Towards a better control of Change Impact Propagation

Adeel Ahmad
Laboratoire d'Informatique du Littoral
Maison de la recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
adeel.ahmad@mel-etu.univ-littoral.fr

Henri Basson
Laboratoire d'Informatique du Littoral
Maison de la recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
basson@lil.univ-littoral.fr

Laurent Deruelle
Laboratoire d'Informatique du Littoral
Maison de la recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
deruelle@lil.univ-littoral.fr

Mourad Bouneffa
Laboratoire d'Informatique du Littoral
Maison de la recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228, Calais Cedex, France
bouneffa@lil.univ-littoral.fr

Abstract — The software evolution requires an exhaustive understanding of all the artifacts or components composing the software. In fact, any intended change may generate an impact propagated to the various software components through the different kind of relationships relating them. This situation is generally called the ripple effect and may cause several types of damages or side effects. In this paper, we focus on the potential flow of change impact propagation as provoked by the carried out change. For that purpose, we propose a Structural Model of Software Evolution (SMSE). This model is intended to represent and manage the different software artifacts and their relationships. The instantiated SMSE is used to build a knowledge base system providing, among other facilities, an assistance to the designer (or engineer) of software evolution for an a priori assessment of the change impact. The major aim of the system is to help a systematic analysis of the change impact and the identification of its propagation.

Keywords — software evolution; change impact analysis; modeling; change impact visualisation;

I. INTRODUCTION

The software evolution is unavoidable and its control is necessary to minimize unexpected difficult situations resulting from applied software changes. The change is generally time and effort consuming mainly because of the difficulty in understanding of various software artifacts which could be concerned by the applied software changes. It is also error or fault prone. Some artifacts are related to source codes which may contain business logic and technical implementation details realized without associated descriptive documents or explaining comments. Constantly evolving software development environments provide implementation expressive facilities which are not always helping against the tendency of increasing complexity of large systems. The Unified Modeling Language [14] describes the software systems at design level without foreseeing the construction of mapping structure describing links between concerned elements of its various models. That does not facilitate tracing the change impact between modeled software components. There is a large set of tools for assistance in reverse engineering, refactoring, or re-engineering [12, 3, 2, 10, 15, 9, 8] but they provide very limited knowledge describing the impact flow of applied changes. We think that the requirement for analyzing a change impact on software components can be met through adapted usable models. These should be based on an exhaustive and stratified description of software components [1] permitting then to provide interactive views of detailed information describing aspects of existing software applications, considered by experts as pertinent in regard to the intended change. On a requested change, an a priori impact analysis greatly helps in the change related decision making [4].

In the following sections, we first present and discuss the model SMSE and its elaboration (section II). We show a classification of the software components (section III) and the various relationship types (section IV) considered and modeled by SMSE. We discuss the graphical representation of change impact analysis (section V) and change propagation process (section VI). Later, we show the propagation tool (section VII) based on a knowledge based system in which the facts are the instantiation of the different SMSE constructs.

II. MODELS FOR SOFTWARE EVOLUTION CONTROL

Models are cost-effective representation of reality for some cognitive purpose; they may allow to avoid complexity and irreversibility in a simplified and safer manner [7]. Representing the system aspects through various models is a widely applied approach. These models can be vulnerable towards incorporating changes or allowing change impact analysis. That makes it necessary an integrated visualization of both the software components and the propagation of the
change impact affecting such components [11]. Several models are destined to software development [13] but they do not allow means of facilities for the change impact analysis. Minor changes in a component can result in unpredictable future modifications in the application behavior. For instance, the slightest change on an individual variable can cause dysfunction in the whole application. In the opposite side, removing hundreds of lines of code may, in some cases, have no significant impact on the present application behavior. Subsequently, a change in any software component must be addressed and treated properly by analyzing its impact. This cannot be achieved without having an exploitable knowledge concerning the software artifacts (from the more abstract to the more detailed ones) and the different kind of relationships linking them. That is the first objective of the SMSE model.

A. The Structural Model of Software Evolution

Coping with the software change involves understanding the multiple aspects of software components considered from architectural, structural, functional, behavioral, or qualitative, points of view. We present the Structural Model of Software Evolution intended to trace the change impact propagation among interrelated software components. In this paper we deal, especially, with the structural aspects. SMSE considers the set of phases which are shared among various software process models. This set noted \( \sum \Phi \) is defined as bellow:

\[
\sum \Phi = \{ \Phi_i: i=1 ... n \}
\]

where ‘\( \Phi_i \)’ represents the ‘\( i \)th’ individual phase, and ‘\( n \)’ is the total number of phases. \( \sum \Phi \) is such as:

\[
\sum \Phi = \Phi_a U \Phi_d U \Phi_c U \Phi_t U \Phi_m
\]

where \( \Phi_a, \Phi_d, \Phi_c, \Phi_t, \) and \( \Phi_m \) are analysis, design, coding, testing, and maintenance phase respectively.

The sub-phases in a phase \( \Phi_p \) for instance, can be constituted as \( \Phi_p1, \Phi_p2, \) ... \( \Phi_pn \), e.g., in the specific case of the design \( \Phi_d \), sub-phases can be \( \Phi_d1, \Phi_d2, \Phi_d3 \), where \( \Phi_d1 \) corresponds to the design specification, \( \Phi_d2 \) represents primary design and \( \Phi_d3 \) represents the detailed design. Hence we can show:

\[
\Phi_d = \Phi_d1 U \Phi_d2 U \Phi_d3
\]

Software development is generally applied following one of the models (Waterfall, V, Spiral, Incremental, Evolutionary, Transformational, Agile, Rapid Prototyping etc.), and it is carried out by a group of Actors realizing a set of Activities. To each actor is attributed a defined role, according which he or she will accomplish a well defined subset of \( \sum Aty \) (the set of activities) across the software development and maintenance. The Activities are composed of software processes, which can be local to an individual phase or global across the whole set of software life phases as project management, change management, quality assurance and configuration management. A particular method from the set of methods \( \sum Mtd \) is often applied by a software process to accomplish component description, achievement, or testing, etc. We denote the used languages by \( \sum Lng \), and all the tools which can be used by \( \sum To \). To the set of phases \( \sum \Phi \) of the whole project corresponds the \( \Delta \Phi \) the high description of phases as being specified by \( \sum Act, \sum Aty, \sum C, \sum Mtd, \sum Lng, \sum To \).

B. Modeling the software components

In our previous study [1], we have shown that, the set of software components developed within an individual phase belong to different levels of abstraction or granularity. Considering specifically the coding phase, a set of granular components can be modules, objects, methods, blocks, or symbols. We denote the set of software components by \( \sum C \).

\[
\sum C = \sum mod U \sum obj U \sum mtd U \sum blk U \sum sym
\]

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

A component or a set of components can be analyzed at modular, object, method, block, and individual symbol level [5]. The set of components can be represented on their corresponding level as a pair \( \sum L, \sum C \) where \( \sum L \) is the set of all the levels. Thence, \( k \)th component of \( j \)th level is denoted as \( \Phi_{ij} \). Although the set of components of the same individual \( i \)th level of an individual phase ‘\( \Phi_i \)’ is denoted by \( \Phi_i \).

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

C. Modeling transformation between phases

We consider the link between two consecutive phases as \( \Phi_i \) and \( \Phi_j \), \( (j = i+1) \) and \( i = 1 ... n-1 \), where ‘\( n \)’ is the total number of phases. Hence \( \Phi_i \) is on higher level of abstraction then \( \Phi_j \), all the components of \( \Phi_i \) necessarily describe the components of \( \Phi_j \). The transformation from \( \Phi_i \) to \( \Phi_j \) can be represented by a transformational function \( Tij \) destined to transform the components of phase \( \Phi_i \) to the components of phase \( \Phi_j \). The function \( Tij \) is represented as:

\[
Tij : \delta \sum \Phi_i \sum C \rightarrow \delta \sum \Phi_j \sum C
\]

The transformation function \( Tij \) translates the component \( Cx \) of phase \( \Phi_i \) (\( Cx \in \Phi_i \)) to another component \( Cy \) of phase \( \Phi_j \) (\( Cy \in \Phi_j \)). If \( j > i \), then we say \( \Phi_i \rightarrow \Phi_j \) is a concrete description of \( \Phi_i \rightarrow \Phi_j \) by \( Tij \) i.e. \( \Phi_i \rightarrow \Phi_j \) is concrete description of \( \Phi_i \rightarrow \Phi_j \) by \( Tij \) i.e. \( \Phi_i \rightarrow \Phi_j \) is abstract representation of \( \Phi_i \rightarrow \Phi_j \) by \( Tij \) i.e. \( \Phi_i \rightarrow \Phi_j \) is abstract representation of \( \Phi_i \rightarrow \Phi_j \) by \( Tij \).
The elements involved in software development and maintenance on all phases are interdependent and inter-related. If these elements are viewed in form of graphs then the graph of a particular phase can be analyzed during its transformation to the graph of the next phase. The impact flow graph of change in the previous phase can be traced and reflected in the next phase. As discussed earlier each phase can have several interacting levels, and each level has several interacting components. The components involved in each phase are significant, a change in a component affects the related components [6]. The impact flow of change can be traced between components considering the intra-level, inter-level, and inter-phase relationships. We restrain the focus of this paper to the software components related to coding phase.

III. CLASSIFICATION OF SOFTWARE COMPONENTS

We classify different sets of components for the simplification. Through their individual analysis, two relevant criteria can distinguish the components in a phase. The abstraction and the granularity levels. 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 software components $C_x$ and $C_y$ can have the same abstraction level ($La$), without having same granularity level ($Lg$). 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 $C_x$ and $C_y$ 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.

In this paper, we narrow our scope to level of granularity without going into the details of level of abstraction. 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.

IV. CLASSIFICATION OF COMPONENT RELATIONSHIPS

Modeling relationships is both complex and important. We categorize three types of relationship:

a) Inter-phase relationship

b) Horizontal relationships

c) 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 of abstraction. In case of a change we denoted it by $\Delta$, we analyze the horizontal (the same level) and vertical relationships (different levels) among these levels, progressively in the same and different phases. This progression on inter-phase relationships provides the aspects of forward and backward relationships. A relationship between two components $C_x$ and $C_y$ can be shown as $rel(C_x, C_y)$. We denote all the available set of relationships as $\Sigma rel$:

$$\Sigma rel = \Sigma ir \cup \Sigma hr \cup \Sigma vr$$

where $\Sigma ir$, $\Sigma hr$, $\Sigma vr$ are set of inter-phase, horizontal, and vertical relationships respectively.

We denote the set of all the relationships in a particular phase $\Phi$, as $\Phi \_\Sigma rel$:

$$\Phi \_\Sigma rel = \Phi \_\Sigma ir \cup \Phi \_\Sigma hr \cup \Phi \_\Sigma vr$$

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

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

$$\Sigma ir = \Sigma irf \cup \Sigma irb$$

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

Therefore,

$$\Phi \_\Sigma rel = \Phi \_\Sigma irf \cup \Phi \_\Sigma irb \cup \Phi \_\Sigma hr \cup \Phi \_\Sigma vr$$

Figure 1. Change impact conductance design or implementation relationship
A. Modeling inter-phase relationship

The set of inter-phase relationships is based on the communication between software development phases. The set \( \Phi_c \sum [rb] \) includes the direct relationships in backward correspondence between a component \( C_y \in \Phi_c \sum C \) and its abstract image \( C_z \in \Phi_{c+1} \sum C \) where \( \Phi_i \) represents next consecutive phase of \( \Phi_{i+1} \). In the same way, the set \( \Phi_c \sum [if] \) includes the forward relationships between a component \( C_x \in \Phi_c \sum C \) and its implementation (or direct refinement) by a component \( C_y \in \Phi_{i+1} \sum C \).

The relationships concerned with conductivity of impact of change in a component \( C_x \in \Phi_{i-1} \sum C \) may cause a breach of consistency between \( C_y \) and its possible description \( C_z \in \Phi_{i+1} \sum C \) and between \( C_y \) and its possible implementation \( C_x \in \Phi_{i-1} \sum C \).

For example, we consider two components \( C_x \) (UML class) and \( C_y \) (C++ class) on two distinct phases; design phase \( \Phi_d \) and coding phase \( \Phi_c \) of software development cycle, respectively as illustrated in figure 1. The relation \( rel(C_y, C_x) \in \Phi_d \sum [hr] \) implies that \( C_y \) is the implementation of \( C_x \) and the relation \( rel(C_y, C_z) \in \Phi_c \sum [rb] \) implies that \( C_z \) is the abstract design of \( C_x \). The forward inter-phase relationship \( rel(C_x, C_y) \) is change impact conductive to \( C_y \) by any modification \( \Delta(C_y) \) on \( C_y \). The backward inter-phase relation \( rel(C_x, C_z) \) is change impact conductive to \( C_z \) by any modification \( \Delta(C_z) \) on \( C_z \). Therefore, inter-phase relationship \( rel(C_x, C_y) \) is conductive of bidirectional change impact.

B. Modeling horizontal (intra-level) relationship

The horizontal relationships are among the components of the same level of a phase. The set \( \Phi_c \sum hr \) represents the set of all horizontal relationships in a phase ‘i’ and may be classified separately on the formalism of programming language. The existing relationships presented here, are among effective (non-generic) modules of source code implemented using a procedural language or an object oriented language. The set of horizontal relationships contains the call, importation, exportation, protection, inclusion, inheritance, instantiation, friendship, communication, synchronization, renaming, and overloading relationships.

For illustration, we consider two components \( C_x \) and \( C_y \) as a couple of modules in coding phase \( (C_x, C_y) \in \Phi_c \sum \mod[C++] \) such that \( C_x \) calls \( C_y \) (figure 2). The calling relation \( rel(C_x, C_y) \in \Phi_c \sum [hr] \) implies that any modification \( \Delta(C_x) \) to the \( C_x \) (called component) affects the \( C_y \) (calling component), but the modifications in \( C_y \) have no impact on \( C_x \). The extent to which \( C_x \) is concerned with the change in \( C_y \) depends obviously on the nature and the type of the change.

C. Modeling vertical (inter-level) relationship

The set of vertical relationships is based on the existing links among components appearing on different levels. The set \( \Phi_c \sum [vr] \) represents set of all vertical relationships in a phase ‘i’. If \( C_x \) denotes a component of individual symbols at symbol level \( (C_x \in \Phi_{sym} \sum [sym]) \) and \( C_y \) denotes a modular component on modular level \( (C_y \in \Phi_{mod} \sum [mod]) \) then the vertical relationship between these can be represented as: \( C_x \) can be declared, initialized, read, calculated, modified, protected, exported, imported, visible, effective parameter in a call, or displayed by \( C_y \).

We consider, for illustration, three components \( C_x \), \( C_y \), and \( C_z \) in coding phase \( \Phi_c \) of software development cycle. \( C_y \) is a couple of modules \( (C_x, C_y) \) \in \Phi_c \sum [mod[C++]] \) and \( C_y \) is a component of individual symbols \( (C_x \in \Phi_{sym} \sum [sym]) \). \( C_z \) is visible to both \( C_x \) and \( C_y \). The vertical relation \( rel(C_y, C_x) \in \Phi_c \sum [vr] \) implies that \( C_y \) is able to modify the \( C_x \). The vertical relation \( rel(C_y, C_z) \in \Phi_c \sum [vr] \) implies that \( C_y \) is able to use \( C_z \). Any modification \( \Delta(C_y) \) on \( C_y \) may affect the \( C_z \) (figure 3).

Hereafter, we explain how we implement the change propagation process, using a Knowledge Base System.

V. GRAPH BASED ASPECTS OF SOFTWARE REPRESENTATION

Software understanding is crucial in order to successfully incorporate changes. The major barrier in comprehension can be large number of lines of code and even greater number of logical paths during their execution.
Visualization may reduce the complexity [18] as it makes use of various forms of imagery to provide insight and understanding. Graph is a mathematical structure represented as set of nodes and edges, such that each edge connects nodes [17]. These can be used for efficient representation of binary relations among components.

For mastering the complexity we apply both the strategies, divide-and-conquer and abstraction, to generate interactive graphical form of components. Graphical representation of components may greatly ease the task of analyzing the change propagation impact on inter-phase, inter-level and intra-level views. On request analysis of selected components at different levels may resolve the problems of large sized code. We consider the set of relationships \( \sum \text{rel} \) as edges and set of components \( \sum \text{C} \) as nodes. The created graphs are meant to assist in change impact analysis and to outline the relationships among components.

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 TGraphs [16]. 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 \( \sum \text{C} \) 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:

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

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 specifying the attributes of nodes and edges. Each edge is labeled with the name of relationship and node with the corresponding component level along with its name. In the relation \( \text{rel(C}_x, \text{C}_y) \), a component \( \text{C}_x \) is directed to a component \( \text{C}_y \) representing \( \text{C}_x = \text{tail(} \text{rel(C}_x, \text{C}_y) \text{)} \) and \( \text{C}_y = \text{head(} \text{rel(C}_x, \text{C}_y) \text{)} \).

VI. THE CHANGE PROPAGATION PROCESS

The change propagation process refers to the process of actually carrying out a set of initial modifications to the software components, and to re-establish the software consistency, by making a set of estimated consequent changes. This process would involve advising the user the software components to be changed and the types of the changes.

The facts of the Expert System are software components and their marking for change. The rules of the Expert System are the change propagation rules. The idea of our approach is to insert components to be changed and their marked neighbors in the knowledge base (KB). In the change propagation algorithm, FireRules designates the set of propagation rules that can be fired:

1. Given a consistent program represented by 
   \[ G=(V,E); \]
2. Select \( a \in V; \)
3. Change(ModificationType, a);
4. insert fact (ModificationType, a);
5. \( G'= (V',E'); \)
6. do
   7. do
      8. Select a rule \( r \) in FireRules;
      9. Trigger \( r\text{.actionSet}; \)
     10. }
    11. while (FireRules \( \neq \emptyset \));
12. Select a \( \in \text{mark}(G); \)
13. Change(ModificationType, a);
14. insert fact (ModificationType, a);
15. \( G'= (V',E'); \)
16. while (\( \text{mark}(G) \neq \emptyset \));

The change propagation algorithm is applied on a software components graph \( G=<V,E,\text{Constr}> \) defined as follow:

- the set of nodes or vertices \( V \) representing the software components
- the set of edges corresponding to the relationships between the software components.

When a change is performed on a software component \( a \), the change propagation algorithm compute the modified program graph \( G' \) by using the corresponding morphism of
ModificationType. Then change propagation rules are inferred to perform actionSet, consisting to mark a node as affected by the change and to propagate to other linked nodes.

Our Change Propagation Process is implemented by the way of plugins technology in the eclipse Platform. In the next section, we show the platform architecture, which implements both SMSE model and Change Propagation Process.

VII. ECLIPSE PLUGIN IMPLEMENTATION

We develop a set of ECLIPSE plugging that are currently used by our team to analyze change impact propagation affecting a java project developed on top of ECLIPSE.

As it is shown in figure 4, our platform is based on four major plugins: the Eclipse Multi-Languages Parser plugin, the Eclipse Software Modeler plugin to represent software structural information, the Eclipse Graph Visualizer plugin, and the Eclipse Change Propagation Engine plugin.

The first element is a plugin to extend eclipse platform to generate parsers using javacc to analyze the multi-language (C, C++, java, Cobol, MySQL, PHP, SQL, and so on) source code of any input software.

It provides an extension point used by each programming language managed plugin. It's easy to add new language in our platform by extending this extension point and to provide a javacc grammar for the language. The input can also be a byte-code. In that case, a decompiling step is necessary, that is why our platform includes a byte-code decompiler (mocha). From the input or resulting source code, the plugins generates a set of XML-based representations that describe the components and the relationships between them, and represent an instance of SMSE for program source code and database schema files. SMSE is implemented as plugins that extend the Eclipse Software Modeler Plugin. This provides basic services for graph creation and transformation based on the XML-based representation. A graph contains nodes and links representing software components and relationships defined in the implemented models plugins. The Eclipse Graph Visualizer Plugin shows the resulting graph and allows to perform basic changes on the graph. The change propagation engine is based on an Engine Rules (JBOSS Rules) and propagates a change performed on a component to the belonged components. The propagation is based on change propagation rules, which are fired by inserting facts, provided by the facts builder. The facts builder inserts facts, that represents the software components and the relationships stored in the repository. A change propagation rule is a Production Rule which is a two-part structure using First Order Logic for knowledge representation. The process of matching the new or existing facts against Production Rules is based on the Rete algorithm. The Eclipse Graph Visualizer Plugin shows the change propagation results as shown in figure 5.

VIII. CONCLUSION AND FUTURE WORK

It is essential for software to evolve by its changing requirements and technology. We presented Structural Model of Software Evolution for better understanding of change impact propagation and discussed its instantiation on coding phase. SMSE provides an insight into software structure and its quality. It gives abstraction of functionality on multiple levels in software development and maintenance phases. It
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 used to develop a knowledge base system, 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.

SMSE implementation is continuing 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

The research of this contribution is funded by Higher Education Commission of Pakistan and the Laboratoire d’Informatique du Littoral, Calais, France.

REFERENCES


