Applying Dynamic Change Impact Analysis in Component-based Architecture Design

Tie Feng
College of Computer Science and Technology
Jilin University, China
Changchun Jilin 130012
fengtie@jlu.edu.cn

Jonathan I. Maletic
Department of Computer Science
Kent State University, US
Kent Ohio 44242
{hkagdi, collard, jmaletic}@cs.kent.edu

Abstract

Change impact analysis plays an important role in maintenance and evolution of component-based software architecture. Viewing component replacement as a change to composition-based software architecture, this paper proposes a component interaction trace based approach to support dynamic change impact analysis at software architecture level. Given an architectural change, our approach determines the architecture elements causing the change and impacted by the change. Firstly, component-based software architecture and component interaction trace are defined. An algorithm for generating component interaction trace from static structure model of software architecture and UML sequence diagram is provided. Secondly, the taxonomy of changes on composition-based software architecture is presented, according to which a set of impact rules are suggested to determine the transfer of the changes in component and among components. Thirdly, by performing slicing on component interaction traces according to impact rules, the impact analysis results are obtained. Finally, the architecture design of SOCIAT, a tool supporting our approach, is developed and explained.

Keywords: Change impact analysis, software architecture, component composition, program slicing

1. Introduction

Frequent changes and expansion to legacy system cause unexpected influence on the other parts of it. Component-based software engineering is an effective way to alleviate this problem to some extent, because the internal information of the component is not available to the public and components only communicate through their interfaces.

However, it is never a simple procedure to retrieve proper components from component repository and replace the old ones with them so that changed system functionality and performance are obtained. In many cases, the existing components have less or more services provided and requiring than expected ones. Besides customizing the retrieved components or developing new components, change impact analysis is also an effective way to support maintenance and evolution of component based software development by determining subsequent changes, if component replacement is considered as a change to legacy system.

Change impact analysis is to identify the potential consequences of a change or estimate what needs to be modified to accomplish a change [1]. Much effort has been made to study static and dynamic change impact analysis technologies at the code level of software systems. However, very few techniques have been proposed to support change impact analysis on component composition at software architecture level, and much fewer research have been conducted to support dynamic change impact analysis on component composition at this abstract level.

On the other hand, composition based software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components [2]. It offers a way to partition complex systems into well-defined parts [3, 4]. We try to apply change impact analysis to composition based software architecture in order to support software architecture maintenance and evolution.

It is important to understand dynamic analysis on component composition at architecture level to determine our research scope. Our approach considers the system dynamic behaviors at a rather higher level and focus on studying effects on software product of adding or removing components at runtime. Starting from component interaction traces describing the interaction sequences among components through their interfaces, our approach study the impacts of changes on architectural elements according to a special system execution with a set of given inputs or interaction triggered by user.

In order to develop dynamic change impact analysis technique on composition based software architecture, interaction mechanism and composition model among components of architecture are required. In this paper,

Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)
0-7695-2611-X/06 $20.00 © 2006 IEEE

components are integrated together through interfaces and methods in them. Moreover, a dynamic component composition model is defined and derived from static structure model of component composition and a set of UML sequence diagrams describing object interaction. Then the dynamic change impact analysis is accomplished through slicing on component interaction traces. The slicing criterion is constructed on intra-component and inter-component impact rules for taxonomy of changes of interface and methods of components.

2. Composition based software architecture

The software architecture of a system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. In this paper, composition based software architecture is defined as a special kind of software architecture in which component is considered as the first-class entity and connectors is not obviously modeled. Components are integrated together through interfaces to achieve special system functionality.

2.1. Component and interface

Each component comprises a set of interfaces and the application logic [5]. Interfaces can be divided into two categories: provider interface providing services to other components or external environment, and requirement interface requiring services from other components or external environment. Each interface is consist of a set of methods indicating the identifications of provided services or required ones. The structure of an example component C is shown in Fig. 1. Component C has 2 provider interfaces (I_{p1} and I_{p2}) and 2 requirement interfaces (I_{r1} and I_{r2}). Interface I_{p1} is composed with 2 methods, each of which indicates a service C provides through interface I_{p1}.

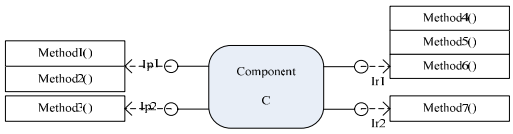

Fig. 1. An example of component

To demonstrate our approach, we introduce following primitives in Table 1.

Table 1. Primitive for Component, Interface and Method

Primitives	Parameters	Return
Interface	*C: Component*	*A set of interfaces of component C*
PorR	*I: Interface*	*Provider if I is a provider interface, Requirement otherwise*
Method	*I: Interface*	*A set of methods of interface I*

Definition 1: Initial Component. Let C be a component, if for $\forall$ I$\in$ Interface(C), PorR(I)=Requirement, then we say C is an initial component.

Definition 2: Final Component. Let C be a component, if for $\forall$ I$\in$ Interface(C), PorR(I)=Provider, then we say C is an final component.

Components communicate through their interfaces and there are three kinds of relationships between two interfaces: interface equivalence, interface satisfaction and interface inclusion.

Definition 3: Interface Equivalence. Let C_1 and C_2 be two components, if $\exists$ $I_1 \in$ Interface(C_1) $\wedge$ $I_2 \in$ Interface (C_2), so that (PorR(I_1)=PorR(I_2)) $\wedge$ (Method(I_1) = Method(I_2)), then we say interface I_1 is equivalent to interface I_2, which is denoted as $I_1 \equiv I_2$.

Definition 4: Interface Satisfaction. Let C_1 and C_2 be two components, if $\exists$ $I_1 \in$ Interface(C_1) $\wedge$ $I_2 \in$ Interface(C_2), so that (PorR(I_1) =Provider $\wedge$ PorR(I_2)=Requirement) $\wedge$ (Method(I_1) $\supseteq$ Method(I_2)), then we say interface I_1 satisfies interface I_2, which is denoted as $I_1 \triangleright I_2$.

Definition 5: Interface Inclusion. Let C_1 and C_2 be two components, if $\exists$ $I_1 \in$ Interface(C_1) $\wedge$ $I_2 \in$ Interface(C_2), so that (PorR(I_1)=PorR(I_2)) $\wedge$ (Method(I_1) $\supseteq$ Method(I_2)), then we say interface I_1 includes interface I_2, which is denoted as $I_1 \geq I_2$.

From above definitions, it is obvious that interface equivalence is a special example of interface inclusion. That is to say if $I_1 \geq I_2 \wedge I_2 \geq I_1$, then $I_1 \equiv I_2$.

2.2. Static structure model of composition

Static structure model is composed of components and connection among components according to interface inclusion and interface satisfaction. We develop two composition rules to configure components as follows.

Rule 1: Let I_1 and I_2 are two interfaces of two different components C_1 and C_2, if it is satisfied that ($I_1 \geq I_2$) $\wedge$ (PorR(I_1)=PorR(I_2)=Provider), then we create a new provider interface I'=I_1.

Rule 2: Let I_1 and I_2 are two interfaces of two different components C_1 and C_2, if it is satisfied that $I_1 \triangleright I_2$, then the binding of the two interfaces indicates that the service required by C_2 through interface I_2 is provided by C_1 through interface I_1.

For example, Fig. 2 shows an example of application of rule 1 and rule 2. Rule 1 is applied between $I_{p1}$and I_{p2} and a new provider interface I_p is created. Rule 2 is applied between I_p and I_{r3}, I_{p3} and I_{r1}, I_{p4} and I_{r2}.Of course,

the application of rules must meet the prerequisite of the rules.

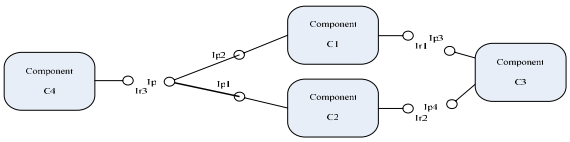

Fig. 2. An example of static structure model of composition-based software architecture

2.3. Dynamic interaction model of composition

Dynamic interaction model represents the interactions among components through their interfaces. Taking the static structure model shown in Fig. 2 for instance, two possible dynamic interactions among components are shown in Fig. 3 (a) and Fig. 3 (b). In this paper, we introduce a new concept of component interaction trace to describe the dynamic behavior model of composition. It is the foundation of our approach on change impact analysis.

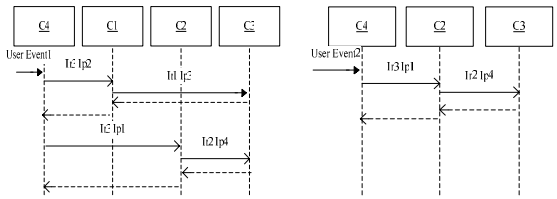

Fig. 3 (a). One possibility **(b).** Another possibility

Definition 6: Component Interaction Trace (CIT).

<CIT>=<CompName>
|<CIT>"("<InterfacePair>")"<CompName>
|<CIT><Mark>"("<InterfacePair>")"<CompName>
<Mark>="r"|"r"<Mark>
<InterfacePair>=<RequirementIName>":"<ProvIName>

"r" indicates the return of control.

For example, the component interaction trace for dynamic interaction models shown in Fig. 3 (a) is C_4 (I_{r3}:I_{p2}) C_1 (I_{r1}:I_{p3}) C_3 rr (I_{r3}:I_{p1}) C_2 (I_{r2}:I_{p4}) C_3 rr and the component interaction for Fig.3 (b) is C_4 (I_{r3}:I_{p1}) C_2 (I_{r2}:I_{p4}) C_3 rr.

In practical development of software systems using UML, the static structure model of composition and the interaction model of objects could be modeled through modified UML component diagrams and sequence diagrams. However, there is no automatic method generating dynamic interaction model of composition and component interaction trace. To perform dynamic change impact analysis on CIT, we present an algorithm as follow to automatically generate one CIT from static structure model and sequence diagram.

Input: Static Structure Model(SSM) and a Sequence Diagram
Output: CIT describing component dynamic interactions

1 *Traverse the sequence diagram to create a vector V in which each element is a message.*
2 *Every message has the structure of <BeginObject, MessageName, EndObject>.*
3 *CIT=Component which includes the BeginObject of the first message.*
4 *For each V[i]*
5 *For each component C_j in SSM*
6 *For each service interface Ip_k of C_j*
7 *If V[i].MessageName∈ Method(Ip_k)*
8 *For each component C_m in SSM satisfying that $\exists Ir \in Interface(C_m) \wedge (Ip_k \triangleright Ir)$*
9 *Put all methods in all service interfaces of C_m into a set S_1.*
10 *u=i; Put all messageNames received by V[u].BeginObject currently into a set S_2.*
11 *While $(u>=0) \wedge (S_1 \cap S_2 = \Phi)$*
12 *u=u-1.*
13 *If $(V[u].BeginObject \notin C_m)$ CIT=CIT+"r".*
14 *Put all messageNames received by V[u].BeginObject into S_2.*
15 *If (u>=0)*
16 *CIT=CIT+"("+Ir+":"+ Ip_k+")"+ C_j.*
17 *Return CIT*

In this algorithm, it is very easy to determine current component and its current provider interface by continuously analyzing the object message trace (line 1 to line 7). However, to determine which requirement interface of previous component is invoking service from current provider interface of current component, we have to retrace the sequence diagram until the message indirectly sent to current component and indirectly triggering the provider interface of current component is found (line 8 to line 15).

This algorithm is provider interface driven because the requirement interface information is not described in sequence diagram although it is defined in static structure model of composition.

3. Changes and impacts on composition based software architecture

First of all, we give taxonomy of changes on composition. Then the transition rules for inter-component and intra-component are studied according to various changes.

3.1. Taxonomy of changes

Our approach focuses on the changes and impacts of elements of components. Concretely, the provider interfaces, requirement interfaces, methods in provider interfaces and methods in requirement interfaces of components are researched.

● Atomic Changes. An atomic change is that applied on components or interfaces and can't be decomposed into smaller changes. The atomic changes on composition based software architecture are listed in table2, which are self-explanatory.

Table 2. Atomic Changes on Composition Based Software Architecture

AEPI	Add an Empty Provider Interface to component
DEPI	Delete an Empty Provider Interface from component
AERI	Add an Empty Requirement Interface to component
DERI	Delete an Empty Requirement Interface from component
AMPI	Add a Method signature to a Provider Interface
DMPI	Delete a Method signature from a Provider Interface
AMRI	Add a Method signature to a Requirement Interface
DMRI	Delete a Method signature from a Requirement Interface

● Composite Change. A composite change, obtained through iterative transitions of atomic changes, is much more practical in change impact analysis. Sometimes, an individual atomic change to software architecture has not practical meanings. For example, typical addition of an empty provider interface is often followed by some additions of methods to it so that new services could be provided through new interface. Four kinds of practical composite changes on composition base software architecture are studied and listed in Table 3.

Table 3. Typical Composite Changes on Composition Based Software Architecture

API=AEPI+n*AMPI	Add an empty Provide Interface to component, then add n methods to it
DPI=n*DMPI+DEPI	Delete n methods from an provider interface, then delete that interface
ARI=AERI+n*AMRI	Add an empty Requirement Interface to component, then add n methods to it
DRI=n*DMRI+DERI	Delete n methods from an Requirement interface, then delete that interface

Therefore, our approach focuses on 8 kinds of practical changes to software architecture elements: API, DPI, ARI, DRI, AMPI, DMPI, AMRI and DMRI. The first 4 changes are related to interfaces of components and the later 4 are related to methods of interfaces.

3.2. Intra-component impact rule and inter-component impact rule

As mentioned in above section, our approach focuses on the changes such as adding and deleting interfaces and methods. It is time to answer following questions: (1) what impacts would be produced on the same component and neighbor components? (2) How does the impact transmit?

● Intra-Component impact transition rule. Generally, to provide a service, a component would always require some services provided by other components. At the same time, a required service is always used to implement one or more provided services. Except for initial components and final components (see definition 1 and 2), a many-to-many mapping exists between methods in requirement interface and methods in provider interface. To initial components or final components, the changes has no impact on the other interfaces and methods of the same component because they have only requirement interfaces or provider interfaces.

For example, table 4 shows a possible intra-component method mapping of a single component shown in Fig. 1. C depends on Method5(), Method6() and Method7() to provide Method1().C depends on Method4()to provide Method2(). According to table 4, it is easy to conclude that interface Ip1 depends on Ir1, Ir2, while Interface Ip2 depends on Ir1 shown in Fig. 4.

Table 4. A Mapping among Methods between Provider Interfaces and Requirement Interfaces

	Method4()	Method5()	Method6()	Method7( )
Method1()		X	X	X
Method2()	X			X
Method3()		X		

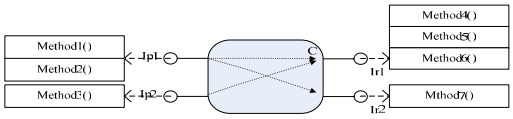

Fig. 4. Dependency among interfaces in a single component

With information on method mapping and interface dependency, it is easy to determine the intra-component impacts. For example, the Ip1 will be impacted by deleting the interface Ir2. Both interface Ip1 and Ip2 will be impacted by deleting Ir1. Generally,

Rule 3: DMRI or DRI cause DMPI or DPI on the same component.

Rule 4: AMPI or API cause AMRI or ARI on the same component.

● Inter-Component impact transition rule.

Definition 7: Neighbor. Let C_1 and C_2 be two components, if $\exists I_1 \in \text{Interface}(C_1) \wedge I_2 \in \text{Interface}(C_2) \wedge I_2 \triangleright I_1$, then we say C_1 is left Neighbor of C_2 and C_2 is right neighbor of C_1.

According to definition 7, the concept of neighbor is for static structure model of component composition. For any segment in component interaction trace with form

$C_m(I_{ri}:I_{pj})C_n(I_{rs}:I_{pt})C_o$, there exist that C_m is the left neighbor of C_n and C_o is the right neighbor of C_n. It is obvious that the requirement interface I_{ri} of component C_m will be impacted by deleting the Ipj or methods in and I_{pj}. Provider interface I_{pt} of component C_o will be impact by adding methods into requirement interface Irs of component C_n. For the trace segment with return marks, it is necessary to refer to static structure model to decide neighbors. Taking the segment of component interaction trace shown in Fig. 3 (a), $C_3rr(I_{r3}:I_{p1})C_2$, for instance, the left neighbor of component C_2 is C_4 but not C_3. Generally,

Rule 5: DMPI or DPI of a component cause DMRI or DRI of its left neighbor.

Rule 6: AMRI and ARI of a component cause AMPI or API of its right neighbor.

4. Perform dynamic change impact analysis

4.1. Slices for component interaction trace

Definition 8: Sub-component and super-component. Let C_1 and C_2 be two components. We call C_1 is the sub-component of C_2 and C_2 is the super-component of C_1 if and only if $\forall I \in$ Interface(C_1), $\exists I' \in$ Interface(C_2), so that $I' \geq I$.

From definition 8, it is concluded that the sub-component of component C would be obtained by deleting some interfaces or interface methods from it. The super-component of component C would be obtained by adding some interfaces or interface methods to it.

Definition 9: Interface Slicing Criterion. A interface slicing criterion for component interaction trace is a tuple with 3 elements (C,I,X) in which C is a component, I∈ Interface(C), and X∈ {+,-} indicating adding if X="+", or deleting if X="-".

Definition 10: Method Slicing Criterion. A method slicing criterion for component interaction trace is a tuple with 4 elements (C,I,M,X) in which C is a component, I∈ Interface(C), M∈ Method(I), and X∈ {+,-} indicating adding if X="+", or deleting if X="-".

Definition 11: Interface Slice. (1) A backward interface slice for component interaction trace T, S_{ibt}, on a given slicing criterion (C,I,X) consists of all sub-components and super-components of C in T that might directly or indirectly cause the addition or deletion of interface I; (2) A forward interface slice for component interaction trace T, S_{ift}, on a given slicing criterion (C,I,X) consists of all sub-components and super-components of C in T that might be directly or indirectly affected by adding or deleting interface I; (3) A unified interface slice for component interaction trace T, S_{iut}, on a given slicing criterion (C,I,X) equals the combination of S_{ibt} and S_{ift}, i.e. $S_{iut} = S_{ibt} \cup S_{ift}$.

Definition 12: Method Slice. (1) A backward method slice for component interaction trace T, S_{mbt}, on a given slicing criterion (C,I,M,X) consists of all sub-components and super-components of C in T that might directly or indirectly cause the addition or deletion of method M of interface I; (2) A forward method slice for component interaction trace T, S_{mft}, on a given slicing criterion (C,I,M,X) consists of all sub-components and super-components in T that might be directly or indirectly affected by adding or deleting method M of interface I; (3) A unified method slice for component interaction trace T, S_{mut}, on a given slicing criterion (C,I,M,X) equals the combination of S_{mbt} and S_{mft}, i.e. $S_{mut} = S_{mbt} \cup S_{mft}$.

With CIT, intra-component transition impact rule, inter-component transition impact rule, interface slicing and method slicing, we can determine all components, interfaces and methods which would cause given changes or are impacted by a given change. In section 4.2, we illustrate the concrete analysis method with two examples.

4.2. Determine impacted architectural elements

To CIT $C_4(I_{r3}:I_{p2})C_1(I_{r1}:I_{p3})C_3rr(I_{r3}:I_{p1})C_2(I_{r2}:I_{p4})C_3rr$, obtained in section 2.3, and a concrete change DPI(Ip2) of component C1, we could get a interface slicing criterion δ =(C_1,I_{p2},-). We perform interface backward slicing on slicing criterion δ to create a set consisting of the sub-components of those components in set $\{C_1,C_2,C_3,C_4\}$. Equally, we try to determine possible components and their interfaces whose changes would cause the deleting of I_{p2}.

According to rule 3 and supposing there is dependency between I_{r1} and I_{p2} by referring to the mapping table among requirement methods and provider methods of component C_1, it is concluded that DRI(I_{r1}) of the same component C_1 would cause the deleting of I_{p2}. Then what change would cause the DRI(I_{r1})? According to Rule 5, change of DPI(I_{p3}) on C_1's right neighbor, C_3 would cause DRI(I_{r1}). Because component C_3 doesn't have right neighbor, the impact transition stops. So we get all components, interfaces and methods that could cause the given change DPI(I_{p2}) on component C_1. Remaining these component and interfaces related to the change and removing those interfaces unrelated to that change, the trace's backward slicing on slicing criterion δ is $C_1\{I_{p2},I_{r1}\}C_3\{I_{p3}\}$. In fact, the stop of impact transition not only would be caused by return marks, but also would be caused by lack of dependency relationship in mapping table of provider interfaces and requirement interfaces.

5. SOCIAT-software composition oriented change impact analysis tool

The architecture of SOCIAT, which is shown in Fig. 5, includes 3 sub-systems: CIT generator, slicing performer

and regression test suite scheduler. SOCIAT mainly has following three functionalities: (1) automatically generating component interaction traces according to a given change; (2) producing component set with indication of interfaces and methods that impact or be impacted by a given change; (3) scheduling regression test suite.

There are plenty of UML modeling tools that could be used to model static structure model of components composition and sequence diagram, such as Rational ROSE. Working principles of CIT generator and slicing performer have been illustrated in above sections. Regression test suite scheduler is used to verification that unchanged parts of system behavior same as before and changed parts of system behavior work as specification. The working principle of this sub-system is omitted in this paper because of the limitation of space.

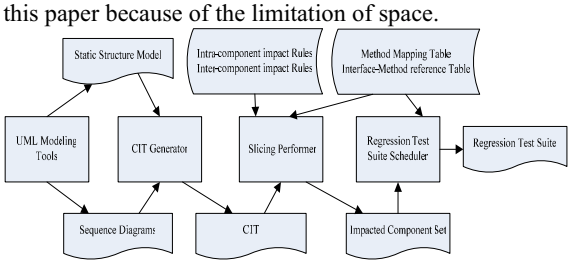

Fig.5. Architecture of SOCIAT

6. Related works

Previous research in change impact analysis of software system has been carried out with various approaches such as transitive closure on call graphs, static slicing, dynamic analysis and integration of static analysis and dynamic analysis. Although these classical impact analyses previous to ours provide some inspiration to our research, they only focus on code-level.

Jianjun Zhao [6] presented an approach to assess the effect of changes of software architecture by analyzing its Wright specification and to compute slices using flow graph-based technique.

L. C. Briand et al. [7] proposed a methodology supported by a prototype tool (iACMTool) to tackle the impact analysis and change management of analysis/design documents in the context of UML-based development. Opposite to our approach, their research focuses on consistency check among documents which is a kind of traceability analysis [1]. However, our approach belongs to the category of dependency analysis [1].

Another work that is similar to ours is that presented by Stafford et al. [8], who introduced a software architecture dependence analysis technique called chaining to support software architecture development such as debugging and testing. However, in contrast to our intra-component and inter-component impact rules based slice, they used a table-based approach to identify the chain sets.

7. Conclusion

An important problem demanding prompt solution with today's software development is to deal with the changes of software products. In this paper, a dynamic change impact analysis at software architecture level is proposed to support software architecture evolution.

The main contributes of this paper are listed as follows:

(1) Composition based software architecture is presented.

(2) Changes and impacts on composition based software architecture are explored.

(3) Interface slicing and method slicing based dynamic change impact analysis procedure is proposed.

(4) The architecture of SOCIAT, software composition oriented change impact analysis tool, is designed to support our approach.

8. References

[1] Bohner SA, Arnold RS, Software Change Impact Analysis, IEEE Computer Society Press: Los Alamitos CA, 1996.

[2] Shaw Mary, Garlan David. Software Architecture: Perspective on an Emerging Discipline. Prentice-Hall: Englewood Cliffs NJ, 1996.

[3] David Hemer, A formal approach to component adaptation and composition, Proceedings of the Twenty-eighth Australasian conference on Computer Science, Newcastle, Australia, Pages: 259 – 266, 2005

[4] Steffen Göbel, Encapsulation of structural adaptation by composite components, Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, Newport Beach, California, Pages: 64 – 68, 2004.

[5] Luca de Alfaro and Thomas A. Henzinger, "Interface Theories for Component-Based Design", Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp.148-165.

[6] Jianjun Zhao, Hongji Yang, Liming Xiang and Baowen Xu, Change impact analysis to support architectural evolution, Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, pp. 317-333, 2002.

[7] L. C. Briand, Y. Labiche, L. O'Sullivan, "Impact analysis and change management of UML models", Proceedings of the International Conference on Software Maintenance (ICSM'03), Amsterdam, Netherlands, pp.256-265, Sep. 2003.

[8] J.A. Stafford, A.L. Wolf and M. Caporuscio. "The Application of Dependence Analysis to Software Architecture Descriptions", Lecture Notes in Computer Science, Vol. 2804 Bernardo, Marco; Inverardi, Paola (Eds.) 2003, pp. 52-62.