Measuring Change Complexity from Requirements: A Proposed Methodology

Sarah Maadawy
Computer Science Department
FCI, Cairo University
Cairo, Egypt
s.maadawy@fci-cu.edu.eg

Akram Salah
Computer Science Department
FCI, Cairo University
Cairo, Egypt
akram.salah@fci-cu.edu.eg

Abstract:

Complexity in software industry is an important issue for software houses as it affects planning and cost estimate of the project leading to productivity and quality drawbacks. Measuring complexity is important in the beginning as well as when changes occur in the system. It is very difficult to produce a system that does not need any future changes after being initially released. The relative cost for maintaining software and managing its evolution is continuously increasing and now it represents more than 90% of the total effort of the software [12].

Numerous measurement methods have been proposed to measure software complexity. These methods are designed mainly to serve for a new project complexity. It measure impact of change as another requirement in the project and not considering the fact that a change has a different nature. This paper presents a methodology to measure software complexity for changes. It studies attributes that affect complexity of change and the relation between requirements and each other to finally find a complexity measure the will serve in finding a precise estimate for the change.

Keywords: Requirement Engineering, Requirement Complexity, Complexity Attributes, Change Impact Analysis, Software Maintenance

I. INTRODUCTION

Requirements change is complex to manage not only because it means more time to be spent on implementing the new features, but also that such change may have direct or indirect impact on other requirements that need to be measured. This impact is sometimes hard to measure. To be able to effectively measure complexity of change, changes should be related to initial requirements by traceable links representing dependencies between the introduced changes, system requirements and other artifacts of the development lifecycle. Software complexity directly affects an estimation needed to build this software.

Software engineers face great challenges for measuring software complexity of a software product. In [1], software complexity measurement is classified to three categories: (1) Size measurement, (2) Structure measurement, and (3) Quality measurement. Size measurement (Code based) is often assessed by measures of length or functionality. The most frequently used code-based measures are McCabe metric, Halstead and LOC (Lines of Code). The problem of code based complexity measure is that it cannot be applied at early stages in
the software development process but rather after implementing it. Structure measurement (Cognitive based measurement) is to analyze the software structure by means of control flows, data flows, nesting, control structures and modular structure and interaction [16]. Quality measurement requires qualitative rather than quantitative forms. This is a tricky kind of measurement and it is out of this research scope.

Function Point Analysis [5] is a structured technique of problem solving. It is a method to break systems into smaller components, so they can be better understood and analyzed. The function point metric was devised in 1979 by Allan J. Albrecht as a means of measuring software size, functionality and productivity. It uses functional and logical entities such as inputs, outputs, and inquiries to measure the functions performed by a given software system.

COCOMO (Constructive Cost Model) was introduced first by Barry Boehm computes software development effort. COCOMO II uses cost drivers with values derived from historical project data. The attributes of the cost drivers has values on six-point scale that ranges from ‘very low’ to ‘extra high’

COSYSMO (COnstructive SYStems Engineering Cost MOdel) is a parametric model introduced by Boehm to estimate system engineering cost. It is created to be the newest member of the COCOMO family. COSYSMO depends on a number of drivers used to establish its model. Drivers are size drivers and effort drivers. Size drivers include number of requirements, interfaces, scenarios, algorithms and volatility factor. Effort multipliers include Requirement understand, architecture complexity, level of service requirements, migration complexity and technology maturity

Measuring complexity of change affects the estimated efforts needed to deliver these changes which affect the estimate of time and cost needed to deliver. Such complexity measurement will help in generating a more realistic time estimate for the software application by using software requirements specification document, relations between requirements and requirements properties.

This research aims to define set of metrics that affect the complexity of change and propose a methodology for measuring this kind of complexity. The proposed methodology will be a hybrid of code-based measures making use of the code already existing as well as cognitive-based measures. We plan to make use of the nature of a change by extracting attributes from already implemented requirements and study its effect on the changes to be measured. The proposed complexity measure and estimation methodology will help on determining change request size on an early stage of the change request life cycle. The proposed analysis will be applied on real life software project as a proof of concept.

II. RELATED WORK

Ashish Sharma [2] proposed a requirement based complexity metrics based on IEEE requirement Engineering document. Its' proposed approach identifies complexity of software immediately after freezing the requirement in SDLC (Software Development Life Cycle) process. In this approach, ten complexity attributes have been defined from the document that will help in the complexity measure and results are compared with various code based and cognitive based complexity measure like KLCID (Kinds of Lines of Code Identifier Density) [15], CFS (Cognitive Functional Size), CICM (Cognitive Information Complexity Measure) [7], McCabe and Halstead.
The defined attributes that are used in the proposed complexity measurement are extracted from the SRS. Some of these attributes are calculated from the SRS, other attributes are extracted from COCOMO drivers (i.e. PCA).

It was found that the proposed complexity measure compares well with established complexity measures like McCabe and Halstead. Although it may not have similar values but it follows the trend of all other established measures.

Sharma’s proposed metrics does not deal with a change differently, the scope of his equations covered estimates of requirements of the initial system.

Yogesh Singh [17] proposed an algorithm to measure the strength of individual requirements or a category/sub-category of requirements. The strength of an individual requirement or a category as defined by Yogesh is ‘a measure of its relative dependency on individual requirements in various categories/sub categories of requirements’

Yogesh calculates a value for each requirement ranges between 0 and 1 where 1 means that this requirement is highly dependent on other requirements and hence will have greater impact when it is changes. A value nearer to zero means that this requirement is highly independent and will probably cost less on change.

The difference between Sharma and Yogesh is that Sharma focused on the properties of a requirement independent of other requirements. Yogesh focused on the relation between requirements and other requirements. Yogesh outputs a value that shows the strength of a requirement but did not go further to measure the complexity of each requirement for sake of time estimates. However, requirement complexity can be considered one of the attributes that will be used as a step to measure change request complexity and provide estimate for it.

De Tran-Cao [6] analyzed software complexity at the analysis phase of the software life cycle. He proposed three measures for software complexity. The three measures are number of data groups (NOD), Number of conditions (NOC) and the entropy of the system (EOS). He proposed an entropy measure to capture the complexity in relationships between functions. He used a weighted directed graph to represent relationships between software functions where each function is considered a node on the graph.

Chetna Gupta [4] proposed an approach for estimating change/impact analysis. He takes into account both the change type and its effect on the system. Changes are classified to four categories: Functional Change Impact, Logical change impact, Structural change impact and behavioral change impact. It analysis the change after it is implemented to help software testers to select regression tests that are needed for each change instead of running all test cases after each change.

James O’Neal [10] proposed a traceability-based methodology to predictably evaluate requirement changes for software development project, he called it TIAM (Trace Based Impact Analysis Methodology). This methodology is applied during the stage of planning the incorporation of a set of requirement changes into a version of a software product under development.

O’Neal extracts the information from requirements by defining a model called WoRM (Work Product Requirement Trace Model). WoRM is the input for TIAM. TIAM produces a set of classes of requirement
changes that have similar impact levels ordered from low to high impact. TIAM also identifies potentially impacted work products by generating a set of potentially impacted work products for each requirement change.

N. Nurmuliani [13] studies the impact of requirements volatility on software development effort estimation. He used a number of attributes extracted from change requests and project data that can be used in his estimates.

N. Nurmuliani aim was to find a relation between requirement change attributes and change effort. He concluded that some change types are more expensive than other types (i.e. missing requirements). He found also that all change attributes defined are statistically significant contributors to change effort.

As we presented in this section key research initiatives in the proposed area of study, we noticed that studies focused on one of the following (1) Estimating the complexity of a new software project by defining metrics extracted from Requirements document, or (2) Measuring change impact on an already existing software by making use of traceability links between requirements. In our proposed approach, we analyze the metrics that can affect the complexity of a change considering the possible effect of attributes on change complexity measure. Traceability links between requirements will be considered in our analysis attempting to define different kinds of relations and the effect of each kind on the change complexity.

III. PROPOSED METHODOLOGY

In this paper, we propose a new methodology to measure complexity of a change in a software system. It is recommended to use the proposed approach in agile development as it is responsible for handling the problem of undefined, changing, and emerging requirements.

A. Scope and Assumptions

The scope of the proposed approach is as follows:

- Measuring the change after releasing at least one version the system
- Covering change requests related to code changes (i.e. Documentation are not covered)
- Considering horizontal relation between requirements. Vertical traceability is out of except for tracing requirements with test cases and defects
- Measuring complexity of NFR is not covered since changes in non function requirements can affect the entire system including the system architecture itself. This is going to make the complexity estimate to a different level than what is discussed in this approach

The assumptions that we are putting are:

- The SRS is always updated with system requirements.
- The changes are applied on OO programs
- Requirements has specific attributes that will be used in the proposed calculations
- Traceability between requirements and each other are well defined
- Requirements are categorized in modules (optional)
Any change in software reflects directly on requirements in the SRS. The change is either a new requirement that will be added to the SRS or an existing requirement that is updated.

B. Complexity Attributes Categories

Each requirement in the SRS has implicit characteristics. For a large number of requirements in the system, these characteristics will not be quite obvious to decision makers. Relations between requirements will not be clear especially the relations that involve more than one level. For example, if Requirement A is related to Requirement B and Requirement B is related to Requirement C (transitivity), this will imply a relation between Requirement A and Requirement C, this relation that will affect the complexity of Requirement A, will not be obvious for decision makers although it can be extracted from the document.

Some of the requirement attributes added in the SRS directly affect the complexity of this requirement which in turn affect the estimate to develop it. Changes in some requirements require changes in other requirements connected to it depending on the type of relation with this requirement.

Other factors are believed to affect the complexity of the change, these factors can directly relate to requirements being change or factors that are related to the characteristics of the change itself. Other factors are related to the environment and how it will be affected by this change.

We propose a set of attributes that we believe affect the complexity of a software change, proposed complexity attribute are categorized to four categories (1) System/Environment Attributes, (2) Changing Module Attributes, (3) Changing Requirement Attributes and (4) Change Attributes

C. System/Environment Attributes

This will include attributes related to the system itself and the environment. This category will be important in case complexities of changes from different systems are analyzed. Suggested attributes are:

- **System Stability (SS):**
  
  This represents the system vulnerability to change. It has been noticed that software maintainability degrades as changes are made to it which increases complexity of the software [14], system stability will be calculated as in (1)

  \[
  SS = \frac{#OR + #CNR + #CUR + #CDR}{#OR}
  \]

  (1)

  Where #OR is the Number of original requirements in the system. #CNR is the Cumulative Number of Requirements added to the system. #CUR is the Cumulative Number of Requirements updated in the system. #CDR is the Cumulative Number of Requirements deleted from the system.

- **Team Stability (TS):**
  
  As team members change in the developed software, complexity increases, this is because new members have different styles in writing their code and being new in the system increase the risk of adding defects. This is measured as in (2):

  \[
  TS = \frac{#ETM}{#CTM}
  \]

  (2)
Where \( #ETM \) is the number of team members currently working in the system. \( #CTM \) is the cumulative number of team members who worked in the system.

The relation between team stability and software complexity is inverse relation. As this number tends to 1, it means that the team is stable and complexity will decrease. As this number tends to 0, this involves increase in the system complexity

- **Staff Skills:**

  This attribute is introduced from cost drivers of COCOMO. In COCOMO, 5-degree scale personnel cost drivers are defined. These cost drivers include analyst capability, application experience, software engineer capability, virtual machine experience and programming language experience. Staff skills in our proposed methodology is calculated as in (3)

\[
Staff\ Skills = \sum_{i=1}^{#ETM} \sum_{j=1}^{Cost\ Drivers} \frac{j}{#ETM}
\]  

(3)

The values of cost drivers will be selected according to the role of the team member in the project

- **Defect Density (DD):**

  This is usually measured by the number of defects compared to the LOC. We propose to measure the number of defects compared to number of requirements in the system. As the density increases, this indicates the system complexity. Defect Density will be calculated as in (4)

\[
DD = \frac{#SSD}{#Requirements}
\]  

(4)

- **Program Age (PA):**

  As new versions are releases in the system, the complexity increases. Releases involve new updates, changes or bugs that are solved. Measuring program age helps in considering hidden changes that are not considered in change requests and defects that are considered some form of change in the system. Program Age will be calculates as in (5)

\[
PA = \frac{3(#MjR) + 2(#MnR) + #PtR)}{6}
\]  

(5)

Where \( #MjR \) is the number of minor releases, \( #MnR \) is the number of major releases. \( #PtR \) is the number of patch releases. As the program age increases, software complexity increases as well.

**D. Changing Module Attributes**

This will include attributes extracted from the module where change will take place. Suggested attributes are:

- **Module Size:**

  This is the module size compared to the system, changes in large modules will probably increase the complexity that changes in small ones because it will require changes in larger number of requirements and will reflect larger number of test cases. Module Size will be calculates as in (6)

\[
Module\ Size = \frac{#MR}{#Requirements}
\]  

(6)

Where \( #MR \) is the number of requirements in that module.
As the module size tends to one, the complexity of the change increases as well.

- **Module Stability (MS):**

  This represents the module vulnerability to change within the system. Since software maintainability degrades as changes are made to it [14], this should apply as well to any module within the system.

  As this factor increases, the complexity of new change will increase as well. Module Stability will be calculated as in (7)

  \[
  MS = \frac{\#ORM + \#CNRM + \#CURM + \#CDRM}{\#ORM} \quad (7)
  \]

  Where \#ORM is the number of original requirements in the module. \#CNRM is the cumulative Number of Requirements added to the module. \#CURM is the cumulative Number of Requirements updated in the module. \#CDRM is the cumulative Number of Requirements deleted from the module.

- **Fan In:** It represents the number of modules that use this module (i.e. call it) while Fan out represents the number of modules that are called by this module. In this approach, we propose that fan-in affects the complexity of a change more than Fan out because changes in a module may affect the functions that are called by out modules (fan in). In the same time, the functions that are called by the changed module will not be affected by the changes in the module.

**E. Changing Requirement Attributes**

This category will include all attributes extracted from requirement(s) being subject to the analyzed change. A change may require more than one requirement to be changed. A change might require more than one requirement to be added, updated or deleted. Attributes will be extracted from each requirement to analyze the impact of change on each one of them and calculate a complexity measure for these requirements combined together. Suggested attributes are:

- **Requirement Type:** New requirements usually need more effort than modifying an already existing requirement which in turn needs more time than deleting a requirement. This will be one of the following types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>3</td>
</tr>
<tr>
<td>Modification</td>
<td>2</td>
</tr>
<tr>
<td>Deletion</td>
<td>1</td>
</tr>
</tbody>
</table>

  In this approach, we assume that adding new requirement involve higher complexity than updating an existing one, similarly, updating an existing requirement involve higher complexity than deleting a requirement. We propose requirement type weights according to its effect on the complexity of a change.

- **Risk Level:** Scale from 1 to 5 where 1 means lowest risk and 5 means highest risk

- **Dependencies:**

  Each requirement can be related to one or more other requirements. Previous researches showed the importance of stating this relation. This is called traceability. Unfortunately, traceability links only shows the
existence of a relation between two requirements but do not show the kind of relation between these two requirements. We believe that analyzing the relation between requirements can show different types of relations that affect software complexity in different way. We propose the following types of relations (1) GUI where Two requirements share the same/common GUI screens, (2) Data where Two requirement manipulate common data (tables in the database, global data), (3) Code where Two requirements share bulk of the business logic code (one or more functions) and (4) Effect where Requirement A provides service/data used by Requirement B.

It is necessary to consider indirect dependencies as well. When calculating indirect dependency, impact should decrease with increasing path length.

\textbf{Single Requirement Complexity:}

Jorge [11] identified four main complexity perspectives to measure complexity of a process. We are proposing to apply these measures on a single requirement instead of a process. Measures are (1) Activity Complexity, (2) Control-Flow Complexity, (3) Data Complexity and (4) Resource Complexity.

\textbf{Defect Rate:}

Maintenance complexity and cost increases significantly based on the defect rate as stated in [9]. The analysis was made on the complexity of a software but it should be applies on the complexity of a single requirement. Defect Rate will be calculated as in (8)

\[ \text{Defect Rate} = \frac{\#SSDR}{\#RTC} \]

Where #RSD is the number of discovered sever defects discovered related to this requirement. #TC is total number of test cases related to this requirement.

\textbf{Previous time spent in this requirement: I}

If the change is a modification or deletion, actual time taken to implement this requirement will be considered with respect to total development time. As time increases, this means that this requirement is more complex. It will be calculated as in (9)

\[ \text{Requirement Original Time} = \frac{\#TR}{\#TS} \]

Where #TR is the Time/Effort spent in this requirement. #TS is the Total time/effort spent in the system. As this number tends to 1, this involves higher complexity for the requirement.

\textbf{Requirement Stability:}

Volatile requirements are considered one of the most important cost drivers and hence it is an important factor that affects complexity. Requirement stability calculates a value that shows the tendency of the requirement to change over time. It is sometimes called ‘Requirement Sensitivity’.

As this factor increase, the complexity of new change will increase as well. Requirement Stability will be calculated as the number of times requirement was updated.
F. Change Attributes

This will include attributes extracted from the characteristic of the change itself. Suggested attributes are:

- **Change Effect**: This will be a value from the following \{corrective, adaptive, perfective, and preventive\}. We believe that each type of maintenance introduces different complexity to the change. Each type will have a different weight showing the complexity of the change.

- **Change Type**: This will be a value from the following \{Add, Modify and Delete\}. Analyzing the data of the projects we have showed that changes that are considered an ‘addition’ to the system always requires more time than modifications which in turn requires more time than deletions.

- **Change Impact**: This will be the area where the change effect will impact. This categorization is modified from taxonomy proposed in [8]. Values will be one or more of the following values \{Data Handling, Parameter Handling, Control Flow, User Interface, Computation, Module Interface and Initialization\}. Each of the following categories will introduce extra complexity to the change.

The final equation that will be used to calculate change complexity will be a function of all attribute in the four previously discussed categories. Change Complexity Measure (CCM) will be calculated as in (10)

\[
CCM = SC + CMC + CRC + CC
\]

Where SC is the complexity calculated from system/environment attributes. CMC is the complexity calculated from Changing Module attributes. CRC is the complexity calculated from Changing Requirement attributes. CC is the complexity calculated from Change attributes.

To test the results of the proposed equation, we will use data of change for two projects. The first project is an agile project implemented from 4 years and is subject to continuous changes since then. The second project is implemented from 2 years and was subject to a number of changes through these years. Part of the data will be used to define weights for some of the attributes discussed; other part will be used to test the effectiveness of the proposed equation.

Calculating Change Complexity Measure will give decision makers an idea about what changes involve higher complexity than others. This will help them estimate the effort and time for their project and plan what changes to be made based on the given budget and time frame. We recommend that this methodology be applied in agile development as it is based on the idea of ‘welcoming changes’. This will help in planning for agile sprints and decreasing the risk of selecting changes that will take much longer than expected.

IV. CONCLUSION

The intent of this paper was to provide insight of how the proposed methodology works and how the suggested attributes can affect the complexity of change in the system. The proposed methodology is highly dependent on a very well defined SRS and traceable requirements. It makes intensive use of relations between requirements and each other. We introduced a categorization for relations between requirements and each other because it was noticed that not all relations make same effect on complexity of a change.
Trying this approach on sample data, we plan to reach results that follow the same trend of the actual estimates. That is, whenever a change is found to have long time relative to other changes in the real data, the proposed methodology results give same results showing that this change has higher complexity than other changes.

We still have significant challenges like trying this approach on real data for a number of projects in different companies. We plan to continue reporting the status of our efforts as we move in this research.

ACKNOWLEDGMENT

Grateful thanks are due to Centrivision software house for having facilitated the industrial data being used in our analysis.

REFERENCES


