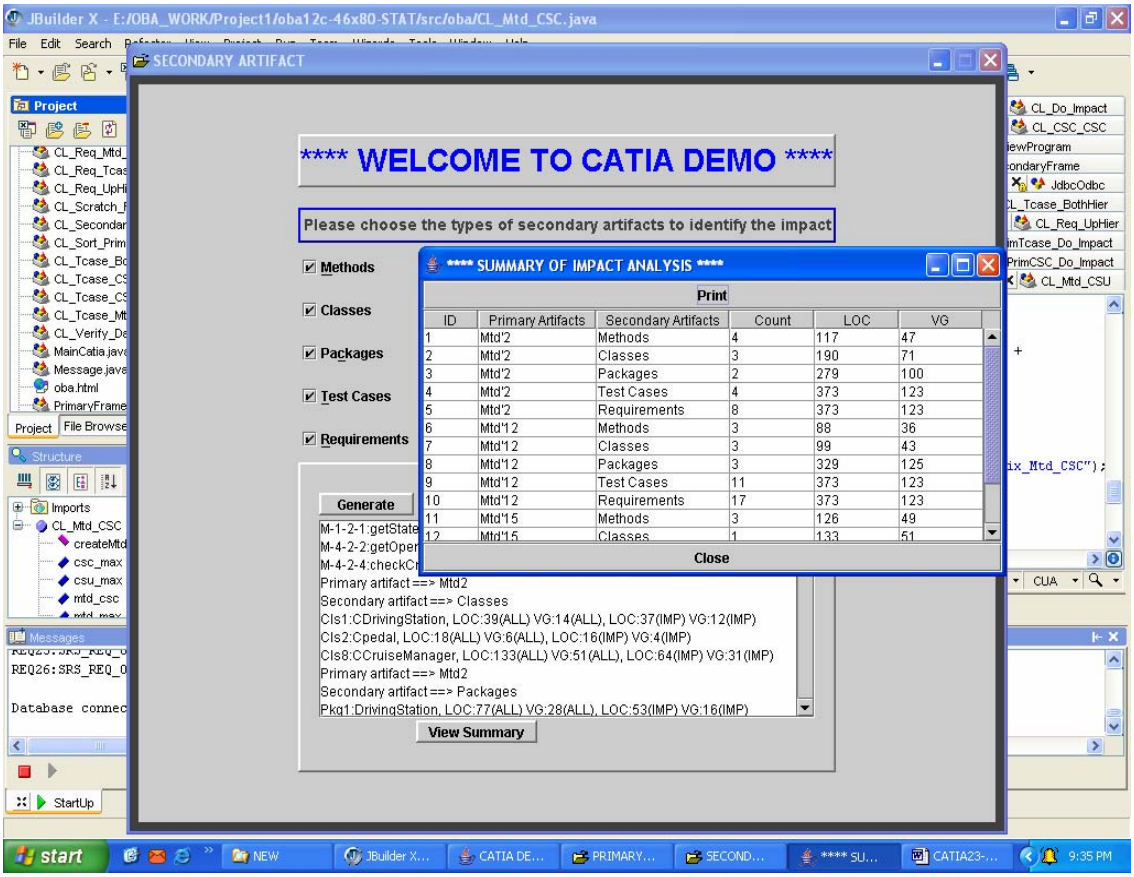

Figure 6 : Output of requirements traceability

methods as the primary artifact and chose the mtd2, mtd12 and mtd15 as the detailed methods of interest.

Figure 6 represents an output of the impacted artifacts and its summary after the user selected one or more types of secondary artifacts. In Figure 6, the user selected all the artifact levels as the secondary artifacts to visualize the impacts. After 'generate button', *Catia* then produced a list of impacted methods, classes, packages, test cases and requirements for each primary artifact chosen earlier. In the summary table (Figure 6), all the impacted artifacts associated to mtd2, mtd12 and mtd15 were shown in terms of counts, LOC and VG. Taking an example of mtd2, this requirement had caused potential effect to 4 methods in the system which brought to the total metrics of LOC (117) and VG (47).

test cases that took up the total metrics of LOC (373) and VG (123) of impacted methods to implement it. Please note that total LOC and VG for both test cases and requirements are the same as the requirements are characterized and executed by the test cases. *Catia* also provides a list of the detailed artifacts with their metric values to allow users to identify the impacted components.

5.2 Some Discussions and Lessons Learnt

There are some points we would like to highlight with respect to the implementation of our prototype.

1. DLL files (4 packages)
   DLL files only contain all the executable files as the reusable software packages and no source code available. As this is the case, there is no way for us to neither using the

McCabe nor *CodeMentor* to capture the methods and classes within the DLL packages. Thus, we treated the DLL files as special packages with no metric values.

2. Self impact
   There were cases in the (MxM) and (CxC) relationships, a component only made an impact on itself not to others. This is due to the fact that a method or class was designed just to provide a service rather than call invocation to others.
3. Non functional requirements (1 requirement)
   There was a timing requirement, STD_REQ-02-19 stated that "fuel inlet mechanism should respond in less than 0.5 seconds on actions by a driver". This requirement had no impact on other classes or methods. This is due to the fact that the timer is produced by the kernel operating system not by any other classes or methods. The result of timing may be needed by some classes or methods for some tasks e.g. in speed calculation, but no action being carried out by any methods or classes to check the violation of timing. The developers verified this requirement manually by running a test driver to spy the timing at the background mode. As no program verification can be made on this particular issue, we dropped this type of requirement from our work.

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 [17], CHAT [18] and OMEGA [19]. OOTME (Object-Oriented Test Model Environment) 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.

CHAT (Change Impact Analysis Tool), 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 [7] supports impacts at testing scenarios using call graphs of method-calls-method relationships, while, Code Surfer [8] provides an impressive impact analysis at the code level based on static analysis. The latter also allows a user to manipulate artifacts 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 [20] 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.

Bianchi *et al.* [21] introduce a traceability model to support impact analysis in object-oriented environment. However, both [20,21] do not involve a direct link between requirements and test cases to the code. Yet their works consider classes as the smallest artifacts of software components. Lindvall and Sandahl [12] present a traceability approach based on domain knowledge to collect and analyze software change metrics related to impact analysis for resource estimates. However, their works do 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 artifacts. This allows potential effects to become more focused.

6. Conclusion 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 artifacts. This strategy allows provision for efficiency as the impacted artifacts can be directly accessed from an artifact 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 artifacts. However, we have to bear in mind that considering those options would create large relationships among the software artifacts 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 [12]. They would then intuitively recognize those detailed parts as they explore further.

Currently our prototype provides traceability infrastructures with some measurements to support change impact. It can be enhanced further to support GUI linked to program views for better user interface of which we reserve for future work.

Acknowledgements

This research is funded by the IRPA of Malaysian Plan (RM-8) under vot no. 74075. The authors would like to thank the Universiti Teknologi Malaysia, the University of Durham, the Universiti Kebangsaan Malaysia and individuals for their involvement and invaluable comments and suggestions throughout the development and review process.

References

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

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

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

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

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

[6] Ibrahim S., Idris N.B., Deraman A., 2003. Case study: Reconnaissance techniques to support feature location using RECON2, Asia-Pacific Software Engineering Conference, IEEE, pp. 371-378.

[7] http://www.mccabe.com

[8] http://www.gramatech.com/products/codesurfer/index.html

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

[10] S. Horwitz, T. Reps, and D. Binkley, 1990. Interprocedural slicing using dependence graphs, ACM Transactions on Progrramming Languages and Systems, 12(1), pp. 26-60.

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

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

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

[14] A. Egyed, 2003. A Scenario-Driven Approach to Trace Dependency Analysis, IEEE Transactions on Software Engineering, vol 29(2).

[15] Joint Logistics Commanders on Computer Resource Management, Overview and Tailoring Guidebook on MIL-STD-498, Arlington, 1996.

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

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

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

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

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