Software evolution control
Towards a better identification of change impact propagation

Adeel Ahmad², Henri Basson³, Mourad Bouneffa³
Laboratoire d’Informatique du Littoral
Maison de la recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
¹ adeel.ahmad@mel-etu.univ-littoral.fr, {²basson, ³bouneffa}@lil.univ-littoral.fr

Abstract—The software evolution is often a continuous process necessary to avoid a short longevity of software use. The evolution control requires analysis to be processed respectively before and after software change implementation. First, an a priori change impact analysis is required to help decisions takers by estimation of intended impact and its propagation to other software components. Secondly, an a posteriori change analysis is destined to estimate to what extent the goals of implemented changes have been effectively reached. Both invoked analysis of change impact require an exhaustive description of individual software components as well as their various types of interdependency. In our paper, we propose an exhaustive stratified description of software components using formal modeling, aiming at providing better change impact analysis. The proposed description focuses on software elements which may play role in change impact and the propagation of its effects from directly modified component to other components of the whole software.

Keywords - software evolution; change impact analysis; modeling; change impact visualization;

I. INTRODUCTION

Software evolution control is an important task aimed at minimizing the change side effects. Such a control needs a stratified and exhaustive descriptions related to changes in software artifacts. The process of change incorporation in a developed software is generally time and effort consuming. It is mainly because of the difficulty in understanding of various software aspects which could be concerned by applied software modifications.

Software evolution can occur on random intervals of time, following a cycle of changes, retests and implementations. Currently, most of the research work in software evolution is focused on tracing the side effects of a carried out change in software components. Traditionally, development engineers use history of software artifacts to analyze their present state and predict future changes[5] complemented by existing reverse engineering approaches [6, 11, 2]. The extraction of historical data on explicit user requests may not be sufficient to have higher quality information for software evolution [10] and instead creates the difficult situations to trace the change impact propagation between modeled software components.

As an a priori change impact analysis requires an identification of ripple effect which can be provoked by a modification [1, 9], we show how the integrated proposed description of software components can help to assure a more precise analysis of ripple effect.

In this work, we classify software components on different levels in software development and maintenance phases (section II). Then, in section III we provide an exhaustive classification of component relationships. Later, we discuss the potential of any relationship between components to propagate change impact (section IV), and we propose a typology of this impact (section V). We demonstrate then how the proposed approach can be used, starting from the directly changed component, to identify the path set of impact propagation (section VI).

II. CLASSIFICATION OF SOFTWARE COMPONENTS

A component is a constituent of software application, having its well defined boundaries and interfaces which allow it to communicate with other components. Software evolution is not generally targeted to only one component, instead it is implied on a precise set of components. Similarly, a change on an individual component may affect the linked components[4]. Change impact propagation can be traced by individual analysis of components on different levels[3]. Therefore, it is important to identify the set of affected components. We classify the set of software components within an individual phase on different levels of abstraction or granularity.

The components having the same abstraction and granularity level can further be distinguished by their context pertinent to the change propagation and qualitative evaluation. It is important to note that two software components can have the same abstraction level, without having same granularity level. In the object oriented paradigm, fields and methods are on the same level of abstraction but are on different levels of granularity. In the same way two software components can have the same granularity level without having the same abstraction level. A generic class indeed can be on the same level of granularity of a very short class, but they belong to different levels of abstraction. We treat separately, the components coded in procedural languages and the
components coded in object oriented languages, though they also have common elements.

The primary classification based on granularity is further refined in respect of the language semantic. In procedural languages, identified components levels are generic modular, effective modular, block, instruction, expression, and individual symbol level fragments. In addition to these, object-oriented languages also identify free functions, objects, methods, and fields levels. Common elements in both procedural and object-oriented languages are further identified on different granular sub-levels. In the modularity level the distinguished sub-levels are specification of the component and body of the component. On block level sub-levels are declarative blocks, control blocks, functional blocks, and commentary blocks. Instruction level fragments distinguish declarative instructions, control instructions, basic functional instructions, and commentary lines. An individual symbol level gives sub-levels relative to operands, operators, and the symbols which are neither operands nor operators.

In general, we can denote the set of software components by $\Sigma C$ as:

$$\Sigma C = \Sigma mod \cup \Sigma obj \cup \Sigma mtd \cup \Sigma blk \cup \Sigma sym$$

where $\Sigma mod, \Sigma obj, \Sigma mtd, \Sigma blk$ and $\Sigma sym$ are sets of modules, objects, methods, blocks, and individual symbols respectively.

To better control the software evolution, component change history can be maintained. The set of reasons and corresponding managerial and/or technical decisions $\Sigma R$ are recorded on the affected set of components $\Sigma AC$. The set of relationship types between each phase and pair $<\Sigma R, \Sigma AC>$ describes the potential components to be affected along with the reasons of change and corresponding decisions.

III. CLASSIFICATION OF COMPONENT RELATIONSHIP

Software components are linked to each other by some specific relation types. A relationship type is a basic entity that provides a path for the change impact flow. We analyze a relationship type as one of the below given three categories (Fig. 1):

- Inter-phase relationships
- Horizontal relationships
- Vertical relationships

Inter-phase relationships represent the traceability links between components issued from two different software development phases. As discussed earlier, inside each phase we have different levels. In case of a change, we analyze the horizontal (on same level) and vertical (on different levels) relationships among these levels, progressively in the same and different phases. This progression on inter-phase relationships provides the aspects of forward and backward relationships.

We denote the set of all the relationships in a particular phase ($\Phi_i$) as $\Phi_i \Sigma rel$:

$$\Phi_i \Sigma rel = \Phi_i \Sigma ir U \Phi_i \Sigma hr U \Phi_i \Sigma vr$$

where $\Phi_i \Sigma ir$, $\Phi_i \Sigma hr$, and $\Phi_i \Sigma vr$ are sets of inter-phase, horizontal, and vertical relationships in $\Phi_i$.

As inter-phase relationship can be forward or backward inter-phase relationships, so:

$$\Sigma ir = \Sigma irf U \Sigma irb$$

where $\Sigma irf, \Sigma irb$ are forward inter-phase relationships and backward inter-phase relationships.

Therefore,

$$\Phi_i \Sigma rel = \Phi_i \Sigma irf U \Phi_i \Sigma irb U \Phi_i \Sigma hr U \Phi_i \Sigma vr$$

IV. QUALIFICATION OF CHANGE IMPACT CONDUCTANCE

The potential of a relationship to propagate the change impact from one of the participating components to the other linked components qualifies its change impact conductivity. The prior prediction of change impact flow is critical and requires the knowledge of change impact conductivity of a relationship type under certain circumstances. This highly depends on the type of relationship and its cardinalities. By cardinalities we mean the conditions and the number of times a relationship is invoked.

Considering a relationship between two components $C_i$ and $C_j$ from a phase $\Phi_i$, such that $rel(C_i, C_j) \in \Phi_i \Sigma rel$. The type of relationship $rel(C_i, C_j)$ can have zero or multiple distinct occurrences. In the same way, each occurrence can be iterative or conditional. The change impact analysis wouldn’t be effective until all the relevant cardinalities of the relationships are considered. Consequently the properties characterizing each occurrence of relationship $rel(C_i, C_j)$ are identified as $Attr(rel(C_i, C_j))$, where the individual value of each property of corresponding occurrence of relationship is accounted.

![Figure 1. Inter-phase, inter-level, and intra-level relationships](image-url)
Typology of Change Impact Propagation

Several dependency aspects of a relationship are considered to assess the degree of importance during a change implementation. A dependency aspect among components may imply the change in one of the participating components in a relationship may concern the changes in others. The typology of change impact propagation may include functional, qualitative, structural, logical and behavioral aspects and these reflects the impact effect. Change impact can be considered in these dimensions to understand the consequences.

A. Structural change impact flow

Structural change impact flow implies that any structural modification in a software component can cause a similar structural change in dependent components. The structural change can be invoked as; merging of two software components of same level into one component, splitting of one software component into two components of same level, addition of software components of different levels in a software component, deletion of software components of different levels from a software component. The structural difference between states of a component \( C_x \), before and after the change identifies the carried out structural change. This can be represented as:

\[
C_{x+} = C_x + \Delta_S
\]

Any structural change in a component may reflects a corresponding change in related components. Structural merging of two components \( C_x \) and \( C_y \) to \( C_z \) can propagate the change impact to the interacting components with \( C_x \) and \( C_y \). The related components of \( C_x \) and \( C_y \) can be directly affected by this change, and they will be sensitive to any changes on \( C_x \) later on. Same is the case of structural splitting of a component \( C_x \) to \( C_x \) and \( C_y \).

B. Qualitative change impact flow

A qualitative change impact flow represents that any qualitative improvement or decline in a component can be concerned by related components in qualitative aspects. We can elaborate qualitative change (\( \Delta_Q \)) by considering a component \( C_x \). We apply some qualitative metrics (M) to measure certain quality values (VM) of \( C_x \). The \( \Delta_Q \) implies corresponding newer values (VM+) as a result of same qualitative metrics of \( C_{x+} \), after the change. This can be represented as:

\[
C_{x+} = C_x + \Delta_Q
\]

\[
M_1, M_2, M_3, \ldots M_n(C_x) = VM_1, VM_2, VM_3, \ldots VM_n
\]

\[
\Rightarrow M_1, M_2, M_3, \ldots M_n(C_x) = VM_{+1}, VM_{+2}, VM_{+3}, \ldots VM_{+n}
\]

Therefore, any of the related component with \( C_x \) will be concerned accordingly.

C. Functional change impact flow

A functional change impact flow corresponds to any functional change in a component that may directly propagate its impact to other related components. For instance consider two components \( C_x \) and \( C_y \), such that \( C_x \) is defined as interface to \( C_y \). The relations rel\( (C_x, C_y) \) implies that \( C_x \) has the descriptions of required and provided data along with the list of data and the type of data. While the \( C_y \) has the descriptions of required and provided services along with the list of services and the type of services. A functional change (\( \Delta_F \)) either in data or services and/or their types in \( C_y \) will directly affect the other. The functional description of a component includes its architectural specifications. Any addition or deletion of functionality in software components can propagate functional change impact accordingly.

D. Logical change impact flow

A logical change impact flow is mostly intra-component. It reflects the change of any condition or logic in use of expression as selection or iteration. Logical change (\( \Delta_L \)) is based on the representation of control flow graph and implies that any change in decisional node can propagate the change impact flow accordingly.

E. Behavioral change impact flow

Collectively, behavioral change in components implies the change in program behavior. A behavioral change impact flow represents that any change in behavior of a component can cause a potential change in the behavior of the related components. For instance, consider a component \( C_x \) representing a program state. In case of certain decision \( C_x \) signals the new state \( C_{x+} \). A behavioral change (\( \Delta_B \)) in \( C_x \) can affect the program behavior to a new state.

VI. Visualizing Change Impact Flow

Visualization may reduce the complexity [8] as it makes use of various forms of imagery to provide insight and understanding. For program analysis, graph models can be used as either simple graph or a combination of different graphs. One such graph model is given by TGrafts [7]. These have total ordering of the node set, the edge set, the set of edges incident to a node. These may have attributed and typed nodes and directed edges.

We have set of all the components (\( \sum C \)) represented as nodes and set of all the relationships (\( \sum \text{rel} \)) represented as edges, so a graph can be shown as an ordered pair of components and relationships:

\[
G = \langle \sum C, \sum \text{rel} \rangle
\]

Two adjacent components are related components through some relation. The directed edge makes one of the senses forward and the other backward. For the sake of increased comprehension, types are assigned individual schema specify-
In the identification of other components, and so on the process is continued. Components that may be affected by change on this dependent component are determined. If a change affects a component, then the impact is calculated. It includes the higher association in the software. The objective is to trace the complexity, by visualizing only specific sub-graphs. Each edge is labeled with the name of the edge and the corresponding component level along with its name. In the relation \( \text{rel}(C_x, C_y) \), a component \( C_x \) is directed to a component \( C_y \), representing \( C_x = \text{tail}(\text{rel}(C_w, C_y)) \) and \( C_y = \text{head}(\text{rel}(C_w, C_x)) \).

If two components \( C_x \) and \( C_y \) are linked by a relation \( \text{rel}(C_x, C_y) \), then \( \text{rel}(C_x, C_y) \) is change impact conductive in the sense that \( C_y \) is concerned to any modification \( \Delta(C_x) \) on \( C_x \). More interaction can be achieved by showing change propagation on concerned node(\( C_x \)). We visualize the same for component \( C_x \) connected to component \( C_y \). The relation \( \text{rel}(C_y, C_x) \) is change impact conductive when \( C_y \) is concerned to any modification \( \Delta(C_y) \) on \( C_y \).

The user control on visibility of components may decrease the complexity, by visualizing only specific sub-graphs. The higher degree of any particular node(component) represents its higher association in the software. The objective is to trace the change impact on a particular component. It includes the identification of the component to be modified and the identification of components which directly or indirectly depend on the selected component. If a change affects a component, then the identification of other components that may be affected by change on this dependent component, and so on the process is continued.

As example, consider the C++ program shown in Table I and Table II. The sample code has set of components \( \Sigma C \) and set of various relationships \( \Sigma \text{rel} \) types. Any change on one of the components may affect the other components. Change impact flow of a change \( \Delta(C_x) \) on component \( C_x \) is illustrated in Fig. 2.

The first part demonstrates a code snippet with following components and relationships:
- \( C_x, C_y \in <L_{\text{vis}} \Sigma C> \)
- \( C_y \in <L_{\text{mod}} \Sigma C> \wedge C_y \in C_r \)
- \( C_x, C_y \in <L_{\text{sym}} \Sigma C> \wedge C_y \in C_r \wedge C_x \in <L_{\text{mod}} \Sigma C> \wedge a, b \in <L_{\text{sym}} \Sigma C> \wedge a, b \in C_r \)

Some of the relationships from \( \Sigma \text{rel} \) are as follows:
- \( \text{function}(C_x, C_y) \in \Sigma \text{rel} \)
- \( \text{friend}(C_x, C_y) \in \Sigma \text{rel} \)
- \( \text{param}(C_x, C_y) \in \Sigma \text{rel} \)

| TABLE I. C++ code components (Part 1) |
|---|---|
| File Cx.h | ==ctype== |
| //prototypes | 
| class Cx; | class Cw; |
| class Cx { | public: |
| private: | double Cp(Cx& x); |
| }; | class Cw { |
| | public: |
| | double Cw::Cp(Cx& x); |
| | } |
| } | double Cw::Cp(Cx& x) { |
| | return x.Ca * x.Cb; |
| } | double Cw::Cp(Cx& x) { |
| | return x.Ca * x.Cb; |
| | } |

| File Cy.cpp | ==ctype== |
| #include<Cx.h> | 
| //prototypes | void Cv(double); |
| double Cs(); | void Cv(double Cb) { |
| double Cu(); | double Ca; |
| } | //calling |
| } | //modification |
| } | Cz=Ca * Cb; |
| } | double Cs() { |
| | double Cq; |
| | //declares |
| | Cw Ct; |
| | /*invokes member function, effective parameter*/ |
| | Cq = Ct.Cp(Cy); |
| | return Cq; |
| | } |
| } | double Cu() { |
| | double Cr; |
| | //use |
| | Cr = Cz * 1.16/5; |
| | return Cr>Cz ? Cz : Cr; |
| | } |

As an example, consider the C++ program shown in Table I and Table II. The sample code has set of components \( \Sigma C \) and set of various relationships \( \Sigma \text{rel} \) types. Any change on one of the components may affect the other components. Change impact flow of a change \( \Delta(C_x) \) on component \( C_x \) is illustrated in Fig. 2.
We identify some of the prominent relationships from $\Sigma_{rel}$ as follows:

- $\text{instance}(C_v, C_i) \in \Sigma_{rel}$ if $\Delta(C_v)$ then $\Delta(C_i)$.
- $\text{call}(C_v, C_i) \in \Sigma_{rel}$ if $\Delta(C_v)$ then $\Delta(C_i)$.
- $\text{modify}(C_v, C_i) . \text{use}(C_w, C_i) \in \Sigma_{rel}$ if $\Delta(C_v)$ then $\Delta(C_i)$.
- $\text{include}(C_v, C_i) . \text{include}(C_w, C_i) \in \Sigma_{rel}$ if $\Delta(C_v) \vee \Delta(C_w)$ then $\Delta(C_i)$.
- $\text{invoke}(C_v, C_i) . \text{param}(C_v, C_i) \in \Sigma_{rel}$ if $\Delta(C_v)$ then $\Delta(C_i)$.
- $\text{data}(C_v, C_i) \in \Sigma_{rel}$ if $\Delta(C_v)$ then $\Delta(C_i)$.

As discussed earlier in section V, the change impact analysis greatly depends on the nature of change and its propagation can be expected in several aspects. We can individually analyze the change impact flow in each of these aspects (Fig. 3).

VII. CONCLUSION AND FUTURE WORK

It is essential for software to evolve by its changing requirements and technology. We presented exhaustive classification for both software components and their interdependent relationships for better understanding of change impact propagation. It provides an insight into several aspects of software structure, quality, functionality, logic and behavior. It gives abstraction of functionality on multiple levels in software development and maintenance phases. This classification covers all types of relationship conductive for change impact and may occur between software components.

The main objective is to determine the change impact conductance between the affected components by an applied change. The generated model is designed to permit, according to user request, to generate specific views of interactive graphs. It enables to identify the elements of software that may be affected by incorporating the change.

The implementation of our approach is continuing to construct a knowledge base system for storage, representation and processing of software specific information. It can be used to generate interactive graphs to visualize and trace the impact of modification among the different concerned software components. In the future, apart from one language, the prototype of validation will accept the components of applications developed in multiple programming languages.

ACKNOWLEDGMENT

We would like to thank Dr. Laurent Deruelle for providing supporting discussions, valued inputs, and comments. The research of this contribution is funded by Higher Education Commission of Pakistan and the Laboratoire d’Informatique du Littoral, Calais, France.

REFERENCES


