International Journal of Electrical and Computer Engineering (IJECE)
Vol. 10, No. 3, June 2020, pp. 2719~2733
ISSN: 2088-8708, DOI: 10.11591/ijece.v10i3.pp2719-2733

A study of code change patterns for adaptive maintenance with AST analysis

Omar Meqdadi, Shadi Aljawarneh
Department of Software Engineering, Jordan University of Science and Technology, Jordan

Article Info

Article history:

Received Jul 3, 2019
Revised Nov 25, 2019
Accepted Dec 6, 2019

Keywords:

Abstract syntax tree
Adaptive maintenance
API-migration
Change repertitiveness
Source code change patterns
Transformational approaches

ABSTRACT

Example-based transformational approaches to automate adaptive maintenance changes plays an important role in software research. One primary concern of those approaches is that a set of good qualified real examples of adaptive changes previously made in the history must be identified, or otherwise the adoption of such approaches will be put in question. Unfortunately, there is rarely enough detail to clearly direct transformation rule developers to overcome the barrier of finding qualified examples for adaptive changes. This work explores the histories of several open source systems to study the repetitiveness of adaptive changes in software evolution, and hence recognizing the source code change patterns that are strongly related with the adaptive maintenance. We collected the adaptive commits from the history of numerous open source systems, then we obtained the repetitiveness frequencies of source code changes based on the analysis of Abstract Syntax Tree (AST) edit actions within an adaptive commit. Using the prevalence of the most common adaptive changes, we suggested a set of change patterns that seem correlated with adaptive maintenance. It is observed that 76.93% of the undertaken adaptive changes were represented by 12 AST code differences. Moreover, only 9 change patterns covered 64.69% to 76.58% of the total adaptive change hunks in the examined projects. The most common individual patterns are related to initializing objects and method calls changes. A correlation analysis on examined projects shows that they have very similar frequencies of the patterns correlated with adaptive changes. The observed repeated adaptive changes could be useful examples for the construction of transformation approaches.

Corresponding Author:

Shadi Aljawarneh,
Department of Software Engineering,
Jordan University of Science and Technology,
Irbid, Jordan.
Email: saaljawarneh@just.edu.jo

1. INTRODUCTION

In software evolution, it is essential to work on sections of a source code to implement a particular change request [1], managing code clones [2], possibly refactoring code to enhance functionality [3], fix an implementation bug [4], fixing code smells [5], or adapting the system to changes in the framework, physical machine [6, 7], or APIs [8, 9]. Since evolution is an often process, developers need frequently to migrate to new releases of their employed API for improved services [10, 11]. Evolution to address changes to dependent platforms, APIs, and compilers are generally named adaptive maintenance tasks [12, 13]. The example-based construction of transformation rules to automate adaptive maintenance tasks have been demonstrated to radically reduce costs and improve quality of API-migration processes [14, 15]. Though, the principal problem of the example-based construction of transformation rules is the need for enough before and after real examples of adaptive code changes that had been made in the history of the system or other systems [14]. Furthermore,

another main challenge is that the examples of API-migrations previously made in the history must also be categorized and then be generalized [15-17] for better system evolution. Thus, evidences show that example-based transformation lies in categorizing repetitive API change edits into abstracting change patterns to create rules that reapply these repetitive changes into other locations in a system [15]. Consequently, mining and observing the most repetitive code edits that are relevant to the API-migration changes is an essential step to collect sufficient examples for the construction of example-based transformational approaches.

There is no doubt that the state-of-the-art studies enriched the software engineering research with the examination of the code change repetitiveness and the recognizing of change patterns relevant to the variety of software evolution tasks. Nguyen et.al [18] introduced a graph-based mining approach, CPATMINER, to mine a set of fine-grained change patterns from history repositories. Among the mined change patterns, their tool classified only 9% as patterns relevant to adaptive maintenance. However, the proposed approach focused on investigative the change patterns at the semantic level and thus narrows the potential comprehension of recurrent code changes of adaptive maintenance. Also, the mined change patterns are only for Java projects and so cannot be generalized for other diverse open-source systems. Niu et.al [19] proposed an approach that identifies API usage patterns based on the co-existence relations between object usages. Additionally, the approach helps recommending changes through retrieving a group of API usage patterns relevant to a given API method as a query. Although those studies focused on adaptive maintenance, their obtained results do not have general analysis on which kinds of adaptive changes are most common and what taxonomies of adaptive change patterns are over the history.

Other earlier studies, on the other hand, have examined code change patterns for non-adaptive software evolution at a high level of details. Campos et al [20] performed a large-scale study to explore the repetitiveness of several bug-fix patterns in history repositories of five Java projects. The study found the prevalence of five bug-fix patterns such as addition of if precondition check and method call with different number of parameters. Actually, the study that was undertaken in [20] focused on a set of patterns that were originally recognized by Pan et al. [21]. Pan and other authors manually classified fixing changes into several patterns based on the syntax of the change. Tsantalis et.al [22] presented a new technique for detecting the set of commonly refactoring change patterns through the comparison of source code between two system versions. Kim et al. [23] offered a taxonomy of signature change patterns over revisions to categorize observed changes through the analysis of eight open source projects. Nguyen et al. [24] presented an examination study for the code change repetitiveness in the histories of software systems through the modeling of code changes as pairs of old and new AST sub-trees at the granularity of methods. They are interested in examining repetitiveness of bug fixing changes and refactoring changes, however, without focusing on the change repetitiveness of other maintenance types such as the adaptive one. That is, prior studies have no sufficient data to draw conclusions regarding the most repetitive API-migration code edits and their relevant change patterns, though such conclusions are the intent that is required to construct qualified transformation rules for adaptive maintenance.

The work offered here aims to address the problem of collecting enough examples for the construction of code transformation rules for API-migration tasks. Our study is with two contributions. The first contribution is the exploration of the highly repetitive adaptive code changes across the histories of open source systems. Our second contribution is the reorganization of the prevalent adaptive change patterns and their frequencies across projects. The motivation of our work is the making of observations that directly guide the future research in automatic adaptive maintenance through the identification of qualified real code examples of changes that could be used later for transformation tool constructions and regression testing.

In this work, we conducted a large case study of six C++ and Java open-source systems that previously underwent major adaptive maintenance tasks. We collected a dataset consisting of 501 adaptive commits with 6380 change hunks (e.g., a continuous set of source code lines that are changed along with contextual unchanged lines). Then, we examined the adaptive changes at three level of granularity, namely commit, source file, and hunks. Our examination has been accomplished through sophisticated comparison algorithms that involve heuristic search on the AST. The main advantage of working at the AST level is the sufficiency in generating fine-grained syntactic code differences between two ASTs before and after undertaking a maintenance change, and hence detecting code change patterns [24]. We used the state-of-the-art AST differencing tool GUMTREE [25] tool to automate the computation of the AST differences between every change pairs (e.g., file versions before and after each undertaken adaptive commit) of all adaptive commits under consideration. Our results indicate that 76.93% of the examined adaptive changes could be represented only by 12 AST code differences. Finally, we evaluated the prevalence of 9 change patterns and statistically compared their repetitiveness frequencies across the examined projects. We found that these 9 patterns covered 64.69% to 76.58% of all adaptive change hunks. The rest of this paper is organized as follows. Section 2 presents our research methodology. Section 3 shows the obtained results and our discussion. We present the related work in section 4, followed by the conclusions and our plans for future research in section 5.

2. RESEARCH METHOD

In this section, we describe our methodology to collect adaptive changes from the version history to build our change database and compute their repetitiveness across adaptive commits. Our methodology is shown in Figure 1. Below, we will explain in details the aforementioned steps of our research methodology.

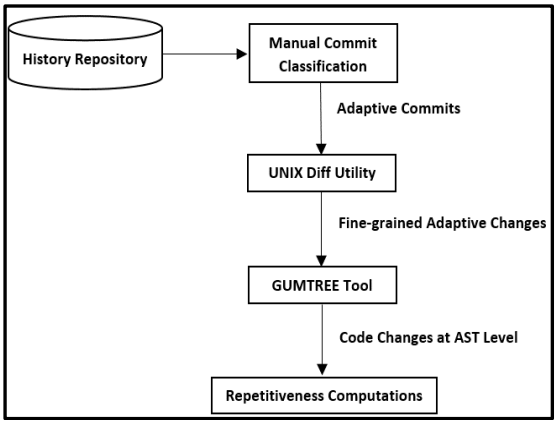


Figure 1. Block diagram of the research methodology

2.1. Data collection

To investigate AST changes in adaptive commits, three open source systems were selected for study. The subject systems are namely: the KDE package Kexi (visual database applications creator), the 3D graphics toolkit OpenSceneGraph (OSG), and the Java reporting library JasperReports. These systems were chosen using the following criteria: 1) have active history repositories from GitHub with a minimum of 500 commits, 2) well documented, 3) have a long evolutionary history that consists of successful API migrations and adaptation tasks, and 4) representative sample of domains and underlying APIs. Moreover, the history of the API migration tasks for these selected systems had been investigated in several previous studies [26, 27]. For instance, the API-migrations of KDE project and OSG system had been investigated in [26], while the migration of JasperReports to use JFreeChart API version 1.0.12 had been studied by Nguyen et al. [27].

The actual adaptive commits of our subject systems had been manually recognized by following the approach performed by Meqdadi. et.al [26], in which if the log message of a commit has the key terms indicating API-migration activities (i.e., involving known API's or language features that were changed to comply with features/interfaces found in the new API), the commit is considered as adaptive, and then the code changes in that commit are considered as adaptive changes. Table 1 shows the subject systems along with the undertaken adaptive maintenance task, examination time period, and the number of manually recognized adaptive commits of each system. We processed all the manually identified 346 adaptive commits of the three examined systems and parsed in total 1530 source files that being added, modified, and deleted by the considered adaptive commits. Recall that a source file could be changed by many commits. If a file changed by N adaptive commits, we count this file N times in our change dataset, since we focus on the code changes occurred in each commit independently. Pan et al. [21] demonstrated that an instance of a change pattern lies in the same source file and even within a single change hunk. Therefore, we processed all 346 adaptive commits and we used the GNU Unix diff utility to identify change hunks (e.g., modified, added, and deleted) having occurred in each adaptive commit. From this, we collected a data set consisting of 4737 adaptive change hunks. Table 1 summaries our dataset of adaptive changes.

Table 1. Selected open-source systems used in our study with their relevant adaptive change hunks

	Kexi	OSG	JasperReports
Language	C++	C++	Java
Adaptive Maintenance Task	Migrating to Qt5.x	Migrating to OpenGL 4.x	Migration to JFreeChart API version 1.5.x
Time Period	7/7/2014 –1/1/2018	1/1/2014 - 1/1/2017	11/11/2017 –1/1/2019
# Commits in the Log File	3283	1984	910
# Adaptive Commits	161 (4.90%)	126 (6.35%)	59 (6.48%)
# Adaptive Change Files	682	491	357
# Adaptive Change Hunks	2104	1521	1112

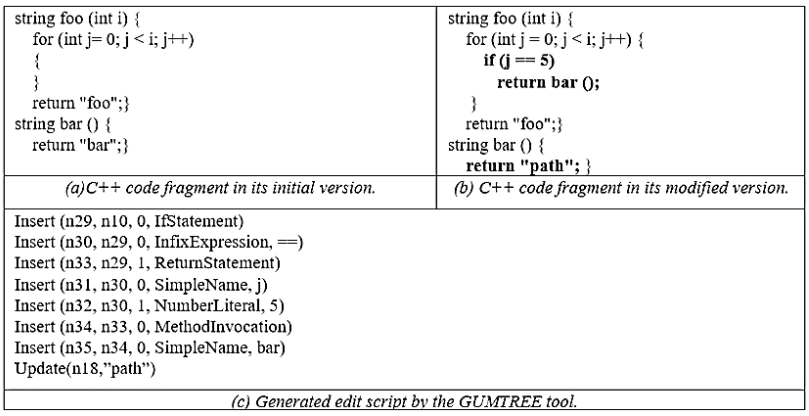
2.2. AST analysis of source code changes

Our methodology is centered on the analysis of source code differences between a pair of successive versions of a source file that was modified by an adaptive commit. There are several approaches for source-code versioning and differencing in the literature [28]. The most popular differencing algorithms is the extraction of textual differences (e.g., character-based view) using diff utility that identifies changed lines (e.g., modified, added, and deleted) recorded by their line numbers. The drawback of this approach is that it overlooks the underlying syntax-structure of the source code changes [28], since the differencing is character-based. Another approach is looking for differences at the syntactical level by a comparison of the two AST's before and after the change. The results of this approach are based on the tree operations occur on the individual nodes of an AST [29]. Syntactical differencing has been widely and successfully used in the literature [24, 28-30].

Thus, in this research, we have used the automated computation of source-code versioning and differencing between the ASTs that are extracted from the successive versions of changed source files. Since the subject systems in our study are C++ and Java open-source systems, we have choose the open-source GUMTREE tool [25], which has the ability to compute the source code versioning for both C++ and Java systems at the AST level, through the usage of srcML [28] format as a backend for the representation of C++ code. The GUMTREE is a complete framework that converts a source file/hunk into an AST format and also computes the differences between two given ASTs. Moreover, the evaluation results of GUMTREE show that it outperforms the diff utility and other tree-based differencing tools such as ChangeDistiller [29]. GUMTREE produces differencing results by computing a sequence of edit actions that transform one AST into another. The possible edit actions are as follows:

- Insert (N, PN, I, L, V): an inserting of a new node N as the I^{th} child of the node PN. The label of node N is L, and its value is V. Here, V is optional.
- Delete (N): deleting an existing node N from the AST.
- Update (N, V): updating the value of an existing node N with a value v.
- Move (N, PN, I): moving a sub tree rooted at the node N to be the I^{th} child of the node PN.

The edit actions to transform the AST version before the undertaken code changes to the AST version after changes are recorded in an Edit Script. The detailed description of the edit scripts and their contents are available in [25, 29]. To better understand the differencing results between two ASTs before and after code changes, let us consider the change example that is shown in Figure 2. The figure shows the initial and the modified versions of a C++ code, and also shows the corresponding ASTs before and after the code changes. Given two ASTs before and after code changes, the GUMTREE generates the corresponding edit script containing 8 edit actions, as the one shown in Figure 2. For instance, one of the edit actions in the script is (Insert (n29, n10, 0, IfStatement)), which represents an insertion of a new node n29 contains an If statement to be the 0th child of the node n10. Similarly, the edit action (Update (n18," path")) represents updating the value of node n18 with a string literal of the value "path".



```
string foo (int i) {
   for (int j= 0; j < i; j++)
   {
   }
   return "foo";}
string bar () {
   return "bar";}
```

(a)C++ code fragment in its initial version.

```
string foo (int i) {
   for (int j = 0; j < i; j++) {
      if (j == 5)
         return bar ();
   }
   return "foo";}
string bar () {
   return "path"; }
```

(b) C++ code fragment in its modified version.

```
Insert (n29, n10, 0, IfStatement)
Insert (n30, n29, 0, InfixExpression, ==)
Insert (n33, n29, 1, ReturnStatement)
Insert (n31, n30, 0, SimpleName, j)
Insert (n32, n30, 1, NumberLiteral, 5)
Insert (n34, n33, 0, MethodInvocation)
Insert (n35, n34, 0, SimpleName, bar)
Update(n18,"path")
```

(c) Generated edit script by the GUMTREE tool.

Figure 2. A change example of a C++ code fragment and the resulting GUMTREE's edit script

However, the edit scripts that are generated by the GUMTREE tool are at too low granularity, where the edit actions computed by this tool are at the level of nodes (e.g., too fine-grained code differences) instead of expressions [31]. But, the representation of code changes using the edit actions at the level of nodes makes

the follow-up analysis and the understanding of maintenance tasks difficult. For instance, developers will focus on the insertion of a variable declaration statement rather than considering the too fine-grained tree operations associated with that insertion. Therefore, it is necessary to consider such related too fine-grained tree operations as a whole in the follow-up analysis tasks. Moreover, the set of related fine-grained edit actions, which describe one high-level change (e.g., method declaration statement), usually would be scattered through the edit script computed by the GUMTREE tool [32]. Therefore, a set of preprocessing steps are needed to generate more understandable code differences from each edit script generated by the GUMTREE tool. Here, the primary idea is to group together all of the fine-grained code differences that are related to a high-level AST element or belong to the same connected component into one cluster, where those related differences might be scattered across the script. Accordingly, we have developed our own tool that traverse the tree actions of an edit script by performing two phases as follows:

a. Phase 1: Summarizing Code Differences.

Particularly, this phase is specific for the insertion and deletion actions within an edit script. Mostly, an insert or a delete action on a declaration or a statement AST node is supplemented by several insertion or deletion actions on composing elements of that AST node. Therefore, as in [32], we differentiate between two types of insertion/deletion actions, as follows:

Inserting/deleting of a *base* node: A base node is recognized as a node involves a high-level AST element (declaration or statement). For instance, we consider nodes that contain either if statement, for statement, return statement, method declaration, variable declaration, or attribute declaration as base nodes. For example, in Figure 2

– , nodes n29, n33, n34, n36, and n38 are considered as base nodes.
– Inserting/deleting of a *composite* node: A composite node is a node that is inserted/deleted as a consequence of an inserting/deleting of a base node. That is, a composite node does not hold statements or declarations. For example, in Figure 2, the node n30 is considered as a composite node since it contains an infix expression. Also, a composite node could be a child of a base node or another composite node. In, the composite node n30 is a child of the base node n29, while the composite node n31 is a child of the composite node n30.

In this phase, since we are only interested on insertion and deletion actions, we have implemented a partial of the Generating Concise Code Differences step of the CLDIFF tool that is proposed in [32]. That is, instead of carrying out the whole concise step of the CLDIFF tool, we have only performed the partial that focuses on grouping an insert/delete action of a base node with the insertion/deletion actions of the composing nodes of that base node. Hence, we parse all the insertion and deletion actions in an edit script. Firstly, for an insertion action of the base node A (e.g., Insert (A, PN, I, L, V)), we traverse A's child nodes in a depth-first way while distinguishing between the base and composing child nodes of A. For each traversed child composite node C, if C is a newly-added by the insertion action on A, we group A to C and continue the traversal on C's child nodes. On the other hand, for each traversed child base node B, we stop traversal on B's child nodes, but continue the traversal on the other child nodes of A. At the end, we replace the insertion of A and all the insertion actions of those composite nodes that were grouped with A by the concise code difference Insert (A, PN, I, L, V)). Then, we perform the same concise step mentioned above for each deletion action of a base node. For example, when traversing the edit script shown in Figure 2, we consider (Insert(n29, n10, 0, IfStatement)) as an insertion of a newly-added base node , and we group it with its descendent insertions of newly-added composite nodes: (Insert(n30, n29, 0, InfixExpression, ==), Insert(n31, n30, 0, SimpleName, j), and Insert(n32, n30, 1, NumberLiteral, 5)). Figure 3 shows the output of this phase when applying it on the edit script given in Figure 2.

b. Phase 2: Textual Representing of Concise Code Differences.

Here, we aim at representing the concise code differences in a manner helps extracting the essence of the changes from those inserted, moved, updated, and deleted nodes, and consequently understand the syntactic types of the undertaken changes. Therefore, we have textually represented each concise edit action in a manner similar to the textual representation of edit actions performed in [30]. The textual representation of an AST concise edit action is a 3-value tuple: (T, E, PE), where T is the change type, E is the code entity correlated to the change, and PE is the parent code entity where the change occurs. In our work, the change type (T) is one of those change types that are defined in [33], which are specific to changes of object-oriented code, such as Final Modifier Insert, Statement Insert, Class Renaming, Method Renaming, Parent Class Update, Parameter Type Change, and Return Type Update. Moreover, since adaptive maintenance is in response to changes in the API-method invocations, we make use of additional change types, namely Argument Insert, Argument Delete, and Argument Update. Those additional change types are also used by the GUMTREE tool. For instance, the 3-value tuple of the concise edit action (Insert (n33, n29, 1, ReturnStatement)), shown in Figure 3, would be: (Statement Insert, Return, If).

In the case of a block statement as a parent entity (PE) in the 3-value tuple of a concise edit action, we replace it by its 'non-block' immediate parent. For example, in Figure 3, the 3-value tuple of the concise edit action Insert (n29, n10, 0, IfStatement) would be (Statement Insert, If, block). Hence, we replace the node n10 (e.g., block) by its immediate parent n6, see Figure 2, and as a result the textual representation would be (Statement Insert, If, For).

At the end of the aforementioned phases, our tool produces a new edit script from the one that has been generated by the GUMTREE tool. In the rest of the paper, we call this new script as Concise Edit Script. Figure 4 shows the concise script associated with the AST edit actions given in Figure 3. In this study, to perform our examination of the frequent code changes relevant to the adaptive maintenance, we formulate a code change by a commit R as in the below definition and thus each concise edit action in Figure 4 represents a single code change.

– Definition 1: A code change by the commit R is represented as a concise AST code difference in the concise edit script of R.

2.3. Computing repetitiveness of AST code changes

We want to examine the repetitiveness of adaptive code changes across the histories of different open-source systems, where we consider the Defination1 to model the code changes performed by each commit under the consideration. The repetitiveness of a code change is computed as the ratio between the number of occurrences of that change over the total number of studied changes. The goal of such examination is to learn what types of AST code differences that are frequently occurring in adaptive commits. To achieve our goal, we need studying the repetitiveness of a code change in two scenarios namely within-system and across-systems. In within-system setting, we aim at mining the repetitiveness of a code change within the history of a specific system, while in the other setting we check the repetitiveness through the histories of different systems. Thus, as in [24], we define a repeated adaptive change within the history of a specific system as follows:

– Definition 2: A code change by the adaptive commit R of the open-source system S is a repeated change if it matches another adaptive code change occurring in an earlier adaptive commit of S.

```
Insert(n29, n10, 0, IfStatement)
Insert(n33, n29, 1, ReturnStatement)
Insert(n34, n33, 0, MethodInvocation)
Update(n18,"path")
```

Figure 3. The edit script associated with code changes of Figure 2 at the end of phase1

```
(Statement Insert, If Statement, For)
(Statement Insert, Return Statement, If)
(Statement Insert, Method Invocation, Return Statement)
(Statement Update, StringLiteral, Return Statement)
```

Figure 4. The concise edit script associated with code changes of Figure 2

On the other hand, we compute the repetitiveness of a code change across the histories of different systems using the following definition:

– Definition 3: A code change by the adaptive commit R of the open-source system S is a repeated change if it matches another adaptive code change occurring in an earlier adaptive commit of S or other open-source systems.

If adaptive changes repeat commonly in the within-system setting, then the repeated changes of a system would be good examples for the example-based construction of the specific transformation rules for that system. If changes are frequent in the across-systems setting, then those repeated changes would be useful examples for the generic transformation rules to be used for different systems. To check code changes for repetitiveness, we have built a Change Dictionary (CD) from the histories of different open-source systems. The dictionary is a set of pairs of the form (C, Rep(C)) where C is a code change and Rep(C) is the repetitiveness of C across the examined systems. Each code change C is stored in the dictionary as a structure with the 3-value tuple: (T, E, PE), as we discussed previously. Figure 5 shows the algorithm that constructs the change dictionary. The algorithm is simple and works much like how developers would compute

change repetitiveness in their head. The algorithm traces all the commits under the examination (line 2). Then, for each commit, the algorithm traces all the source files touched by the commit, and generates the concise edit script associated with the changes performed over each touched source file (lines 3-5). Next, for each concise code change belongs to the generated concise script, the total number of changes is incremented (line 7) and then if the change is encountered for the first time, the algorithm inserts the change with a counter value of 1 into the Occurrence Dictionary OC (line 9). Otherwise, the count value for the change is incremented (line 11). Finally, the repetitiveness of a code change is computed as we discussed early (lines 16-19) and values are stored in the Change Dictionary. The same algorithm is used with the two settings. In within-system setting, we follow Definition 2 and hence the SourceList includes only one open-source system and the *CommiList* is a list of all collected adaptive commits of that system. With the other setting, the *SourceList* contains all the examined open-source systems and so *CommiList* is a list of all collected adaptive commits across the studied systems.

3. RESULTS AND ANALYSIS

We have applied our examination methodology on the all adaptive commits that were collected from the history repositories of the examined systems. Here, we assume that each adaptive commit only contains adaptive changes (e.g., no changes relevant to other types of maintenance actions). This assumption represents one of the threats to validity of our study, as we will discuss later. Afterward, we have extracted the concise edit script associated with each considered commit. We observed in total 11281 concise edit actions. This dataset of code change was used to answer our research questions at different levels of granularity such as commit and change hunk. Table 2 reports our change dataset that is used in this study. Below, we present the details behind our obtained results and answering the research questions posed before.

```
Input: SL - SourceList: a list of studied open source systems
Output: CD – Change Repetitiveness Dictionary – a table holds repetitiveness for all concise code changes, elements are the pairs (Ci,
Rep(Ci))), where Ci is a 3-value tuple: (T, E, PE) and Rep(Ci) is the repetitiveness of Ci
For each system Si ∈ SL, BuildChangeRepetitivenessDictionary (Si, OC, CD) for all systems Si ∈ SL.
algorithm BuildChangeRepetitivenessDictionary (System Si, Dictionary OC, Dictionary CD)
1.    Begin:
2.    For each Commit R ∈ CommiList (Si)
3.      For each SourceFile F touched by R
4.        // compute the concise edit script related to F and store them in a script called ConciseScript
5.        ConciseScript <- ComputeConciseEditScript (F);
6.        For each Change Ci in ConciseScript
7.            NumCi++ // increment total number of examined changes
8.          if OC not contain Ci then
9.            Insert (Ci,1) into OC // set count of Ci to 1
10.         else
11.           OC[Ci].Count++ //increment the count of change Ci
12.         end if
13.       end for
14.     end for
15.   end for
16.   For each Change Ci in OC
17.       Insert (Ci,0) into CD // set repetitiveness of Ci to 0
18.       CD[Ci].Rep = OC[Ci].Count / NumCi // compute the repetitiveness of Ci
19.   end for
      end
```

Figure 5. Algorithm to compute repetitiveness for each change across examined systems

Table 2. Collected concise code changes from examined systems

System	Total Number of Concise Code Changes
Kexi	4991
OSG	3569
JasperReports	2721
Total	11281

3.1. RQ1: What are the most common types of AST code differences that are tightly correlated with adaptive maintenance tasks?

To answer this research question, we applied the repetitiveness computation algorithm, shown in Figure 5, with the across-systems setting (e.g., Definition 3) and at the granularity of source files, where we generated a separate concise edit script for each touched file. Then, we ranked the captured concise AST code changes in the constructed change dictionary using their repetitiveness. The repetitiveness of a concise AST

change is computed as the ratio of occurrence count of that change across all systems over the total number of concise AST changes of all types in all systems (we did not compute individually for each system). Table 3 presents the top concise AST code changes with their repetitiveness across the three studied systems. In Table 3, we only reported the concise changes that have a repetitiveness greater than 1.00%. From Table 3, our first finding is that there are a few numbers of concise AST code changes that represent the majority of the undertaken adaptive changes. For instance, 76.93% of the examined adaptive changes fit into only 12 concise AST changes. The main explanation of this finding is that the adaptive maintenance is in response to changes in the feature/interface of underlying API. For instance, the class QML in Qt4 is now available under the name QtQuick in Qt5, and hence specific statements that include a usage of the class QML must be identified and changed, which usually are in the form of include, instance decelerations, and method invocation statements.

Our second finding is that the updating of a statement that contains creation of a class instance (e.g., new operator) is the most popular AST change with a ratio of 16.51% among the undertaken adaptive changes (the explanation of this was posted above). Additionally, the updating of system method interfaces by using different parameter types repeats much less frequently (e.g., 1.08%) when compared to other top concise AST code changes. Previous work in [24] illustrated that changes to constructor calls (instance creation) have less repetitiveness when compared with other non-fixing changes. However, with the further focusing on adaptive maintenance, our results show that constructor calls have the most changes. Thus, we conclude that the repetitiveness of changes varies according to maintenance types, and hence meaning that our investigation is needed to draw conclusions regarding the qualified code examples for the construction of transformational rules.

Table 3. Top concise AST code changes relevant to adaptive changes across kexi, OSG, and jasper reports systems

AST Code Change	Change Description	Repetitiveness
(Statement Update, Simple Type, Class Instance Creation)	Updating of a statement that contains creation of a class instance (e.g., new operator) by calling different constructor or change the type of the instance.	1862 (16.51%)
(Statement Update, Simple Name, Include) or (Statement Update, Simple Name, Import)	Changing of an include/import statement since the included library/module has been renamed, replaced, or moved to another package. Note that import is specific to Java files and include is related to C++ files.	1525 (13.52%)
(Argument Type Change, Argument List, Method Invocation)	A change of a method invocation statement by using a different argument types, since of an updating of method interface or calling an overloaded method.	1114 (9.88%)
(Method Renaming, Method Invocation, For)	A change of a method invocation statement by calling another member method of a class instance within a For block, since of a method being renamed or replaced.	1059 (9.39%)
(Statement Insert, If, For)	An insert of a new If statement within a for block.	997 (8.84%)
(Argument Insert, Argument List, Method Invocation)	A change of a method invocation statement by using a different number of argument, since of an updating of method interface or calling an overloaded method.	622 (5.52%)
(Statement Insert, Method Invocation, If)	An insert of a method invocation statement within an If block.	490 (4.35%)
(Statement Update, Super Constructor Invocation, Method)	A change of a super constructor invocation statement	277 (2.46%)
(Statement Insert, Return, If)	An insert of a return statement within an If block.	254 (2.25%)
(Return Type Change, Simple Type, Method Deceleration)	A change of a method interface (declaration statement) by using a different return type.	196 (1.74%)
(Condition Expression Change, If, For)	A change of the condition expression of an If statement within a For block.	161 (1.43%)
(Parameter Type Change, Parameter List, Method Deceleration)	A change of a method interface (declaration statement) by using different parameter types.	122 (1.08%)
Total		8679 (76.93%)

By further analyzing the top concise AST changes, we found an overlapping between some changes with respect to the statement kinds. For instance, method invocation statements appear in several top AST changes, as shown in Table 3. Thus, we have deepened our analysis of the top AST changes in order to categorize them based on which statement kinds they occur, and so discover which statement kinds being touched by adaptive commits more frequently than others. Table 4 shows the results we obtained, where the repetitiveness frequency of a statement kind is equivalent to the ratio of repetitiveness count of the top AST changes relevant to that kind over the total count of the top concise AST code changes in all systems

(e.g., the total count is 8679 as shown in Table 3). For instance, among the top AST changes, the only change relevant to the return statement is (Statement Insert, Return, If), and thus the count of this statement kind is 254 with a frequency of 2.93% (254 / 8679). In our analysis, we consider the statement of calling a super constructor as a method invocation statement, while the calling of a constructor using a new operator was considered as a modification to an instance creation statement.

Hence, our results provide general features of adaptive changes and thus can be leveraged by automatic transformation rule generation algorithms to prioritize some kinds of statements (e.g., include and method invocation) and AST changes. To be precise, our results recommend that in order to push the degree of correctness of transformations, developers need to focus on the top AST code changes, shown in Table 3, to collect real qualified examples that will be used in conjunction with a syntactic differencing tool to generate transformations that can be applied to other system that yet need to undergo the same adaptive change. At this point further investigation is necessary to categorize the collected examples that address the same problem and so generate the transformation necessary to solve a specific adaptive maintenance task. We plan to address this investigation in future.

As shown in Table 4, there is a prevalence of include/import, method invocation, method deceleration, instance deceleration (e.g., new operator), If, and return statements when compared to the others. Moreover, our observation is that 41.04% of the top AST changes are relevant to method invocation statements, and hence this kind of statements represents the most frequently modified statement by adaptive changes, as expected. That is, changes related to method invocation statements are the most popular and frequently repeated statements that touched by adaptive changes across C++ and Java open-source systems. Thus, this finding indicates there is no doubt that existing tools of API usage pattern recommendations (e.g., altering a sequence of method calls required to implement a functionality) like the tools proposed in [19, 27] would be useful in extracting the code examples for the construction of transformational rules. Nevertheless, because method invocation statements represent only 41.04% of top statements touched by API-migrations, we still lack approaches for automating the extraction of code examples related to other statements (e.g., instance creation and IF statements). Also, we would observe that most of adaptive changes were performed within either for or if blocks. A recent study by Campos et al [20] illustrated that there is a prevalence of IF and return statements when compared to the other bug fixing statements. This result is consistent with our finding. On the other hand, changes to some statement kinds such as catch, try, and switch statements are rarely found in the adaptive commits, despite that the repetitiveness of these statements were founded relatively high (e.g., catch statement had a repetitiveness of 37%) in fixing changes as illustrated by the results of [24].

Table 4. Top statement kinds that touched by the top AST changes of the adaptive commits

Statement Kind	Relevant AST changes	Repetitiveness
Include/import	(Statement update, Simple Name, include) or (Statement update, Simple Name, import)	1525 (17.57%)
Method Invocation	(Method Renaming, Method invocation, For) (Argument Type Change, Argument list, Method invocation) (Statement update, Super constructor invocation, Method) (Statement insert, Method invocation, If) (Argument Insert, Argument List, Method invocation)	3562 (41.04%)
Method Deceleration	(Return Type Change, Simple Type, Method Deceleration) (Parameter Type Change, Parameter list, Method Deceleration)	318 (3.66%)
Instance Creation	(Statement Update, simple type, class instance creation)	1862 (21.45%)
IF	(Statement Insert, If, for) (Condition expression change, If, For)	1158 (13.35%)
Return	(Statement Insert, Return, If)	254 (2.93%)
Total		8679 (100%)

– Observation 1: 76.93% of the adaptive changes fit into only 12 concise AST changes, meaning that developers need to focus on these concise changes to collect real qualified examples to construct high correctness example-based transformation rules.

3.2. RQ2: What are the most common adaptive change patterns?

Here, RQ2 is important to detect common change patterns appeared in the historical adaptive changes of C++ and Java open-source systems, and so we could observe which patterns are the most correlated with API-migration tasks. Moreover, although the change patterns proposed in the literature are relevant with non-adaptive maintenance tasks (e.g., bug-fix and refactoring tasks), RQ2 is interesting by investigating which of these previously proposed patterns are also strongly related with API-migrations. Answering RQ2 has two

main components. The first is the identification of change patterns that could be suggested as potential patterns for the majority of adaptive changes. The second is the validation of those suggested patterns through evaluating their repetitiveness in the change history of other arbitrary open source systems that were not used in the identification step of patterns. To undertake the first component that is outlined above, we surveyed prior works that have focused on studying possible change patterns of software evolution, and then we have leveraged the knowledge from several valuable prior studies [21-23, 34-36], where we exposed numerous known change patterns that were frequently occurred in the evolution histories of open-source systems. Then, we have used the obtained results from RQ1 to narrow our investigation of the possible change patterns that might be strongly relevant with adaptive changes. That is, we have made attentions only to the patterns that are relevant to those top AST changes and statement kinds that were found as most popular in the adaptive commits, such as if-related and method call patterns, while we ignored the patterns relevant to the statements with insignificant repetitiveness. Examples of ignorable patterns include changes to catch, try, switch, and else statements. In particular, based on the results of RQ1, we can hypothesize that a large portion of adaptive changes are instances of the following change patterns, which were originally proposed in the literature, as follows:

- Method Call with Different Number of Parameters or Different Types of Parameters (MC-DNP): A change of a method invocation statement by calling the same method but with a different number or a different type of arguments, which is because of an updating of the method interface or calling an overloaded method. This pattern was proposed for bug fixing changes by Pan et al. [21].
- Addition of Precondition Method Invocation (IF-MC-ADD): A change that adds a new IF block that involves a method invocation statement. This pattern was proposed for bug fixing changes by Martinez et.al [34].
- Change of Method Deceleration (MD-CHG): A change of the declared interface for a method by using a different return type or different number or types of parameters. This pattern was proposed as a signature change pattern by Kim et.al [23]. Also, this pattern was considered as a bug-fix pattern by Pan et al. [21].
- Change of Method Call to a Class Instance (MC-DM): A change that calls a different member method of the same object variable. For example, this pattern is because of a method being replaced or renamed by a new method in the new API release. The pattern was introduced by Pan et al. [21] and also it was suggested as a refactoring pattern by Tsantalis et al. [22].
- Add Precondition Check with Jump (IF-APCJ): A change that adds a new if block that involves a jump statement. In our study, we only focus on return as a jump statement. This pattern was considered as bug-fix pattern by Pan et al. [21].
- Update of Super Constructor Invocation (CONS-UP): A change that modifies the invocation of a super class constructor. This change is related to a change in the class deceleration by updating the super class of a given sub class. This pattern was proposed Fluri et.al [35] as a general change pattern and also by Martinez et.al [34] as a bug fix pattern.
- Initializing an Object (IAO): A class instance initialization statement, which involves the operator new, is changed because of a calling of different constructor and/or changing the type of a declared object. This pattern is proposed by Soto et.al [36].
- Change of If Condition Expression (IF-CC): A change that modifies the expression of an If condition statement. This pattern was considered as bug-fix pattern by Pan et al. [21].

On the other hand, since the results of RQ1 show that the changes to include/import statements have significant repetitiveness in adaptive commits, we suggest a new meaningful change pattern that would be a potential pattern for adaptive changes. The suggested pattern is as follows:

- Change of Include/Import Statements (IS-CHG): The API-migration changes the affect the include/import statements of API libraries, classes, and interfaces. This kind of changes generally leads to changes at call sites to API features.

Our hypothesis is that the patterns aforementioned are good candidates for API-migration change patterns. To validate this hypothesis, as in [21], we need to explore the *hunk coverage*, which represents the percentage of the adaptive change hunks that contain at least one pattern from among the suggested 9 patterns posted above. Also, it is important to know if the frequencies of the suggested patterns are similar across many open-source systems. If the frequencies are similar, this would provide an understanding of the most common types of adaptive changes occurring in software systems. As such, the second component of RQ2 would be the evaluation of the repetitiveness of the change patterns under consideration in the historical API-migration changes at the granularity of hunks. Since we have based on the dataset of Kexi, OSG, and JasperReports systems in our abovementioned hypothesis, this dataset is still not enough to substantively answer RQ2 and hence validating our hypothesis. Accordingly, in addition to the early used dataset, we now need to extend our dataset by new adaptive changes from arbitrary C++ and Java open source systems other

than those used in answering RQ1. We have collected the additional dataset of adaptive changes from the change history of the quantitative finance library QuantLib, the KDE editor Kate, and the task editing of the task-focused interface for Eclipse Mylyn Tasks. These selected systems have successful API-migration tasks within a long evolutionary history [12, 37]. For instance, the API-migrations of QuantLib and Kate systems have been studied in [12], while the migration to newer eclipse versions of Mylyn project was investigated in [37]. We manually recognized the adaptive commits of the new selected systems in the same manner that was performed by Meqdadi. et.al [26].

Table 5 summarizes these new selected systems and their relevant change dataset. Our experiments begin with creating a concise edit script for each change hunk from the history of the six examined systems. As shown in Table 1, Table 2, and Table 5 our new dataset has in total 6380 change hunks (e.g., concise edit script) and 15144 concise code changes. Next, to validate our hypothesis, we need to search for the instances of the abovementioned 9 patterns in the collected 6380 concise edit scripts and then find the percentage of the adaptive change hunks that match these 9 patterns.

Automatic searching for instances of change patterns within change hunks shows great promise. For instance, Martinez et.al [30, 34] proposed a novel algorithm for searching change pattern instances from a change history using AST differences. The algorithm aims at deciding whether or not a particular change pattern is existing inside a given change hunk. That is, the algorithm accepts two inputs: an AST representation of a specific change pattern and the set of change hunks. The output of the algorithm is the counting of instances of the pattern in the inputted set of change hunks. Details of the algorithm and its phases are carefully clarified in [34]. In our work, we have implemented and applied the algorithm that is proposed by Martinez et. al [30, 34]. Also, we have followed their definition of change patterns that aims at representing a specific pattern at the AST level. In this introduced AST representation, a change pattern is characterized with a structure consists of three components: a list of potential micro-patterns (MP), a relation map (RM), and a list of undesired changes (UC). However, our considered 9 patterns have no UC component [34]. Thus, in this work, we have omitted this component from our representation of change patterns at the AST level. Below, we will discuss the MP and RM components.

Firstly, every change pattern involves a list of micro-patterns that are associated with it [30]. A micro-change pattern is an abstraction over concise AST changes. That is, a micro-pattern would with a tuple (T, E, PE), where the field PE is not a mandatory and thus could take any value (e.g., a wildcard character "*"). Therefore, each concise code change would be simply recognized as an instance of a specific micro-change pattern. For instance, the concise code change (Condition expression change, If, For) is identified as an instance of the micro-change pattern (Condition expression change, If, *). The list of micro-patterns of a change pattern is an ordered list according to their position inside the source code. That is, the pattern formed by the concise change M1 followed by the concise change M2 is not equivalent to the pattern formed by M2 followed by M1. As an example, the list of micro-patterns that relevant to the change pattern "Addition of Precondition Method Invocation (IF-MC-ADD)", proposed in [34], consist of: M1= (Statement insert, If, Method) and M2= (Statement insert, Method invocation, If), and thus PE component is a mandatory in M1 and M2. Secondly, the component RM of a specific change pattern represents a set of relations between the changes of the entities (E) involved in the potential MPs of that change pattern. For example, the pattern IF-MC-ADD needs the entity method invocation (component E of M2) to be enclosed by an If, which is affected by the change M1. That is, M2.PE == M1.E. Table 6 illustrates the AST representations of those 9 change patterns that were investigated in our experiments.

Table 5. New selected systems for the experiments relevant to RQ2

	QuantLib	Kate	Mylyn Tasks
Language	C++	C++	Java
Adaptive Maintenance Task	Migration to Visual C++ 2017	Migrating to Qt5.x	Migration to Eclipse 4.x
Time Period	(1/1/2017–1/1/2018)	1/1/2015 –1/1/2018	(1/1/2013–12/31/2017)
# Commits in the Log File	628	922	519
# Adaptive Commits	59 (9.4%)	54 (5.9%)	42 (8.1%)
# Adaptive Change Files	206	186	154
# Adaptive Change Hunks	550	571	522
# Concise Code Changes	1298	1369	1196

Addressing RQ2 involves applying the formerly discussed algorithm over the concise edit scripts of each studied system separately in order to search of how many instances of each pattern were observed over the examination period of that system, and so we would compute the frequencies of those 9 patterns for each examined system. The frequency of a pattern is computed by taking the number of hunks identified as instances of the pattern, and dividing it by the total number of change hunks detected for that system. Recall that a change

hunk could represent an instance of more than one change pattern. For example, a hunk could involve updating a method call with different number of parameters and also changing the condition expression of a particular if statement, and so this hunk is an instance of MC-DNP and IF-CC patterns. On the other hand, if there are more than one instances of a particular change pattern in a change hunk, we count the hunk only once as instance of that pattern. For instance, if a hunk contains an updating of an include statement two times, we count it as only one instance of IS-CHG pattern, similarly if a hunk involves parameter addition change three times, we count it as only one instance of MC-DNP pattern.

Table 7 presents our obtained results with respect to the hunk coverage. The first finding is that our suggested patterns cover approximately 64.69% to 76.58% of all studied change hunks. This result provides an indication for the effectiveness of our suggested patterns in classifying significant ratio of the undertaken adaptive change hunks, and thus these patterns are the most tightly correlated patterns with API-migration tasks. Moreover, when relating our findings with the results identified in [21], the hunk coverage is higher with adaptive changes compared with to the bug fixing. Therefore, we would conclude that the adaptive change hunks have no random code alterations, and thus responsive to be performed by automatic transformations. Thus, among the 27 patterns that were proposed by Pan et al. [21], there are only 5 patterns that were efficient in representing the adaptive changes. Other interesting results are obtained from the occurrence frequencies for each suggested pattern across the examined systems. Figure 6 shows the computed frequency of every considered pattern, where the frequency of a pattern in a system is computed as the ratio of how many adaptive hunks classified as instances of this pattern over the total number of adaptive hunks in that system. Hence, we would observe that the IAO and method call patterns (MC-DM and MC-DNP) are the most prevalent adaptive change patterns across all of the studied systems. For instance, together they cover for 33.33% to 59.01% of the all adaptive change hunks. We would naively expect this result, since open source systems accessing their APIs through instantiating objects or via the invocation of API methods, and hence migrating to a new API release causes making new objects to be passed to new API methods and handling changes that surrounds the calling of updated, deprecated, or overridden API methods. Moreover, as in [21], we applied the Pearson's correlations [38] between the frequencies of our considered patterns across the examined systems. Our finding is that the occurrence frequencies of the suggested adaptive change patterns tend to be similar across the examined open-source systems, even though examined systems have different underlying APIs (e.g., Qt, OpenGL, Visual C++, Eclipse, and JFreeChart).

Table 6. AST representations of potential adaptive change patterns

Pattern Name	AST Representation	
	Micro-Patterns (MP)	Relational Map (RM)
IS-CHG	M1 = (Statement update, Simple Name, include)	
(2 subclasses)	M1 = (Statement update, Simple Name, import)	
MC-DNP	M1 = (Argument Insert, Argument List, Method invocation)	
(2 subclasses)	M1 = (Argument Type Change, Argument list, Method invocation)	
MC-DM	M1 = (Method Renaming, Method invocation, *)	
IF-MC-ADD	M1 = (Statement insert, If, *) M2 = (Statement insert, Method invocation, If)	M2.PE == M1. E
IF-APCJ	M1 = (Statement Insert, If, *) M2 = (Statement Insert, Return, If)	M2.PE == M1. E
MD-CHG (6 subclasses)	M1 = (Parameter Type Change, Parameter list, Method Deceleration) M1 = (Return Type Change, Simple Type, Method Deceleration) M1 = (Return Type Change, Primitive Type, Method Deceleration) M1 = (Parameter Insert, Parameter list, Method Deceleration) M1 = (Parameter Delete, Parameter list, Method Deceleration) M1 = (Parameter Ordering Change, Parameter list, Method Deceleration)	
IF-CC	M1 = (Condition expression change, If, *)	
IAO	M1 = (Statement Update, simple type, class instance creation)	
CONS-UP	M1= (Statement update, Super constructor invocation, Method)	

Table 7. Hunk coverage in the examined systems

System	Hunk Coverage
Kexi	76.58%
OSG	64.69%
JasperReports	74.01%
QuantLib	70.03%
Kate	75.76%
Mylyn Tasks	68.87%

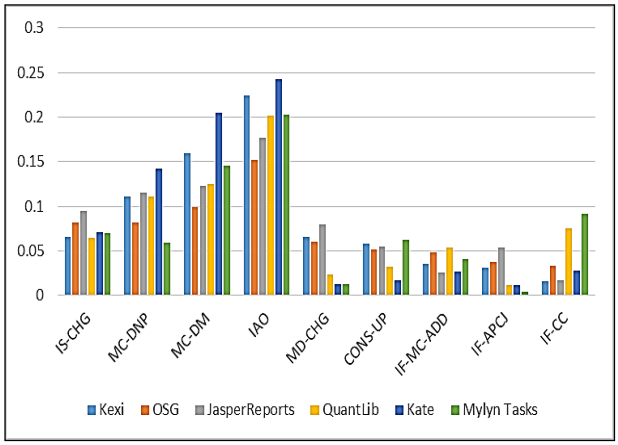


Figure 6. The frequencies of suggested patterns in adaptive change hunks for the examined systems

– Observation 2: The obtained results clearly demonstrate that current automatic API-migration approaches would be improved by more focusing on the 9 repetitive patterns that have covered 64.69% to 76.58% of adaptive change hunks.

4. RELATED WORKS

Our research is correlated with the studies on the analyzing of code change patterns using AST approaches. With respect to AST Differencing problem, Falleri et al. [25] addressed the limitations of ChangeDistiller and developed GUMTREE tool that works on ASTs at short times and with the support of wide range of programming languages such as C++. Huang et al. [32] extend GUMTREE tool by implementing a code differencing approach, named CLDiff that aims at generating concise linked code differences that more easily be understandable by developers. Fluri and Gall [33] proposed an Eclipse plugin ChangeDistiller tool to generate a tree edit script from two coarse grained ASTs. The tool is built on an analysis of basic tree edit operations (e.g., insert, delete, update, and move). However, ChangeDistiller cannot generate fine-grained scripts on programming languages that have a lot of composite elements in statements (such as C++). Jiang et al. [39] proposed an approach to aggregate relevant code changes that were committed through the history from version control systems using change operations and locations.

The area of automatic API-migrations by learning patterns is a main interest in several researches in the literature [27, 37, 40]. For instance, SemDiff was proposed in [37] as a recommendation system that guides adaptations to client programs by examining how a framework adapts to its own changes. When applied on three client programs that use Eclipse JDT framework, SemDiff recommended necessary adaptive changes with a significant precision. Bregmann et al. [40] introduced change driven transformational models to automate the usage of a programming language for different change scenarios. However, the main limitation of these two studies is that they recommend adaptive changes that only associated with framework method invocations.

5. CONCLUSION AND FUTURE WORK

Previous researches were interested in change patterns correlated with non-adaptive maintenance tasks such as bug fixing and refactoring changes. This work explored the repetitiveness of adaptive code changes (in the context of API-migrations) mined from the histories of several C++ and Java open-source systems, based on the analysis of AST differences undertook by a set of adaptive commits. We found that that the repetitiveness of adaptive changes would be very high across systems, and thus the mining of popular adaptive changes that previously made from the history of a system and then using them as real examples to guide the future adaptive maintenance tasks of other systems, via the example-based transformation rules, is very useful approach. Additionally, the results show that only 12 concise AST code differences covered about 76.93% of the examined adaptive changes of the Kexi, OSG, and JasperReports systems, where the updates of class instance initialization statements represented the most common concise AST change among the undertaken adaptive changes.

Next, we used the obtained results to suggest 9 change patterns that would strongly be adaptive change patterns such as Initializing an Object, Method Call with Different Number of Parameters or Different Types of Parameters, and Addition of Precondition Method Invocation patterns. Our validation results show that 64.69% to 76.58% of the adaptive change hunks were covered by our suggested patterns. Also, we observed that the most common categories of change patterns in adaptive hunks are related to initializing objects and method calls. It is future work to take into account the outcomes of this study and setting up reasonable experiments to assess whether suggested change patterns are valid with API-migrations of other systems from different domains such as commercial systems or those systems written in programming languages other than C++ and Java.

REFERENCES

[1] J. Singh, *et al*., "Reengineering Framework for Open Source Software using Decision Tree Approach," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 3, pp. 2041–2048, 2019.

[2] D. Kim, "Enhancing Code Clone Detection using Control Flow Graphs," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 3804–3812, 2019.

[3] Z. Chen, *et al*., "Clone Refactoring Inspection by Summarizing Clone Refactorings and Detecting Inconsistent Changes During Software Evolution," *Journal of Software Evolution and Process*, vol. 30, no. 10, pp. 1-24, 2018.

[4] G. Özdağoğlu and E. Kavuncubaşı, "Monitoring the Software Bug-Fixing Process Through the Process Mining Approach," *Journal of Software Evolution and Process*, vol. 31, no. 7, pp. 1-11, 2019.

[5] D. Kim, "Finding Bad Code Smells with Neural Network Model," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 7, no. 6, pp. 3613–3621, 2017.

[6] Mazrekaj, *et al*., "An Overview of Virtual Machine Live Migration Techniques," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 4433–4440, 2019.

[7] N. Chandrakala and B. Thirumala Rao, "Migration of Virtual Machine to improve the Security of Cloud Computing," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 8, no. 1, pp. 210–219, 2018.

[8] S. Schach, *et al*., "Determining the Distribution of Maintenance Categories: Survey versus Measurement," *Empirical Software Engineering*, vol. 8, no. 4, pp. 351-365, 2003.

[9] E. Swanson, "The dimensions of Maintenance," *2nd International Conference on Software Engineering (ICSE 1976)*, pp. 492-497, 1976.

[10] D. Danny and J. Ralph, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 83–107, 2006.

[11] H. Alrubaye and M. Wiem, "Variability in Library," *Software Engineering for Variability Intensive Systems: Foundations and Applications*, pp. 295, 2019.

[12] O. Meqdadi, *et al*., "Mining Software Repositories for Adaptive Change Commits Using Machine Learning Techniques," *Information and Software Technology Journal*, vol. 109, pp. 80-91, 2019.

[13] C. Newman, *et al*., "Simplifying the Construction of Source Code Transformations via Automatic Syntactic Restructurings," *Journal of Software Evolution and Process*, vol. 29, no. 4, pp. 1-18, 2017.

[14] M. L. Collard, *et al*., "A lightweight transformational approach to support large scale adaptive changes," *IEEE International Conference on Software Maintenance (ICSM '10)*, pp. 1-10, 2010.

[15] R. Rolim *et al*., "Learning Syntactic Program Transformations from Examples," *39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 404-415, 2017.

[16] N. Meng, *et al*., "Lascad: Language-agnostic software categorization and similar application detection," *Journal of Systems and Software*, vol. 122, pp. 21-34, 2018.

[17] N. Meng, *et al*., "Systematic Editing: Generating Program Transformations from an Example," *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 329-342, 2011.

[18] H. A. Nguyen, *et al*., "Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns," *41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 819-830, 2019.

[19] H. Niu, *et al*., "API usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, no. C, pp. 127–139, 2017.

[20] E. Campos and M. Maia, "Discovering common bug-fix patterns: A large-scale observational study," *Journal of Software: Evolution and Process*, vol. 31, no 7, pp. 1-28, 2019.

[21] K. Pan, *et al*., "Toward an Understanding of Bug Fix Patterns," *Empirical Software Engineering*, vol. 14, no 3, pp. 286–315, 2008.

[22] N. Tsantalis, *et al*., "Accurate and Efficient Refactoring Detection in Commit History," *40th IEEE/ACM International Conference on Software Engineering (ICSE 2018)*, pp. 483-494, 2018.

[23] S. Kim, *et al*., "Analysis of Signature Change Patterns," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1-5, 2005.

[24] H. A. Nguyen, *et al*., "A study of repetitiveness of code changes in software evolution," *28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, pp. 180-190, 2013.

[25] J.R. Falleri, *et al*., "Fine-grained and accurate source code differencing," *29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, pp. 313–324, 2014.

[26] O. Meqdadi, *et al*., "Towards Understanding Large-Scale Adaptive Changes from Version Histories," *IEEE International Conference on Software Maintenance (ICSM 2013)*, pp. 416-419, 2013.

[27] H.A. Nguyen, *et al*., "A Graph-Based Approach to API Usage Adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302-321, 2010.
[28] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," *20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pp. 210-219, 2004.
[29] N. Palix, *et al*., "Improving pattern tracking with a language-aware tree differencing algorithm," *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, pp. 43-52, 2015.
[30] M. Martinez, *et al*., "Automatically Extracting Instances of Code Change Patterns with AST Analysis," *IEEE International Conference on Software Maintenance (ICSM 2013)*, pp. 388-391, 2013.
[31] R. Reudismam, *et al*., "Learning Quick Fixes from Code Repositorie, " in *arXiv preprint arXiv:1803.03806*, 2018.
[32] K. Huang, *et al*., "ClDiff: Generating Concise Linked Code Differences," *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, pp. 679-690, 2018.
[33] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," *14th IEEE International Conference on Program Comprehension (ICPC 2006)*, pp. 35-45, 2006.
[34] M. Martinez, *et al*., "Accurate Extraction of Bug Fix Pattern Occurrences using Abstract Syntax Tree Analysis," *Technical Report*, hal-01075938, Inria., 2014.
[35] B. Fluri, *et al*., "Discovering Patterns of Change Types," *23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 463-466, 2008.
[36] M. Soto, *et al*., "A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions," *13th IEEE/ACM Working Conference on Mining Software Repositories (MSR 2016)*, pp. 512-515, 2016.
[37] B. Dagenais and M. Robillard, "Recommending adaptive changes for framework evolution," *30th ACM/IEEE International Conference on Software Engineering*, pp. 481-490, 2008.
[38] R. E. Courtney and D. A. Gustafson, "Shotgun correlations in software measures," *Software Engineering Journal*, vol. 8, no. 1, pp. 5-13, 1993.
[39] Q. Jiang *et al*., "Understanding Systematic and Collaborative Code Changes by Mining Evolutionary Trajectory Patterns," *Journal of Software Evolution and Process*, vol. 29, no. 3, pp. 1-22, 2017.
[40] G. Bergmann, *et al*., "Change-Driven Model Transformations," *Software and System Modeling*, vol. 11, no. 3, pp. 431- 461, 2012.

BIOGRAPHIES OF AUTHORS

Omar Meqdadi is an Assistance Professor in the Department of Software Engineering at Jordan University of Science and Technology, Jordan. His research interests are in software engineering with focus on software evolution. He obtained his PhD in Computer Science from Kent State University, Ohio, USA.

Shadi Aljawarneh is a Professor in the Department of Software Engineering at Jordan University of Science and Technology, Jordan. His research interests are in software engineering with focus on e-learning, cloud computing, bioinformatics, and ICT fields. He holds a PhD in Software Engineering from Northumbria University-England.