The Impact of Rate of Change on Object Oriented Paradigm

Mohammed GH. I. AL Zamil and Eman AL Momani

Department of Computer Information Systems,
Yarmouk University,
Irbed, Jordan
Mohammedz@yu.edu.jo, emolgf2005@yahoo.com

Abstract

Object oriented technology accommodates a set of relationships that affect the quality of coding and, therefore, output programs. In this paper, we present a hybrid technique that relies on the paradigm of dependency graph to detect the impact of change over time on object oriented relationships such as: inheritance, class size, coupling and cohesion. Our goal is to provide a systematic framework to measure how rate of change affects such features in a concurrent environment. The answer of such question takes place in code analysis and model checking of concurrent programs. The contribution of this research is to study the static and dynamic aspects that contribute in enhancing the quality of coded programs, which implies increasing maintainability.

In order to satisfy our research goal, we performed experiments on a collection of concurrent Java programs, analyzed them, and then concluded the behavior of these programs over time. The results indicated that there is a significant positive correlation between some of these features and rate of change.

Keywords: Rate of Change; Object Oriented Paradigm; Model Checking

1. Introduction

Program understanding and analysis depends on its internal dependencies; the connectivity among all constituents of the program [1]. Several software engineering applications such as reverse engineering and software maintenance take into account the structure of the software program. Object oriented paradigm defines relationships to relate internal entities such as classes and attributes. As time moves on, code might be changed to cope with user needs and the natural evolution of a software product, which could affect the quality of resulted program.

Studying internal dependencies has attracted researchers for the purpose of enhancing the coding task. Recent techniques focused on modeling programs in order to provide information about the relationships among its components [2]. However, one of the most interesting visual descriptions was the dependency graphs, which depict the connectivity among different program parts. Previous researches implement dependency graphs in order to extract and quantify certain characteristics of programs under analysis. For instance, it could provide useful information at tasks such as debugging, testing, fault detection and embedded security issues.

Such information is hard to detect in a concurrent environment. Many object oriented programming languages support multithreading programming paradigm, in which many threads run at the same time. Such feature might lead the execution to an unpredictable behavior. In fact, concurrent programs are much complicated as compared to sequential ones. Therefore, there is a need to intelligent techniques to check some programs’ features using a sort of static analysis. However, static analysis does not provide reliable conclusions as dynamic behavior is crucial in such situations. Thus, a hybrid framework of both code and
model analysis would be of great interest to achieve our goal as model checking mimic the dynamic behavior of concurrent systems.

In this paper, we proposed a framework to analyze and understand the behavior of Java programs for the purpose of enhancing their quality. The goal is to measure the effect of change over time on different object oriented relationships. We used to model Java programs using Code Metrics tool, which provides the static aspects of our analysis. Furthermore, we extracted the dependency graph from the resulted models to analyze the static aspects of such programs. Moreover, to ensure that Java concurrent programs preserve its temporal properties, model checking techniques have been applied using UPPAL tool.

In order to evaluate the proposed framework, we performed experiments on a collection of Java classes in order to elicit information about the relationship among inheritance, size, coupling, cohesion, and rate of change. Then, we reported the results, which included a statistical analysis to support our conclusion.

The rest of the paper is organized as follows: Section 2 discusses the related work to our method. Section 3 offers a background of techniques used in this research and clearly defines the research questions. Section 4 explains the framework of our proposed method. Section 5 illustrates our experiments on the benchmark collection. Section 6 comments on the results and the limitations of this work. Finally, Section 7 summarizes the contributions and the conclusion of this research.

2. Related Work

Program analysis has been studied by many researchers for the purpose of enhancing coding quality. Recent researches have focused on advances in model checking techniques [3, 4, and 5] and program slicing [6, 7] in order to verify some systems’ properties. Separation of code and model analyses does not provide enough information to handle the relationships among some emergent quality attributes. In fact, studying such attributes requires combining knowledge from both sources as behavioral aspects emerge to show how a system acts and reacts to a particular event or input. Moreover, static analysis of code gives an indicator of coding quality and organization of code as well as dependencies among different entities within programs.

Further, object oriented relationships, such as cohesion, affect program testability. In other words, it affects the process of generating test cases automatically as concluded in [8]. This study highlight the importance of studying object oriented relationships in order to discover its impact on specific development tasks such as program correctness. The study also provides a detailed review of many metrics that contribute in quantifying and analyzing the impact of some attributes on the complexity of object oriented programs.

In the literature, there are several tools to analyze programs that provide variety of valuable information to understand the code. The Java System Dependency Graph (JSysDG) has been proposed by Neil Walkinshaw [9] to provide query-like manipulations on the dependency graph. DA4Java [10] is another example of static tools to analyze Java code that relies on combination of top-down and bottom-up approaches to extract detailed information about programs under analysis. Indeed, such tools contribute to the development of useful coded projects as they give the opportunity to developers to discover design errors and enhance the overall quality of programs.

Such tools are still insufficient to act as a mature framework that guarantees producing reliable software. Changes that could be suggested as a result to static analysis do not take into account the dynamic and temporal properties of concurrent programs. Such properties are crucial and essential for multithreaded and concurrent activities. While static analysis gives a value on the right hand, we, in this research, give equal value to the model analysis on the other hand. For this reason, we believe that a hybrid framework of both techniques would be of great interest for producing high quality code.
Program structuring is a well known static property. The main goal in this research is to experiment how we could enhance such static property while preserve its dynamic consequences. IBM’s Structural Analysis for Java (SA4J) [11] is an example of structural analysis tools. Its distinguishing feature is to detect anti-patterns in Java programs. However, like other static analysis tools, AS4J provides no guarantee that changes according to program analysis would preserve dynamic properties of code.

Recent software engineering researches have focused on formal methods to verify software dynamic properties using model checking [12]. The idea behind model checking is that the application can be normalized to check every possible state that could be generated during the life of a given program or design. Specifically, constructing a formal model that describes the behavior of the system during runtime as finite-state automata, and then, checking the existence of some properties in this model. Although, formal methods might not guarantee the performance of programs [13], we applied it for verifying correctness only.

3. Background and Problem Definition

In this section, we will provide background information including brief mathematical definitions of dependency graphs and model checking. Furthermore, we will end the section with a clear set of research questions that would be answered through the proposed methodology and experiments.

3.1. Dependency Graph

A dependency graph is a set of entities depicted as nodes within the graph and connected by edges. An edge depicts a relationship between two nodes which represents dependency between two entities. In this context, we define the dependency graph as follows:

Given a set of nodes \( N = \{n_1, n_2, \ldots, n_k\} \) and a set of edges \( E = \{e_1, e_2, \ldots, e_n\} \), a dependency graph \( G \) is defined as \( G \subseteq (N \times E) \) in which the following conditions MUST hold:

1. \( e \rightarrow \rho(n_i, n_j) \)
2. \( i \neq j \)

Where \( \rho \) is a mapping function between the two nodes \( n_i \) and \( n_j \) that represents a relationship between the two nodes. Such relationship should represent a sort of dependency in which any change in the entity \( n_i \) might affect the structure or the behavior, for instance, of entity \( n_j \). Furthermore, the second condition assert that circulation on the same entity is not allowed in dependency graphs. Figure 1 shows a dependency graph example.

![Figure 1. Dependency Graph](image)
3.2. Model Checking

Given a system modeled in some specification language as M with starting state s, model checking problem is defined as the automatic extensive checking of whether this model satisfies a given specification or property p.

\[ M, s \models p \]  \hspace{1cm} (1)

In order to find an objective solution to this problem, the model and its specifications are formulated using a formal language (mathematical bases). However, the concept of model checking is general and could be suitable to handle several kinds of structures. In this context, we propose to model programs under analysis using this technique for the purpose of verifying that the coded program is still preserving its designated properties after reengineering changes.

3.3. Research Questions

Followings are the research questions that we will try to answer through this research. Is the information provided by the dependency graph enough to handle the ripple effect of changing some entities in a given system? What if the system has a transitive dependency in which an entity change might affect the correctness of the overall system state? What about the specifications? How to ensure that the target system is still preserving its designated specification after implementing changes?

These questions are crucial and, in many systems, are critical. In fact, there is no way to answer such questions without checking every possible state. Model checking provides the means of answering these questions by generating every possible state that a given system might enters during runtime.

In this research, we are going to evaluate a framework that combines both static analyses using dependency graph and model analysis by model checking target systems written in Java.

4. Framework

In this section, we describe the phases of our proposed framework to detect the impact of change over program attributes and predefined specifications. First, we will discuss the code analysis phase. Then, provide brief information and description regarding model checking technique.

4.1. Code Analysis

Dependency information is typically collected from the source codes during the static analysis phase. Locating dependencies among programs requires analyzing fields and methods through their reference locations. During the construction of object oriented programs, compilers create class files which hold most necessary information to proceed in analysis. The following subsections illustrate this process.

4.1.1. Extracting Raw Dependency Data:

Metadata within class files provides references to parent classes. Such information comprises the architectural dependencies among the constituent classes. Indeed, such general dependencies cover external classes, methods, and fields that may contribute in class’s behavior. Thus, the static analysis tool would be able to extract different kinds of static relationships including inheritance and associations.
4.1.2. Reference Dependency Locations:

A major phase in program static analysis is to scan the source code for the purpose of locating references of different entities. Therefore, a tool would be able to make a semantic connection among nodes within dependency graphs. In other words, connect constituents according to a specific description of a relationship. So, we could extract useful information from code such as: Inheritance, Composition, Fields, and Number of classes, Number of Interfaces, Depth of Inheritance, and Number of subclasses in an inheritance relation.

4.2. Model Analysis

Code analysis is able to collect the required information that lead to comparing relationships with each others. However, in some types of critical systems, especially concurrent programs, dynamic analysis must take place for the purpose of measuring the change as a comparison direction in this research. The effect of changes on coded programs has dynamic aspects. It must be measured or quantified during the code execution. Changes might take place at compilation time or at run time; as system states are changed from time to time (e.g., The Temporal Property of Concurrent Execution).

In this research, we will model and check coded programs in order to enrich our conclusion with dynamic aspects of changes during execution time as well as the static ones. The goal is to verify that changes will not affect the correctness of the overall system specifications. Figure 2 depicts how models are checked using this technique.

![Figure 2. Model Checking Phases](image)

5. Experiments and Results

In order to evaluate the effect of rate of change on inheritance, coupling, cohesion, and size of classes, we conducted a set of experiments that take into account the static and behavioral aspects of Java programs. First, we will describe the experimental collection of programs. Then, describe the experimental setup including evaluation metrics. Finally, we will list and comment on the resulted data.

5.1. Experimental Collection

For the purpose of testing the proposed framework, we collected a large number of open source Java programs from different resources. Further, to proceed with our research goal of
studying evolution, we chose only programs available in 5 different releases or versions. Thus, our collection will hold changing programs in terms of releases. Then, we filtered the sample programs to include only concurrent Java programs. The resulted collection consists of 380 different concurrent programs. Further, these programs hold 6400 classes and consist of more than 1 million lines of code. For the purpose of this research, we ran our algorithms on each version independently in order to measure how dependencies affect the rate and correctness of changes. Table 1 shows a numerical description of each version.

<table>
<thead>
<tr>
<th>Version</th>
<th>NO. Programs</th>
<th>NO. Classes</th>
<th>LOC (Rounded)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver. 1</td>
<td>380</td>
<td>6800</td>
<td>1000K</td>
</tr>
<tr>
<td>Ver. 2</td>
<td>380</td>
<td>7255</td>
<td>1200K</td>
</tr>
<tr>
<td>Ver. 3</td>
<td>380</td>
<td>8010</td>
<td>1250K</td>
</tr>
<tr>
<td>Ver. 4</td>
<td>380</td>
<td>8642</td>
<td>1420K</td>
</tr>
<tr>
<td>Ver. 5</td>
<td>380</td>
<td>9231</td>
<td>1601K</td>
</tr>
</tbody>
</table>

Figure 3 shows the distribution of programs as compared to their size. The figure highlights that our selected collection includes programs of different sizes.

5.2. Experimental Setup

In this section, we will describe the metrics used to construct the dependency graphs and to report the results, the way we compute the rate of change of collected programs, and the model checking tool that will be used to verify programs’ specifications.

5.2.1. Metrics:

In these experiments, we used DIT (Depth of Inheritance), NOC (Number of Children), and LOC (Line of code). These Metrics have been used in order to measure the complexity of the inheritance relation for a specific class and to quantify the class size [14]. Although there are many metrics to compute class size, such as function-point, we chose LOC as it reflects functional and non-functional blocks of code. Moreover, LOC is widely used in many software engineering tools, easy to understand, and easy to compute as compared to other measurements. Thus, our experiments could be re-implemented using different tools in which LOC is the only size metric.
In this paper, we used Code Metrics tool [15] for the purpose of computing DIT and NOC. Figure 4 shows an example of code metrics snapshot. For DIT, code metrics tool relies on the definition in [16] that considers DIT as the maximum length from the node to the root of the inheritance tree.

\[ DIT = \text{max} \text{length from node to root} \]

Figure 4 shows an example of code metrics snapshot. For DIT, code metrics tool relies on the definition in [16] that considers DIT as the maximum length from the node to the root of the inheritance tree.

In addition, we used DOC (Degree of coupling) and DCH (Degree of Cohesion). These two metrics provide our study with the following advantages:

1. Measuring the functional strength of the classes of concurrent Java programs.
2. Measuring the dependency of the classes to construct the dependency graph.

We used the models in [17] to compute DOC and DCH as follows:

\[ DOC = \frac{MRC}{MPC} \]  \hspace{1cm} (2)

Where MRC (Message Received Coupling) calculates the complexity of messages that have received from a specific class and MPC (Message Passed Coupling) computes the number of messages that have been passed among objects of the class.

\[ DCH = \frac{NAU}{TNA} \]  \hspace{1cm} (3)

Where NAU is the number of attributes used in the class and TNA is the total number of attributes. Both DOC and DCH can be analyzed and computed using code metrics tool by constructing dependency graphs for classes and attributes to analyze the code.

5.1.2. Rate of Change:

As the experimental collection is classified into a set of versions, each version is considered as a change. In other words, the rate of change of a class is computed as the number of times it appears in different versions. Although time is a measure factor to analyze rate of change, we ignore it in this research as code analysis is a static activity.

5.1.3. Model Checking:

For the purpose of checking whether the system confirms to its specifications in a concurrent environment, we used the model checker UPPAL [18] to handle this task. UPPAL is a modeling tool to verify software that is based on constraint solving and on-the-fly techniques. It has been developed to verify non-deterministic processes with finite control structure. The design goal of UPPAL is to check invariant and reachability properties by exploring the state space of a system. Therefore, we chose this tool as our specifications are mainly focusing on reachability. Figure 5 shows a sample UPPAL model generated from our collection.
5.3. Results

As described in the above section, our collection of programs is divided into five sets in which each one of them represents a change to the previous one. Notice that, the newer versions are really different from the previous ones as they have been modified to reflect some changes. To list our experimental results, we extracted metrics about coupling, cohesion, code level metrics to measure inheritance relations, rate of change, and model checking reachability properties.

Table 2 shows the values of message passed coupling and message received coupling. These values were calculated by examining the message calling graph that has been generated automatically using DMS Software Reengineering Toolkit [19]. The resulted degree of coupling value has been rounded up for each version of classes.

Table 3 displays the measurements needed to compute the degree of cohesion. Here, the metric takes into account the number of attributes that have been used by the methods within the class while ignoring the others.
Table 4 shows the information extracted directly from the code itself using code metrics tool. This information includes the number of lines of code (repeated from Table 1), the average depth of inheritance, and the average number of children (which is reported for each class but to avoid huge Table size we reported the average).

Table 4. Code Metrics

<table>
<thead>
<tr>
<th>Version</th>
<th>LOC</th>
<th>Average Depth of Inheritance</th>
<th>Average NO. Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver. 1</td>
<td>1000K</td>
<td>≈2.30</td>
<td>≈4.2</td>
</tr>
<tr>
<td>Ver. 2</td>
<td>1200K</td>
<td>≈2.35</td>
<td>≈5</td>
</tr>
<tr>
<td>Ver. 3</td>
<td>1250K</td>
<td>≈3.00</td>
<td>≈6.4</td>
</tr>
<tr>
<td>Ver. 4</td>
<td>1420K</td>
<td>≈3.55</td>
<td>≈6.9</td>
</tr>
<tr>
<td>Ver. 5</td>
<td>1601K</td>
<td>≈3.80</td>
<td>≈7.2</td>
</tr>
</tbody>
</table>

Table 5 reports the ratio of changes on newer versions of classes for the same set of programs. It represents the rate of change as compared to the previous version.

Table 5. Rate of Change over Versions

<table>
<thead>
<tr>
<th>Versions</th>
<th>Rate of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver. 1</td>
<td>0</td>
</tr>
<tr>
<td>Ver. 2</td>
<td>≈1.06</td>
</tr>
<tr>
<td>Ver. 3</td>
<td>≈1.11</td>
</tr>
<tr>
<td>Ver. 4</td>
<td>≈1.08</td>
</tr>
<tr>
<td>Ver. 5</td>
<td>≈1.07</td>
</tr>
</tbody>
</table>

Finally, we performed model checking on every version independently and reported the number of unreachable states, number of faulty states and the ratio of satisfied specifications for each version; according to programs’ specifications.

Table 6. Reachability Analysis (Model Checking)

<table>
<thead>
<tr>
<th>Versions</th>
<th>No. Unreachable States</th>
<th>No. Faulty States</th>
<th>Ratio of Satisfied Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver. 1</td>
<td>12</td>
<td>7</td>
<td>≈0.87</td>
</tr>
<tr>
<td>Ver. 2</td>
<td>15</td>
<td>11</td>
<td>≈0.82</td>
</tr>
<tr>
<td>Ver. 3</td>
<td>33</td>
<td>29</td>
<td>≈0.73</td>
</tr>
<tr>
<td>Ver. 4</td>
<td>56</td>
<td>34</td>
<td>≈0.73</td>
</tr>
<tr>
<td>Ver. 5</td>
<td>71</td>
<td>43</td>
<td>≈0.71</td>
</tr>
</tbody>
</table>

6. Discussion

In this section, we provide a detailed description and analysis of the empirical results. The following subsections are dedicated to describe the impact of rate of change over a set of program attributes. Notice that, we applied student t-test to verify the significance of our findings. The test has been performed on the exact values; i.e., not the averages. However, as the resulted database of numbers is very huge, we summarized the results using the average Metric.

Before proceeding with our detailed discussion, our collection of concurrent Java programs shows that there is a statistically positive correlation between LOC and rate of change. In
other words, the size of programs is increased overtime during enhancements and maintenance operations. Figure 6 depicts our conclusion.

![Figure 6. LOC over Versions](image)

6.1. Inheritance vs. Rate of Change

Inheritance represents the generalization-specialization relationship in object oriented programming languages. In order to measure the effect of changes over inheritance, we measured the depth of inheritance (DIT) and the change over the number of subclasses (NOC). To simplify the representation, Figure 7 depicts this correlation using the average of both metrics.

![Figure 7. DIT Vs NOC](image)

We found that there was a significant positive correlation between the depth of inheritance and rate of change. Also, the correlation between the number of subclasses (children) and rate of change was also founded positive. According to the documentation, every version reflects a set of enhancements over the previous one, which justifies our conclusion. Indeed, we cannot generalize this conclusion over all object oriented programs as, in many cases; changes might collapse the inheritance diagram. On the other hand, we measured the relationship between DIT and NOC and found that there was a positive correlation between them.
6.2. Coupling vs. Rate of Change

Coupling can be defined as interrelationship among classes within a program. It shows how classes are dependent on each other. During this analysis, we found that the correlation between coupling (DOC) and rate of change is negative; and sometimes stable (see Figure 9). We justify this conclusion by examining the values of MPC and MRC in Figure 8. It is clear that in this collection the increasing rate of MPC was faster than MRC especially at version 4 and 5. The actual reason behind this conclusion is that enhancements made on newer versions decrease the coupling among classes.

![MPC Vs MRC](image)

**Figure 8. The Rate of Passed and Received Messages**

![Degree of Coupling](image)

**Figure 9. DOC Value per Version**

6.3. Cohesion vs. Rate of Change

Cohesion metrics measure the homogeneity among class contents. However, our empirical results indicate that there is no clear evidence that the rate of change affect the cohesion metrics. Figure 10 shows that the total number of attributes and total number of used attributes are increased from one version to another. However, small fluctuation in this relation causes the cohesion metric to be fluctuated drastically in different point (see Figure 11).
6.4. Impact of Class Attributes Metrics on Rate of Change

Up to this point, we saw from previous subsections the correlations between rate of change and some relationships of object oriented programs. The remaining and most important research question in this paper is the effect of changes on the correctness of programs. Notice that, our collection represents concurrent Java programs in which each one of them has its own specifications.

First, Figure 12 shows two important aspects; unreachable states and faulty states. Our empirical results clearly show that the number of unreachable states increases as more changes are made. Also, the number of faulty states increases with respect to the rate of change. By this relation, we want to show that correctness is a critical and important aspect to measure the effect of changes.
Figure 12. Unreachable and Faulty States during Changes

Figure 13 shows that the ratio of programs specifications that have been satisfied decreased as more changes are applied. Meaning that, temporal and logical errors might arise from changes as well.

Figure 13. The Ratio of Satisfied Specifications

7. Conclusion

In this paper, we presented a hybrid framework to measure the impact of rate of change over different object oriented relationships. The contributions of this research are: 1) studying the impact of rate of change on inheritance relationship, coupling, cohesion and size of classes 2) measuring the impact of rate of change on program specifications.

The experiments performed in this research were based on three tools to extract static information and analyze the code: Code Metrics, DMS, and UPPAL. We evaluated the proposed method and reported the correlation among different OO relationships and rate of change. The results indicate that rate of change positively affects the complexity of program structure in terms of inheritance, cohesion, and coupling. Furthermore, we concluded that rate of change negatively affects the correctness of programs’ specifications.

References


