A Co-evolutionary Method Between Architecture and Code

Tong Wang*, Bixin Li†,Lingyuan Zhu†
*School of Computer Science and Technology, Anhui University of Technology, Maanshan, China
†School of Computer Science and Engineering, Southeast University, Nanjing, China

***Abstract*—Code evolution and architecture evolution are respectively related to functional requirements and non-functional requirements. According to the type of requirements, architects or developers evolve one of them. That causes the unevolved one is inconsistent with the evolved one. To solve the problem of inconsistency, we propose a co-evolutionary method to keep the consistency between architecture and code. In our method, two evolutional scenarios are considered, including co-evolving code based on evolved architecture and co-evolving architecture based on evolved code. In the former method, we first convert architecture change to code change based on mapping rules, then modify code to implement the corresponding code changes. In the latter method, we first modify the file dependency graph based on mapping rules, then recover architecture based on the modified file dependency graph. We conduct our experiments with eight open source projects, the experimental results indicate that our method can keep the consistency in the two evolutional scenarios, so that, our co-evolutionary method between architecture and code is effective.**

***Index Terms*—Software architecture, code source, co-evolution**

I. Introduction

In the software life cycle, architects and developers frequently evolve code and architecture to keep competitiveness and vitality of software [1]. Code is the actual implementation of software, so developers evolve it for meeting new functional requirements, such as new functions, high performance, and so on. Architecture is the high abstraction view of software, so architects evolve it for meeting new non-functional requirements, such as testability, maintainability [2], and so on [3].

According to the content of new requirements, developers and architects decide which one should be evolved. When developers or architects evolve code or architecture, another one is not consistent with the evolved one. The problem of inconsistent may cause the unevolved one to mislead developers and architects. To solve the problem, many co-evolutionary methods are proposed.

There are mainly four types of evolutionary methods, architecture recovery [4], code automatic generation [5], multi-view software evolution approach [6], and information fusion approach [7]. However, not all the above methods have taken all evolution scenarios into consideration, resulting in the limitation of effectiveness.

For resolving the above problems, we propose a co-evolutionary method to keep the consistency between architecture and code. The contributions of the paper are as follows:

DOI reference number: 10.18293/SEKE2020-151

- The method supports the two-way co-evolution, that is, it supports co-evolving code base on evolved architecture and co-evolving architecture base on evolved code.
- More types of co-evolutionary actions are taken into consideration to improve the effectiveness of co-evolution.
- We implement our method on eight open source projects, the experimental results indicate that our method can keep the consistency between code and architecture effectively.

The paper is organized as follows. Section II introduces the two-way co-evolutionary method. In Section III, we implement our method on open source projects to analyze the effectiveness of our method. Section IV introduces related work. Section V draws the conclusion and introduces the future work.

II. Our Method

Code and architecture belong to different levels of granularity. According to new requirements, one of architecture and code will be evolved first. According to the type of the evolved object, our method is divided into two parts, the method of co-evolving code based on architecture and the method of co-evolving architecture based on code.

A. A Co-evolutionary Method of Code based on Evolved Architecture

Architecture is represented by CDG, and CDG consists of component nodes and dependency edges between nodes, so architecture changes are reflected by component node changes and dependency edge changes [8]. Code changes determine file changes and dependency changes, that is, code changes are reflected by FDG. So the relation between architecture and code can be converted to the relation between CDG and FDG.

To locate which parts of FDG need to be co-evolved, we propose mapping rules to convert CDG changes to FDG changes, then modifying code to implement these FDG changes.

Edge changes of CDG contain the following types: adding an edge, deleting an edge, increasing the weight of the edge, and reducing the weight of the edge.

Adding an edge indicates that some functions are invoked by another component, which can avoid the duplication of functions in multiple components. The corresponding code change is adding a dependency between files contained in the involved components.

Increasing the weight of the edge means that several functions are enhanced, and the corresponding code change is to increase the dependency intensity between the corresponding file sets.

Adding an edge is a special situation of increasing the weight of the edge, that is, the dependency intensity is increased from 0.

Deleting an edge means the relation between components is removed. The corresponding code change is deleting the corresponding dependency between corresponding sets of files.

Reducing the weight of the edge indicates that several functions should be reduced to avoid mutual influence. The corresponding code change is to find the corresponding file sets, then the dependency which has the lowest dependency intensity is deleted.

Deleting an edge is a special situation of reducing the weight of the edge, that is, the dependency intensity is reduced to 0.

According to the above analysis, edge changes are related to dependency intensity. Here, we introduce the definition of dependency point.

Dependency point: If file A has a dependency with file B, and the dependency is caused by that statement a of file A has a dependency with statement b of file B, then statement a and statement b are the dependency points.

We use slice technology detect the direct and indirect effects caused by dependency points. Program slicing is an important technology to analyze programs [9]. The core concern is the slice criterion which is usual a tuple <*s,v*>, herein, *s* means the location and *v* means a variable or a set of variables that is defined or used in *s*. The slice of a program about a criterion is the set of statements that potentially have influenced on *v* of *s* or be affected by *v* of *s* [10]. The dependency point as the slice criterion, then we use slice technology to extract the related statements.

Node changes of CDG contain the following types: adding a component-node, deleting a component node, merging component node, and splitting a component node.

Adding a component node indicates that the software needs to be added new functions where the corresponding code change is adding highly cohesive files. These highly cohesive files refer to increase the set of relevant files rather than the isolated and fragmented files. A new component indicates a new function, however, we cannot get the detail of code based on CDG, so it is implemented by adding a set of files, then the code is developed based on the detail of functions.

Deleting a component node indicates that a function needs to be deleted where the corresponding code change is deleting all files contained in the involved component.

Merging component nodes represent integrating related functions of the software, and the corresponding code change is increasing the dependency intensity between the corresponding sets of files. The higher dependency intensity indicates that the two sets of files can be clustered into a new component.

Splitting a component node indicates that the functions of the component need to be refined. The corresponding code change is splitting the set of files contained in the involved component into two sets.

This operation contains two steps, splitting a set of files into two sets of files, then reducing the dependency intensity between the two sets of files.

We split the set of files based on the loss function. The loss function is a function that maps random events or their related random variables to non-negative real numbers to represent the "risk" or "loss" of the random event. The coupling and cohesion are the important metrics for software, so we define the loss function based on cohesion and coupling as shown in Formula 1.

$$L(\theta, f) = \alpha * D(f) + \beta * S(f) \tag{1}$$

In Formula 1, $L(\theta, f)$ is the loss function, θ represents architecture change, f indicates a splitting scheme, $D(f)$ shows the impact of the splitting scheme on cohesion, $S(f)$ remarks the influence of the splitting scheme on the coupling, and α and β are the weights of cohesion and coupling.

Formula 1 is refined to Formula 2.

$$L(\theta, f) = \alpha * \sum_{p \in N} D_p(f_p) + \beta * \sum_{p,q \in N} S(f)(f_p \neq f_q) \tag{2}$$

In Formula 2, N represents the files contained in the split component. f indicates a splitting scheme for these files N. p and q are two files of N, f_p represents the set to which the split file p belongs, and $D_p(f_p)$ shows the loss resulting from splitting p by f, which is inversely proportional to the cohesion of the divided set of files, and $S(f_p \neq f_p)$ remarks the loss resulting from the different belonging of p and q, which is proportional to the coupling of the new sets after division.

The weight of the dependency edge is proportional to the dependency intensity. When the dependency intensity is low, the weight is low, that is, the related two files can be classified into different sets to obtain the minimal overall loss. The algorithm is shown in Algorithm 1.

B. A Co-evolutionary Method of Architecture based on Evolved Code

Architecture is a high abstraction view of FDG, and FDG reflects the attributes of files and the dependencies between files. So, the impact of evolved code on architecture depends on whether the evolved code changes the files or the dependencies between files. When the evolved code does not change files, architecture does not need to be evolved, otherwise, architecture needs to be modified based on code change. In this paper, we only need to consider the latter situation.

The method consists of three steps: (1) Obtaining code change by using a change detection method; (2) Modifying FDG based on code change; (3) Recovering architecture based on the modified file dependency graph.

We propose mapping rules from code change to FDG, then according to code change, we modify FDG based on mapping rules. Code changes can be divided into two types: edge changes and node changes. The following types of changes

Algorithm 1 The algorithm of calculating the approximate optimal solution of loss function

Input:
let CDG be the component dependency graph
let FDG be the file dependency graph
let req be the code change
Output:
let $result$ be the divided set of files
Function $partition(CDG, FDG, req)$
$node \leftarrow findNode(CDG, req.comp)$
$k \leftarrow req.comp$
/*Construct the local dependency graph named ldg formed by the files contained in the split components */
$ldg \leftarrow findLocalDG(FDG, node.files)$
/* Combine the outgoing and incoming edges of nodes in the graph named ldg to form an undirected dependency graph*/
$udg \leftarrow transfer(ldg)$
$result \leftarrow preclassify(udg, k)$
/*add label nodes according the pre-classification result*/
$addLabelNodes(udg, s1, s2, ..., sk, result)$
$curNum \leftarrow 0$
while $curNum < k$ **do**
    /* Renaming two label nodes with the largest weight by S and T respectively*/
    $< S, T > \leftarrow renameLabelNodeWithLargestWeight(udg, s1, s2, ..., sk)$
    $cutEdges \leftarrow mincut(udg, < S, T >)$
    $< n|n \in S, n|n \in S > \leftarrow restorePartition(udg, cutEdegs)$
    if $n|n \in S$ **then**
        $recordPartion(newComps, result)$
        return $newComps$
    else
        $removeNode(udg, n|n \in S)$
        $addPartition(newComps, n|n \in S)$
        $curNum = curNum + 1$
    end if
end while
return $newComps$
Procedure $mincut(udg, < S, T >)$
$cutEdges \leftarrow \emptyset$
$maxflow \leftarrow 0$
while $findAugPath(S, T) = true$ **do**
    $maxflow \leftarrow maxflow + maxFlowByDfs(S, T)$
end while
$cutEdges \leftarrow findCutEdgeByBfs(S, T)$
return $cutEdges$
Procedure $preclassify(udg, k)$
/* The sum of weight of edge of each node is calculated */
$Emap \leftarrow calWeight(udg)$
/* The first k nodes with the maximum weight are selected as central node */
$centers \leftarrow findKthNode(map)$
$disArray \leftarrow \emptyset$
$curNum \leftarrow 0$
while $curNum < k$ **do**
    $< center, node, dis > = calDis(udg, centers)$
    $curNum+ = < center, node, dis >.size$
    $disArray = disArray \cup < node, center, dis >$
end while
$result \leftarrow \emptyset$
for each *node* $\in udg.nodes$ **do**
    /* The sum of weight of edges belongs to the node is calculated as totalW*/
    $totalW = calWeight(udg, node)$
    $disMap \leftarrow \emptyset$
    for each *center* $\in centers$ **do**
        $dis \leftarrow findDisWithNodesBelongCenter(disarray, center, node)$
        $disMap \leftarrow disMap \cup < center, totalW - dis >$
    end for
    /*Find the target center with the minimum distance*/
    $targetCenter \leftarrow findTargetCenter(disMap)$
    $result \leftarrow result \cup < node, center >$
end for
return $result$

are related to nodes. In this paper, we use a multiple-level change detection method to extract changed code [11].

Adding a file corresponds to adding a file node in FDG. The statements belong to the added file are the slice criteria, and we obtain which files have dependencies with the new file, then we add related dependency edges between the added file node with other file nodes.

Deleting a file corresponds to deleting a file node from FDG, and its related edges are deleted.

The following types of changes are related to edges.

Increasing dependency intensity between files corresponds to increasing the weight of the edge, and the weight of the edge is assigned based on the dependency intensity.

Adding a dependency between files corresponds to adding an edge between two file nodes.

Reducing dependency intensity between files corresponds to reducing the weight of the edge, and the weight of the edge is assigned based on the dependency intensity.

Deleting a dependency between files corresponds to deleting a dependency edge between two file nodes.

After modifying FDG, we use cluster methods to obtain new architecture based on modified FDG. In this paper, we adopted a cohesive hierarchical clustering method [12]. This clustering method initially treats each file in the file dependency graph as a cluster, and then continuously merges the clusters with a small distance between clusters, and updates the distances between the new clusters and other clusters. The algorithm is shown in Algorithm 2. Finally, we obtain new architecture based on the evolved code by implementing the above algorithm.

III. Experiments and Evaluation

A. Experiment Setup

In the section, we conduct experiments to evaluate the effectiveness of our method. We conduct our method with eight open source programs to answer the following research questions.

RQ1: is the co-evolutionary method of code based on evolved architecture effective?

RQ2: is the co-evolutionary method of architecture based on evolved code effective?

We randomly selected eight open source projects as the experimental cases, and these projects contain Java projects, C projects, and C++ projects. The information about these projects is listed in Table I.

TABLE I
The information about experimental cases

Project	Evolution process	Language	LOC
Apns	0.1.5→0.2.0	Java	3K
La4j	0.5.0→0.5.5	Java	9K
AssertJ	3.2.0→3.3.0	Java	19K
GoogleMock	1.5.0→1.6.0	C++	16K
Filezilla	3.30.0→3.31.0	C++	128K
Lua	5.0.0→5.0.1	C	11K
Libev	1.3.2→1.4.8	C	25K
Bash	4.4.12→ 4.4.18	C	103K

Algorithm 2 The hierarchical clustering algorithm

Input:
Let fg be the file dependency graph after evolution
Let $reqs$ be the code change requirements
Let $compLocsb$ be the collection of the lines of code for every components
Let $locb$ be the total lines of code before evolution
Output:
Let cg be the component graph after evolution

Function $clustering(fg, reqs, compLocsb, locb)$
$cg \leftarrow clone(fg)$
$n_b \leftarrow size(compLocs_b)$
/* Calculate the number of components before evolution*/
$loc_a \leftarrow loc_b$
/* Calculate the expected number of components after evolution as stop condition*/
$stopc \leftarrow generateStopCondition(cg, reqs, compLocsb, locb, nb)$
$n_a \leftarrow n_b$
while $n_a > stopc$ **do**
  /*Find the edge with min distance from component diagram*/
  $< src, dest > \leftarrow findMinDPair(cg)$
  $merge(cg, < src, dest >)$
  $n_a \leftarrow n_a - 1$
end while
return cg
Procedure $generateStopCondition(cg, reqs, compLocs_b, loc_b, n_b)$
$average_b \leftarrow loc_b/n_b$
for each $req \in reqs$ **do**
  if $reqinstanceofAddComp$ **then**
    $loca \leftarrow loca + averageb$
  else
    if $reqinstanceofRemoveComp$ **then**
      $loc_a \leftarrow loc_a - compLoc_b.get(req.comp)$
    end if
  end if
end for
if $loca \neq locb$ **then**
  $stopc \leftarrow loc_a/average_b$
end if
return $stopc$
$merge(cg, < src, dest >)$
/* Find the node whose id is src from component graph remove the graph*/
$nodeSrc \leftarrow findNode(cg, src)$
$nodeDest \leftarrow findNode(cg, dest)$
$inEdgeSrc \leftarrow findInEdges(cg, src)$
$outEdgeSrc \leftarrow findOutEdges(cg, src)$
$inEdgeDest \leftarrow findInEdges(cg, dest)$
$outEdgeDest \leftarrow findOutEdges(cg, dest)$
$nodeSrc.name \leftarrow nodeSrc.name + nodeDest.name$
$inEdgeSrc \leftarrow inEdgeSrc \cup EdgeDest$
$outEdgeSrc \leftarrow outEdgeSrc \cup EdgeDest$
$cg \leftarrow cg - nodeDest$
/*Remove the cluster named nodeDest from component graph*/

In our experiments, we take actual history versions as experimental cases. V_b and V_a are two software versions before and after evolution. The actual code and actual architecture of V_b are respectively denoted in V_{b_c} and V_{b_a}. Similarly, the actual code and actual architecture of V_a are respectively denoted in V_{a_c} and V_{a_a}.

B. Results and Evaluation

RQ1: Is the co-evolutionary method of code based on evolved architecture effective?

We conduct the following two experiments to analyze the effectiveness of our method: (1) Is the co-evolved code consistency with the actual architecture after evolution? (2) Is the co-evolutionary content of co-evolved code consistency with the actual code changes?

The first experiment is that, we analyze the consistency between the actual code V_{a_c} with the co-evolved code which is obtained by using our method base on V_{a_a}.

Code and architecture belong to different granularity levels, and they cannot be compared directly. We measure the effectiveness of our method based on architectural similarity. The details of the experiments are as follows:

- Co-evolving code by using our method to obtain the new FDG.
- Recovering architecture based on the new FDG to obtain the new architecture.
- Calculating the architectural similarity between the actual architecture and the new architecture.

We use component similarity to assess the similarity between the actual architecture and the new architecture. It is calculated as Formula 3.

$$ComSimilarity(C_i, C_j) = \frac{|F_i \cap F_j|}{|F_i \cup F_j|} \tag{3}$$

In Formula 3, $ComSimilarity(C_i, C_j)$ is the component similarity between the i_{th} component C_i and the j_{th} component C_j, F_i is the set of files which belongs to the i_{th} component, and $|X|$ is the number of elements of the set X. Architecture similarity is the average value of all components similarity.

The component similarity and the architecture similarity are shown in Table II.

TABLE II
THE COMPONENT SIMILARITY AND THE ARCHITECTURE SIMILARITY

Project	Actual component	New componet	Component similarity	Architecture similarity
La4j	data1	decomposition	0.563	
	data2	matrix/source	0.619	
	operation	operation	1	0.836
	linear	linear	1	
	matrixOperation	matrixOperation	1	
Filezilla	interface1	interface#85	1	
	interface2	interface\setting	1	
	interface#1	interface#86	0.882	
	engine	engine	0.961	0.978
	putty	putty	1	
	dbus	dbus	1	
	interface#2	interface#5	1	
Bash	readline1	readline#13	1	
	readline2	termcap	1	
	[ROOT]#1	[ROOT]#32	0.931	
	[ROOT]	[ROOT]#29	0.885	0.977
	support	suppport	1	
	examples\loadables	examples\loadables	1	
	lib\sh	lib\sh	1	
	lib\intl	lib\intl	1	
Libev	libcork1	libcork\config	1	
	libcork2	libcork#50	1	
	libipset	libipset	0.625	
	\	libipset\map	0	0.703
	\	libipset\set	0	
	libcork#1	libcork#52	1	
	libev#47	libcork#52	1	
	libev	libev#47	1	
GoogleMock	gmock1	gmock#10	1	
	gmock2	gmock#15	1	1
	src	src	1	
	internal	internal	1	
Bash	readline1	readline#13	1	
	readline2	termcap	1	
	[ROOT]#1	[ROOT]#32	0.931	
	[ROOT]	[ROOT]#29	0.885	0.977
	support	suppport	1	
	examples\loadables	examples\loadables	1	
	lib\sh	lib\sh	1	
	lib\intl	lib\intl	1	

TABLE III
THE CO-EVOLUTIONARY CONTENT OF CODE AND THE ACTUAL CODE CHANGES IN LIBEV PROJECT

Architecture change	The co-evolutionary content of code	Actual code changes
Adding component named libipset	Adding highly cohesive files	Adding two directories named libipset\include\ipset and libipset
Adding method invocation dependencies between the two components named src and libipset	Adding method invocation dependencies between the files contained in the two components named src and libipset	Adding method invocation dependencies between src\acl.c and libipset\include\ipset\ipset.h.
Increasing include reference dependencies between the two components named src and libev	Increasing include reference dependencies between the files contained in the two components named src and libve	1) Adding include reference dependencies between src\tunnel.h and libev\ev.h. 2) Adding include reference dependencies between src\udprelay.h and libev\ev.h.
Splitting the component core into core#1 and core#2	Splitting the files contained in the component named core into two parts and reducing the dependencies between the two parts.	1) Reducing dependencies between org.assertj.core.api and org.assertj.core.condition. 2) Reducing dependencies between org.assertj.core.api and org.assertj.core.util.

As Table II shows that, the average architecture similarity is 0.899. The similarity indicates that our method can co-evolve architecture based on evolved code effectively.

Here, we take `Libev` as an example to show the corresponding relations between the co-evolutionary content of code and the actual code change, the corresponding relations are shown in Table III.

According to Table III, the co-evolutionary content of code is consistent with the actual code change, so our method is effective for co-evolving code.

RQ2: Is the co-evolutionary method of architecture based on evolved code effective?

The co-evolutionary change method of architecture based on evolved code aims at keeping the consistency between architecture and the evolved code. We evaluate the effectiveness of the method by analyzing the consistency between the co-evolved architecture and the actual architecture change.

We conduct the following two experiments: (1) Is the co-evolved architecture consistency with the actual architecture after evolution? (2) Are the details of co-evolved architecture consistency with the actual architecture changes?

The experiment is performed in the following steps:

- Obtaining the actual code change by using the change detection method.
- Obtaining the co-evolved architecture by using our method.
- Obtaining the actual architecture change by comparing CDG before and after evolution.
- Analyzing the consistency between the co-evolved architecture and the actual architecture change.

The co-evolved architecture and the actual architecture changes are about CDG, and the graph consists of the component and the dependency edges between components, that is if the co-evolved architecture is consistent with actual changes, the number of changed nodes and changed edges of them are the same. We summarize the number of changed nodes and the number of changed edges of each project to analyze the consistency, and the information is shown in Table IV.

Table IV shows statistical information about the number of changed nodes and dependency edges. As the table shows that the number of co-evolved objects is equal to the number of actual changes, that is, our method can co-evolve architecture which is consistent with the actual architecture changes. So the co-evolutionary method of architecture based on evolved code is effective.

TABLE IV
THE NUMBER OF CHANGED NODES AND CHANGED EDGES.

		Type	La4j	AssertJ	Lua	Libev	Apns
Co-evolved objects	Node	Adding	2	23	4	2	6
		Deleting	29	0	2	4	2
	Edge	Adding	30	231	10	124	15
		Deleting	274	10	2	0	5
		Increasing	3	44	9	8	13
		Reducing	3	6	5	3	2
Actual change	Node	Adding	2	23	4	2	6
		Deleting	29	0	2	4	2
	Edge	Adding	30	231	10	124	15
		Deleting	274	10	2	0	5
		Increasing	3	44	9	8	13
		Reducing	3	6	5	3	2

In the second experiment, we further analyze the detail between actual architecture.

Here, we take `La4j` as an example to analyze the consistency between the co-evolutionary content of architecture and the actual change. The details are shown in Table V.

As Table V shows that, actual architecture change in the actual evolution is "Add org.la4j.Matrix node", the corresponding co-evolutionary content of the architecture is "Add org.la4j.Matrix node", so the co-evolutionary content matches the actual architecture change. The evolution of dependency is mainly reflected in the addition or deletion of dependency edges between nodes or changes in the weights of edges. According to Table V, we know that, the actual architecture change is consistent with the co-evolved architecture.

IV. RELATED WORK

At present, there are mainly four co-evolutionary methods which are shown in the first column of Table VI.

Direction. The first two methods only support one-way co-evolution, so the two methods support fewer application scenarios than the other methods.

Representation. Most of these methods use the component dependency graph as the representation of architecture, except the multi-view method. The component dependency graph is a widely acceptable representation of architecture. Considering architecture and code may be used in other researches, we think that the component dependency graph is more applicable for representing architecture.

TABLE V
THE CO-EVOLVED ARCHITECTURE AND THE ACTUAL ARCHITECTURE CHANGES IN LA4J PROJECT

Actual Code change	**The co-evolutionary content of architecture**	**Actual architecture change**
Adding a file Matrix in the package org.la4j.	Adding a new node org.la4j.Matrix.	Adding a new node org.la4j.Matrix.
Deleting the file CCSFactory of the package org.la4j.factory.	Deleting the node org.la4j.factory.CCSFactory.	Deleting the node org.la4j.factory.CCSFactory org.la4j.factory.
Adding 4 method invocation dependencies between org.la4j.LinearAlgebra and org.la4j.Matrix.	Adding a new dependent edge between org.la4j.LinearAlgebra and org.la4j.Matrix, and the weight is 4.	Adding a new dependent edge between org.la4j.LinearAlgebra and org.la4j.Matrix, and the weight is 4.
Deleting the generalization dependencies between the two files ArrayVectorSource and VectorSource which are contained in the package org.la4j.vector.source.	Deleting the dependent edge between the two nodes org.la4j.vector.source. ArrayVectorSource and org.la4j.vector.source. VectorSource.	Deleting the dependent edge between the two nodes org.la4j.vector.source. ArrayVectorSource and org.la4j.vector.source. VectorSource
Increasing the parameter dependencies between the two files org.la4j.linear.JacobiSolver and org.la4j.Matrix, and the increased dependencies are 2 times.	Increasing the weight of edge between the two nodes org.la4j.linear. JacobiSolver and org.la4j.Matrix by 2.	Increasing the weight of edge between the two nodes org.la4j.linear.JacobiSolver and org.la4j.Matrix by 2.
Reducing the method invocation dependencies between the two files AbstractSolver and LinearSystemSolver which are contained in the package org.la4j.linear.	Reducing the weight of edge between the two nodes org.la4j.linear.AbstractSolver and org.la4j.linear. LinearSystemSolver by 1.	Reducing the weight of edge between the two nodes org.la4j.linear.AbstractSolver and org.la4j.linear. LinearSystemSolver by 1.

TABLE VI
THE COMPARISON BETWEEN RELATED METHODS

Method	**Direction**	**Representation**	**Technology**	**Implementation**
Architecture recovery	One-way	Component dependency graph	Mapping rules, clustering algorithm	Automatic
Generate code	One-way	Component dependency graph	Mapping rules, clustering algorithm	Automatic
Multi-View	Two-way	UML	Mapping relations	Automatic
Information fusion	Two-way	Component dependency graph	Logic element programming, metadata, language development	Artificial

Technology. The table shows that the mapping rules are widely used in many methods. It indicates that the mapping rules are effective for co-evolutionary methods.

Implementation. Most of these methods are implemented automatically, but the information fusion method is implemented artificially. So the information fusion method is not suitable for large-scale programs.

According to the above analysis of related work, we know that the above four methods do not support the two-way co-evolution automatically between component dependency graph and code automatically. So, we propose a co-evolutionary method to solve these problems.

V. CONCLUSION AND FUTURE WORK

In the paper, we propose a co-evolutionary method between architecture and code, including the co-evolutionary method of code based on evolved architecture and the co-evolutionary method of architecture based on evolved code. In our method, more types of change actions are taken into consideration, and the changes are converted based on mapping rules, then we use FDG as the intermediate level between architecture and code to perform co-evolutionary algorithms. We conduct the experiments with eight open source projects, and the experimental results indicate that the co-evolutionary method of architecture based on evolved code and the co-evolutionary of code based on evolved architecture are all effective. In our future work, we will combine our method with architecture quality, then we improve architecture quality automatically.

REFERENCES

[1] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97. IEEE, 2019.

[2] Daniel Link, Pooyan Behnamghader, Ramin Moazeni, and Barry Boehm. The value of software architecture recovery for maintenance. In *Proceedings of the 12th Innovations on Software Engineering Conference*, pages 1–10, 2019.

[3] Ana Paula Allian, Bruno Sena, and Elisa Yumi Nakagawa. Evaluating variability at the software architecture level: an overview. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 2354–2361, 2019.

[4] Burak Uzun and Bedir Tekinerdogan. Domain-driven analysis of architecture reconstruction methods. In *Model Management and Analytics for Large Scale Systems*, pages 67–84. Elsevier, 2020.

[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Ma, and Jean Bernard Stefani. The fractal component model and its support in java. *Software Practice & Experience*, 36(11-12):1257–1284, 2010.

[6] R. France and J. M. Bieman. Multi-view software evolution : A uml-based framework for evolving object-oriented software. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 386–395, 2001.

[7] Pooyan Jamshidi and Claus Pahl. Business process and software architecture model co-evolution patterns. In *International Workshop on Modeling in Software Engineering*, pages 91–97, 2012.

[8] Mitchell A Potter and Kenneth A De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2014.

[9] Amir Ngah and Siti Aminah Selamat. Using object to slice java program. *Journal of Engineering and Applied Sciences*, 13(6):1320–1325, 2018.

[10] Yeounoh Chung, Tim Kraska, Neoklis Polyzotis, Ki Hyun Tae, and Steven Euijong Whang. Slice finder: Automated data slicing for model validation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1550–1553. IEEE, 2019.

[11] Tong Wang, Dongdong Wang, Ying Zhou, and Bixin Li. Software multiple-level change detection based on two-step mpat matching. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, 2019.

[12] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.