The Journal of Systems and Software 208 (2024) 111875

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

EsArCost: Estimating repair costs of software architecture erosion using slice technology☆

Tong Wang [a,*], BiXin Li [b]

[a] School of Computer Science and Technology, Anhui University of Technology, Maanshan, China
[b] School of Computer Science and Engineering, Southeast University, Nanjing, China

ARTICLE INFO

Keywords:
Software architecture
Architecture erosion
Repair costs
Software evolution

ABSTRACT

Software architecture erosion has a negative effect on software quality, software performance and evolution cost, so repairing architecture erosion is an important task. However, in the actual evolution process, due to the pressure of cost, it is not practical to repair all erosion problems. To repair more erosion problems at a certain total cost, developers would better know the appropriate repair cost of each erosion problem. In this paper, we propose an approach called EsArCost, which can locate the reasons for architecture erosion and estimate the repair cost of each erosion problem. To this end, EsArCost detects multi-level changes of software architecture and calculates the architecture erosion degree, then locates which changed codes cause architecture erosion. EsArCost further calculates the full slice of each erosion problem to estimate the difficulty and repair costs. We evaluate our approach on small and large open source programs, and the experiment results indicate that EsArCost can effectively and efficiently estimate repair costs.

1. Introduction

Software architecture is continuously evolved to meet new requirements (Paixao et al., 2017). However, unreasonable changes will lead to architecture erosion (Bhat et al., 2020; Hohpe et al., 2016). Architecture erosion will increase the evolution cost, prolong the evolution cycle, increase the difficulty of development and reduce software performance (Li et al., 2022a; Kadri et al., 2020; Behnamghader et al., 2017). Software architecture erosion, like software bugs, as software evolves in size, the negative impact will increase, so repairing architecture erosion problems is very important to avoid the continued increase in negative impacts (Kadri et al., 2021; Allian et al., 2019). But in an actual development environment, there may not be enough time and cost to repair all problems, so we need an approach to estimate the repair cost of each erosion problem, then we can select the problems that require less cost to repair first, so that we can repair more erosion problems at a certain cost.

In terms of cost estimation, there exists a vast literature on software cost estimation methods, and they have high practical value (Chatzipetrou, 2019; Huang et al., 2017). The estimation methods of software can be divided into the following types: expert estimation, AI estimation based on historical databases, parametric models and size-based estimation models (Kuan, 2017; Pospieszny, 2017; Pospieszny et al., 2018). However, there are two important differences between software and software architecture that cause the methods of estimating costs of software cannot be used for repairing architecture erosion. Firstly, software architecture is a high-level abstraction of software, so some software faults that are not important to architecture may be overlooked in the process of progressive abstraction, that is, not all faults will cause software architecture erosion, and architects only need focus on architecture-related faults, but the existing methods did not analyze which faults are related to architecture. Secondly, the way to repair architecture erosion is to modify software architecture. However, the estimation methods of software did not analyze the cost of modifying architecture-related code elements. In a word, the estimation methods of software are not applicable to software architecture and architecture erosion.

According to the above analysis, we can know the cost estimation of software is not different from that of architecture erosion. In addition to that, due to the features of architecture erosion, two issues need to be addressed. First, the cause of architecture erosion is unreasonable

☆ Editor: Laurence Duchien.
* Corresponding author.
E-mail address: tongwang_se@163.com (T. Wang).

https://doi.org/10.1016/j.jss.2023.111875
Received 1 December 2022; Received in revised form 11 April 2023; Accepted 10 October 2023
Available online 25 October 2023

changes in code, and the number of code lines is an important factor in estimating costs. However, it is a problem to correlate the changes between low and high levels to locate the cause of architecture erosion at the code level, and estimate costs based on the number of code lines. Second, when repairing architecture erosion, we not only need to modify the statements that cause architecture erosion but also their related statements, so whether all related statements can be extracted is an important factor for the accuracy of estimation results. And in large-scale programs, the relationship between statements is complex, how to extract all related statements and estimate costs based on them is a difficult problem.

To address the above issues, we propose an approach called EsArCost, which estimates the repair cost of architecture erosion using slice technology. In our approach, we first construct MAT (Multilevel Architecture Tree) to detect changes before and after architecture evolution, because the unreasonable changes cause architecture erosion. Second, we locate erosion points at the code level based on analyzing the measurement formula of architecture erosion. Third, we calculate the slice of each erosion point, so that we can know which codes affect or are affected by the erosion point, and we estimate the repair cost based on the slice.

To sum up, this work makes the following contributions.

- We efficiently and effectively locate the changed codes using a change detection method based on two-step matching.
- We analyze reasons for architecture erosion at the code level, so we can precisely locate the cause of erosion and facilitate the implementation of repairs.
- We estimate the repair costs based on the slices of changed codes. The set of statements in the slice can be used to effectively estimate the amount of work involved in repairing erosion problems.

The paper is organized as follows. In Section 2, we introduce some basic terminologies used in our paper. In Section 2 and III, we present our approach and illustrate the working process based on an example. In Section 4, we implement our approach on open source projects to evaluate our approach. Section 5 and Section 6 present the threats and the conclusion.

2. Background

This section covers the supporting technologies behind our work, including software architecture erosion and program slicing.

2.1. Software architecture erosion

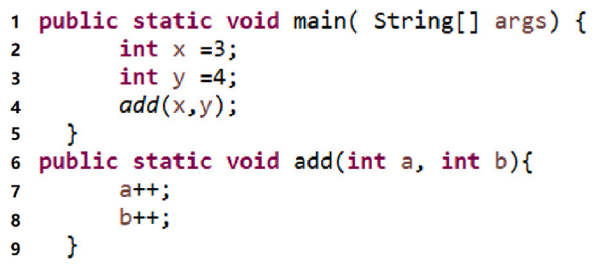
In this section, we introduce some basic terminologies used in our approach.

In this paper, we use CDG (Component Dependency Graph) to present software architecture. It is a high abstraction level of the file dependency graph (Li et al., 2022a). In the component dependency graph, a node is a component, and an edge is a dependency between components.

Definition code Entity: It is a unit collection of source code, which can refer to a statement, a file, a class, a method, etc., depending on the granularity of the collection.

The attributes caused by the implementation of code are not be expressed in front of developers and users, such as the size of component, the dependency between components, etc., so they are called internal attributes. On the contrary, some quality attributes can be expressed in front of developers and users, such as understandability, modifiability, etc., and they are called external attributes.

The phenomena for architecture erosion include unable to meet new functions, unable to continue maintenance, unable to operate for a long time, and errors in the software (Mitra et al., 2021; Li et al., 2022b).



```
1 public static void main( String[] args) {
2       int x =3;
3       int y =4;
4       add(x,y);
5   }
6 public static void add(int a, int b){
7       a++;
8       b++;
9   }
```

Fig. 1. The source code of the slicing example.

At the same time, architecture erosion is also related to operating conditions, such as memory leaks and performance degradation (Knieke et al., 2021; Lutellier et al., 2017). Although researchers analyze and evaluate architecture erosion from different viewpoints, the consensus that can be reached is that the root cause and most significant manifestation of architecture erosion is the degradation of software quality, so we define architecture erosion based on architecture qualities.

Definition architecture erosion: A_i and A_j are two architectures before and after evolution. A_i is evolved to A_j, then the software architecture quality is decreased with evolution, and A_j is eroded. We call the phenomenon architecture erosion.

2.2. Program slicing

Program slicing is an important technique for analyzing programs, and it analyzes the dependence of source code statements, and gives program fragments, i.e., a slice, relative to the user concern.

The slicing process consists of two parts, one is the construction of a program dependence graph, and the other is traversing the graph with the slicing criterion.

Definition system dependence graph(SDG): it is a graph for presenting the dependencies between code entities, where a node represents a code entity, and an edge represents a type of dependency between two code entities.

Definition slicing criterion: It contains two elements $\langle s, v\rangle$, where s denotes the location of a code entity, such as a statement, a variable, etc., and v denotes a variable or a statement.

Here, we take an example to show how to calculate a slice. Fig. 1 is the source code of a simple program.

The first step of calculating slices is constructing the dependence graph. We construct SDG for the above program as shown in Fig. 2. In the SDG, there are three types of dependencies, control dependence, data dependence and parameter dependence. Besides the above dependencies, there may be anonymous dependencies, communication dependencies, thread dependencies, etc. In the SDG, line 4 is refined into three nodes, line 4, the variable y on line 4 and the variable x on line 4. In a word, slice technology can convert a program into a fine-grained graph.

The second step of calculating slices is executing the slicing algorithm based on SDG, and in this paper, we adopt the sub-statement type slicing algorithm (Lulu et al., 2020). According to the slicing direction, slices can be divided into forward and backward slices. The forward slice is the set affected by the slicing criterion, and the backward slice is the set that will affect the slicing criterion. Here, we calculate the forward slice with the slicing criterion $\langle 6, a\rangle$ which means we calculate which statements affect the variable a of the sixth line. We traverse the SDN, then we extract four nodes, nodes 1, 4, 6 and 4.x. The statements corresponding to these nodes contain the first, fourth and sixth lines. So the slice with the slicing criterion $\langle 6, a\rangle$ is 1,4,6.

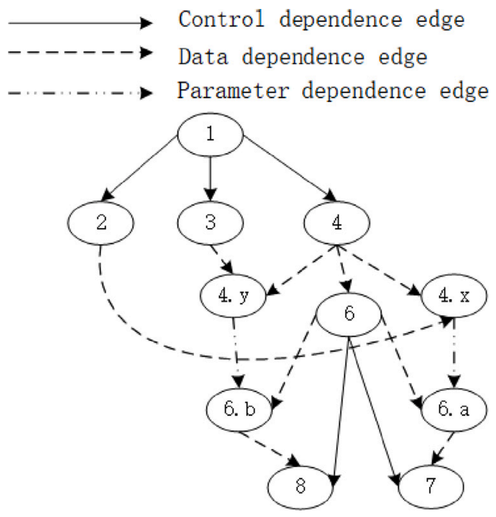


Fig. 2. The SDG of the slicing example.

3. Our approach

In this paper, we propose an approach, EsArCost, aiming at estimating the repair cost of each architecture erosion point, to help developers repair more erosion points at a certain repair cost. Fig. 3 illustrates the working process of EsArCost which consists of three steps.

Step 1: Detecting architecture changes. We construct MAT for each program and implement the two steps algorithm to detect changes at each level. As shown in Fig. 3, we detect different types of changes, and they are marked by different colors, where the red node is the deleted node, the blue node is the updated node, and the yellow node is the renamed node.

Step 2: Locating architecture erosion points. We locate the erosion points based on erosion reasons analyzed from the measurement formula. As shown in Fig. 3, the gray square is the erosion point.

Step 3: Estimating repair costs based on slices of erosion points. We calculate the slice of each erosion point to extract which statements affect or are affected by erosion points, and as shown in Fig. 3, all green nodes are the slice nodes of the erosion point. Then we estimate the cost based on the location of the related statement and the difficulty of the modification.

3.1. Detecting architecture changes

To facilitate the analysis of the correlation between code entity changes, we propose MAT. MAT is a tree representing the multilevel architecture of a program. The root node of MAT represents the overall software architecture, then the root node is refined into three levels, component level, file level and statement level, where the component level is used to represent the components contained in the architecture, the node of the file level represents the file, and the node of the statement level represents the statement. In MAT, a node represents a code element, and an edge between nodes indicates the relation between the upper code element and the lower code element. A node has three attributes: node level, node type and node value. The node level refers to the abstraction level of the node in MAT; the node type refers to the type of code element in the syntactic rule, such as *PackageDeclaration*, *Modifier* and so on; the node value refers to the code string corresponding to the node, such as the value represented by a *String* value. The construction process of MAT includes three steps: first, we construct the architecture structure tree containing root node, component node and file node; then, we construct the AST (Abstract Syntax Tree) for each file as the statement nodes; finally, we combine architecture structure tree and ASTs to generate MAT.

When the MAT tree is constructed, a two-step matching detection algorithm is implemented based on MAT, which consists of top-down matching and bottom-up matching.

(1) The top-down matching process can quickly identify whether two subtrees are identical, so it reduces the number of nodes to be matched by quickly eliminating a large number of nodes contained in the subtrees, thus improving the overall change detection efficiency.

(2) In the bottom-up matching algorithm, the similarity between subtrees is first calculated, and then the remaining nodes are matched.

We use Formula (1) to calculate the similarity between two code entities. x and y are two nodes, if $label(x) = label(y)$ and $similarity > minSim$ ($minSim$ is the matching threshold), x and y constitute a pair of matching nodes. $common(x, y)$ refers to the number of the same matching nodes between x and y, $sim_{contn}(x, y)$ is the similarity between x and y, and $size(x)$ refers to the total number of nodes contained in x.

$$sim_{contn}(x, y) = \frac{2 * common(x, y)}{size(x) + size(y)} \qquad (1)$$

To identify the change types based on similarity, we define two thresholds. The first threshold is $HighThreshold$. If the similarity is greater than $HighThreshold$, we deem that the two code entities are the same, that is, the code entity is not changed during the evolution process. The second threshold is $LowThreshold$. If the similarity is less than $HighThreshold$ but higher than $LowThreshold$, we deem that the change type between the code entities is updating. If the similarity is less than $LowThreshold$, we deem that the two code entities are the new code entity of the current architecture and the deleted code entity of the original architecture. The algorithm for identifying change operations is shown in Algorithm 1.

Algorithm 1 The algorithm for identifying change operations

```
Require: The code entity list of the original version $oriCom$
    The code entity list of the current version $curCom$
    The threshold of similarity $threshold$
Ensure: The list of the adding change $add$
    The list of the deleting change $delete$
    The list of the updating change $update$
1: Let $oriMatched$ store the matched component of $oriCom$
2: Let $curMatched$ store the matched component of $curCom$
3: Let $match$ mark the component whether has been matched successfully
4: Set $match$ = false
5: for each $oriComponent \in oriCom$ do
6:   if match = true then
7:     $match$ = false
8:     Continue
9:   end if
10:  for each $curComponent \in curCom$ do
11:    Let $similarity$ is the similarity between $oriComponent$ and $curComponent$
12:    if $similarity = 1$ then
13:      put $oriComponent$ in $oriMatched$
14:      put $curCmponent$ in $curMatched$
15:      $match$ = true Break
16:    else
17:      if $similarity > threshold$ then
18:        put $oriComponent$ in $oriMatched$
19:        put $curCmponent$ in $curMatched$
20:        put $< oriComponent, curCmponent >$ in $update$
21:        $match$ = true Break
22:      end if
23:    end if
24:  end for
25: end for
26: $delete$ = $oriCom$ - $oriMatched$
27: $add$ = $curCom$ - $curMatched$
```

To mark the change between architectures before and after evolution, we present the dentition of the changed pair.

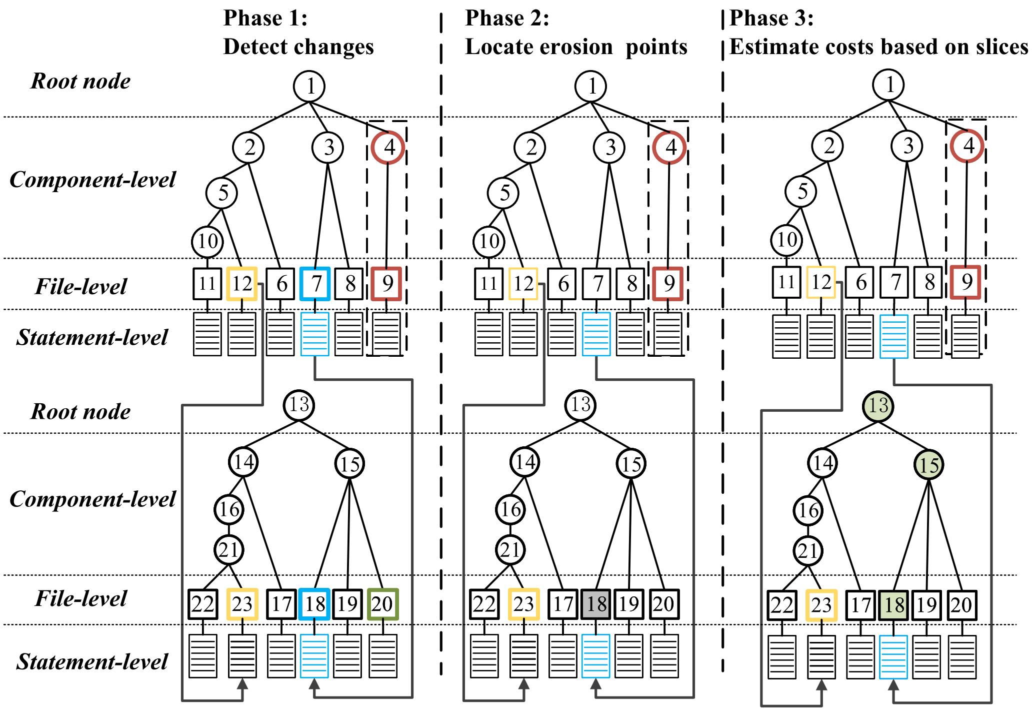


Fig. 3. The workflow of our approach.

Definition changed pair: C_i and C_j are two code entities, and they belong to different architectures A_i and A_j. A_i is evolved to A_j, and C_i is evolved to C_j, then we call $\langle C_i, C_j\rangle$ is a changed pair. When C_i is deleted from A_i, its changed pair is $\langle C_i, \phi\rangle$. When C_j is a new code entity of A_j, its changed pair is $\langle \phi, C_j\rangle$. Between their MATs, we establish a dependency edge from C_i to C_j, and the dependency edge is called the matching edge.

3.2. Locating architecture erosion points

According to the definition of architecture erosion, we present the definition of erosion degree and quantitative it.

Definition erosion degree: A_i is evolved to A_j, and A_j is eroded, then the degree of difference in the overall architecture qualities between A_i and A_j is called the erosion degree.

According to the definition of the erosion degree, we propose Formula (2) to calculate it, where b and a are the architecture before and after evolution, $ED(a,b)$ is the erosion degree of a which is evolved from b, N is the set of focused architecture quality attributes, $|N|$ is the number of elements of N, N_{i_a} is the measurement result of the i_{th} quality attribute of a, w_i is the weight of the i_{th} quality attribute.

$$\mathrm{ED(a,b)} = \frac{\sum_{i=1}^{|N|}\left[w_i * \left(N_{i\_a} - N_{i\_b}\right)\right]}{\sum_{i=1}^{|N|} w_i} \tag{2}$$

If $ED(a,b)$ is greater than 0, a is not eroded, otherwise, its erosion degree is $ED(a,b)$.

The erosion degree is a relative value, that is, we use it to evaluate which version is severely eroded. For example, if the measurement results of architecture quality for the first, second and third versions are 0.9, 0.6 and 0.5, respectively. Then the erosion degrees of the second and third versions are 0.3 and 0.1, respectively. Although the second version has a better architecture quality than version 3, its quality is greatly decreased after evolution, so the second version is more eroded than the third version.

Definition erosion point: If $\langle C_i, C_j\rangle$ is a changed pair, and its change decreases architecture quality, then C_j is an erosion point.

According to Formula (2), we can know that the erosion degree depends on the measurements of architecture quality attributes, and the measurements of architecture quality attributes depend on internal attributes of architecture. So we analyze and locate erosion points based on the internal attributes of architecture. For example, if architecture is eroded, then we analyze which architecture qualities are decreased, and we further analyze which internal attributes related to these architecture qualities are decreased. So the fundamental reason for architecture erosion is the decreasing internal attributes.

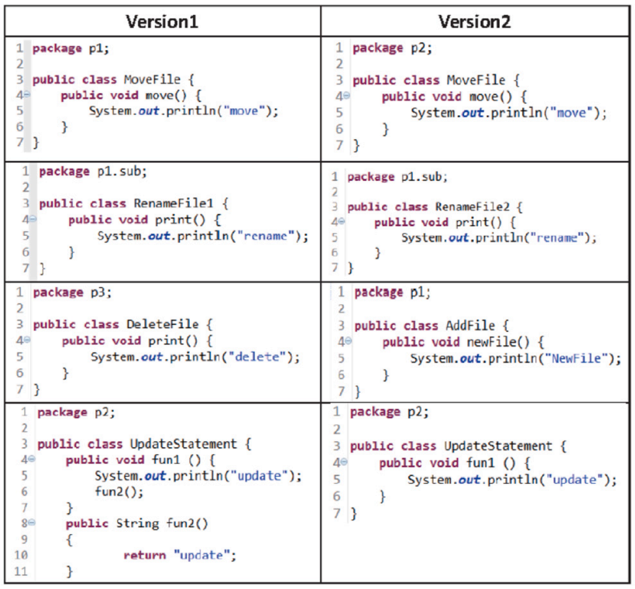
3.3. Estimating repair costs based on slices

The architecture is abstracted from code, so in practice, repairing the architecture erosion needs modifying the code, that is, repairing erosion points on the code level.

When modifying the erosion point statement, due to the dependency between statements, some statements that affect the erosion point and some statements that are affected by the erosion point need to be co-modified. And the co-modified work is an important factor in determining the cost of repair.

To facilitate the extraction of all statements associated with an erosion point in a large and complex structured program, we use slice technology to detect the indirect impact of the erosion point. We use the erosion point as the slicing criterion, and then we calculate its slice. First, we calculate the forward slice to obtain the statement set which is affected by the erosion point, then we calculate the backward slice to obtain the statement set which affects the erosion point, and finally, we calculate the union set of the forward slice and the backward slice called the full slice, and we estimate repair cost based on the full slice.

The repair cost depends on two factors: the maintenance difficulty and the number of related statements. The first factor depends on the degree of closeness between the erosion point and the statements that



Version1	Version2

`package p1;`
`public class MoveFile {`
`    public void move() {`
`        System.out.println("move");`
`    }`
`}`
	`package p2;`
	`public class MoveFile {`
	`    public void move() {`
	`        System.out.println("move");`
	`    }`
	`}`

`package p1.sub;`
`public class RenameFile1 {`
`    public void print() {`
`        System.out.println("rename");`
`    }`
`}`
	`package p1.sub;`
	`public class RenameFile2 {`
	`    public void print() {`
	`        System.out.println("rename");`
	`    }`
	`}`

`package p3;`
`public class DeleteFile {`
`    public void print() {`
`        System.out.println("delete");`
`    }`
`}`
	`package p1;`
	`public class AddFile {`
	`    public void newFile() {`
	`        System.out.println("NewFile");`
	`    }`
	`}`

`package p2;`
`public class UpdateStatement {`
`    public void fun1 () {`
`        System.out.println("update");`
`        fun2();`
`    }`
`    public String fun2()`
`    {`
`        return "update";`
`    }`
	`package p2;`
	`public class UpdateStatement {`
	`    public void fun1 () {`
	`        System.out.println("update");`
	`    }`
	`}`

Fig. 4. The source code of the two programs.

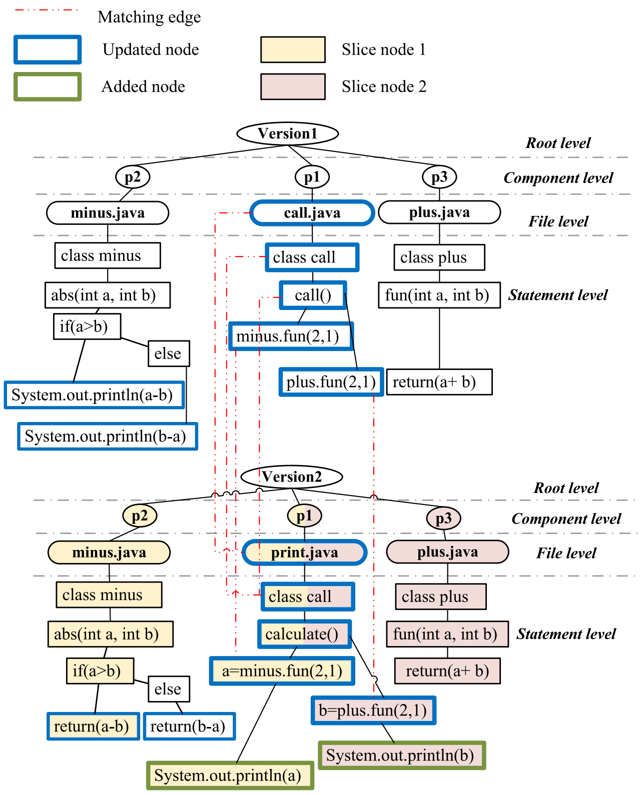


Fig. 5. MATs and the slices of the two programs.

needs to be modified, and the second factor can be calculated based on the full slice. The maintenance difficulty weight denotes the difficulty of co-modification. If two statements are in the same file, The probability of them having the same function and the same related statement set would be higher, then the difficulty of co-changing is lower; While they belong to different components, they may have different functions, and the communication between the two components should be taken into consideration too, so the difficulty of repairing the erosion point may be higher. Another factor is the number of related statements, and it denotes how many statements may be co-modified with the erosion point statement. The more related statements, the more workload is needed. So, the core idea of estimating repair costs is to combine the difficulty and the corresponding workload.

Based on the above analysis, we use Formula (3) to estimate the repair cost of each erosion point, where m_e is the repair cost of the erosion point e. F_e is the statement set contained in the file where e is located, C_e is the statement set contained in the component where e is located, P_e is the statement set of the overall program, S_e is the statement set contained in the slice with e as the slice criterion. If X is a set, $|X|$ is the number of elements contained in X. α, β and γ are the maintenance difficulty weights for the same file, the same component but different files, and different components, respectively.

$$m_e = \alpha * |F_e \cap S_e| + \beta * |(C_e - F_e) \cap S_e| + \gamma * |(P_e - C_e) \cap P_e| \tag{3}$$

The repair cost is a relative value, that is, we use it to evaluate which erosion point needs less cost, and we can repair them first, then we can use less total cost to repair more erosion points.

4. Use case

Here we take an example to illustrate the process of our approach. The first version and the second version are two versions before and after evolution, and the source code is shown in Fig. 4.

For ease of description, in this example, we consider each package as a component, so each program has three components. The function of the first version is that the component p1 invokes the two components p2 and p3 to calculate the sum and the absolute value of 2 and 1. In the evolution process from the first version to the second version, two statements are added in p1 so that p1 can print out the results of the two components of p2 and p3.

Fig. 5 shows MATs and the slices of the two programs, then we will illustrate how to construct and use them in each step.

Step 1 Detecting changes. We construct MATs for the two programs and detect the changed nodes. As shown in Fig. 5, the nodes in the blue border are the updated nodes, and the nodes in the green border are the added nodes. We establish matching edges between changed pair nodes, if a changed node is not connected by a matching edge, that means it is deleted during the evolution process or it is a new node in the new architecture. For example, $plus.fun(2,1)$ is evolved to $b = plus.fun(2,1)$, and the change type is updating. They constitute a changed pair, and there is a matching edge between them.

Step 2 Locating erosion points. Here, we take replaceability and modifiability as indicators to calculate architecture erosion.

Replaceability denotes the difficulty of replacing components (Chhabra et al., 2003). Its calculating formula is shown in Formula (4).

$$Replaceability = \frac{\sum_{i=1}^{N} f(C_i)}{N} \tag{4}$$

where $Replaceability$ is the measurement of replaceability, N is the number of component, C_i denotes the i_{th} component, $f(C_i)$ is the sum of dependency intensity of C_i. Each type of dependency has a corresponding dependency intensity. The dependency intensity is set based on the granularity of the code entity. For example, the dependency intensity of invocation dependency is 0.3, the dependency intensity of reference dependency is 0.4, and the dependency intensity of the implementation dependency is 0.5.

Modifiability describes the capability of SA to be modified in accordance with new requirements (Breivold et al., 2012). The internal

Table 1
The information of experimental cases.

Scale	Program	Version	Size (LOC)
Small	ExcelCompare	0.6.0→0.6.1	1.53K→1.54K
	blitz4j	1.20.0→1.37.0	3.62K→3.97K
	swaggersocket	2.0.0→2.1.0	3.05K→3.25K
	pocketknife	2.0.0→3.2.1	4.98K→6.20K
	Pury	1.0.2→1.0.3	2.68K→3.17K
Large	common-project	2.7.1→2.7.3	294.9K→297.6K
	hdfs-project	2.7.1→2.7.3	400.3K→404.5K
	mapreduce-project	2.7.1→2.7.3	255.7K→259.5K
	tools	2.7.1→2.7.3	111.8K→111.9K
	yarn-project	2.7.1→2.7.3	365.9K→359.9K

deterministic factor of the modifiability is the degree of encapsulation, so the modifiability can be calculated as (Harrison et al., 1998):

$$modifiability = \frac{\sum_{i=1}^{N}\sum_{m=1}^{M_d(C_i)}\left(1-\frac{\sum_{j=1}^{N} f(M_{mi},C_j)}{N-1}\right)}{\sum_{i=1}^{N} M_d(C_i)} \quad (5)$$

where N is the number of components, $C_k (k = 1, 2, \ldots, N)$ is the k_{th} component, M_{mi} is the m_{th} module of the component C_i, $M_d(C_i)$ is the number of declared modules in the component C_i.

$$f(M_{mi}, C_j) = \begin{cases} 1 & j \neq i \wedge \exists E_{ij} \to \{P_{mi}, C_j\} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

where $f(M_{mi}, C_j)$ is the relationship between the module P_{mi} and the component C_j, E_{ij} is an dependence edge between the package M_{mi} and the component C_j. If the component C_j invokes the package P_{mi}, the relationship is 1, otherwise 0.

We calculate the erosion degree of architecture based on the above attributes, then we know $ED(version2, version1)$ is less than 0, so the architecture is eroded.

According to the formulas of the two software architecture attributes, we analyze the reasons for architecture erosion. First, we calculate the measurement of modifiability is decreased. Second, we find that, when the number of components is not changed, the reason for decreasing modifiability is that there are more dependence edges in the new architecture.

According to the changes between the two architectures, we can know that there are two new dependency edges. The first new dependency is from $version2.p2.line5$ to $version2.p1.line4$, so they constitute a changed pair, and because the change causes architecture erosion, $version2.p1.line4$ is an erosion point. The second new dependency is from $version2.p3.line4$ to $version2.p1.line6$, so they constitute a changed pair, and because the change causes architecture erosion, $version2.p1.line6$ is an erosion point.

In this paper, we focus on demonstrating the usage process and effectiveness of EsArCost, so we do not consider specific domains or scenarios and set the same weights for all quality attributes. However, for domain software, one or some quality attributes are more important and can be set relatively high weights based on AHP or experience.

Step 3 Estimating repair costs based on slices. We take $version2.p1.line4$ and $version2.p1.line6$ as the slicing criteria to calculate their full slices.

The full slice of $version2.p1.line4$ contains lines 1 to 5 of version.p1 and lines 1 to 5 of version.p2, so its slice contains 10 statements, that is, there are 10 statements may be co-modified with $version2.p1.line4$. The full slice of $version2.p1.line6$ contains lines 1 to 3, lines 6 and 7 of version2.p1 and lines 1 to 4 of version2.p3, so its slice contains 9 statements, that is, there are 9 statements that may be co-modified with $version2.p1.line6$.

As Fig. 5 shows that the yellow squares denote the nodes contained in the full slice of version2.p1.line4, and the pink squares denote the nodes contained in the full slice of version2.p1.line6. Slice is a statement set, but in the source code there are no statements about the file name, so the file node not used for estimating repair costs. But they still have effects on the erosion point, so we mark them in different colors.

According to Formula (3), we estimate the repair cost of each erosion point. There are many methods to set the weights of Formula (3), such as questionnaire-based hierarchical analysis, validation of historical data, etc. In the example, we set the weights based on our experience. The maintenance difficulty factors for the same file, the same component, and the overall program is 0.2, 0.3 and 0.5, respectively.

When the erosion point e is version2.p1.line4, there are 9 statements in F_e and C_e, and the program has 18 statements. The slice of e contains a total of 10 statements, 5 of which are in the same file and the other statements in other components. The repair cost of $version2.p1.line4$ is estimated as Formula (7).

$$\begin{aligned} m_{\langle \text{version2.p1.1ine4} \rangle} &= \alpha * |F_e \cap S_e| + \beta * |(C_e - F_e) \cap S_e| \\ &+ \gamma * |(P_e - C_e) \cap P_e| \\ &= 0.2 * 5 + 0.3 * 0 + 0.5 * 5 \\ &= 3.5 \end{aligned} \quad (7)$$

When the erosion point e is $version2.p1.line6$, there are 9 statements in F_e and C_e, and the program has 18 statements. The slice of e contains a total of 9 statements, 5 of which are in the same file and the other 4 statements in other components. The repair cost of $version2.p1.line6$ is estimated as Formula (8).

$$\begin{aligned} m_{\langle \text{version2.p1,line6} \rangle} &= \alpha * |F_e \cap S_e| + \beta * |(C_e - F_e) \cap S_e| \\ &+ \gamma * |(P_e - C_e) \cap P_e| \\ &= 0.2 * 5 + 0.3 * 0 + 0.5 * 4 \\ &= 3 \end{aligned} \quad (8)$$

The estimated cost of $version2.p1, line6$ is less than $version2.p1.1ine4$, so V$ersion2.p1, line6$ needs less repair cost. When the total repair cost is limited, we can repair it first.

5. Evaluation

In this section, we evaluate the effectiveness and efficiency of EsArCost and focus on the following research questions:

RQ1: Accuracy of locating architecture erosion points. Can EsArCost locate all erosion points?

RQ2: Effectiveness of estimating repair costs. Can EsArCost effectively estimate repair costs of architecture erosion?

RQ3: Efficiency of EsArCost. How about the efficiency of EsArCost?

In large-scale programs, due to the transferability between statements, the number of statements contained in the slice is relatively large, and it is difficult to verify the accuracy by manual means, so in order to improve the feasibility of manual verification, this paper uses small-scale programs as test cases for accuracy experiments. For the efficiency experiments, in order to reduce the influence of non-core factors such as database connection and database driver, so as to obtain a more realistic efficiency evaluation, the large-scale programs are chosen as the test cases in this paper.

We select five programs that are small-scale and have received many stars from GitHub.com as the small-scale experimental cases and five sub-programs from Hadoop which is a famous distributed system infrastructure as the large-scale experimental cases. The information on experimental cases is shown in Table 1, where the first column denotes the scale of the program, the second column is the program name, the third column denotes the selected versions and the evolution process between them, and the fourth column denotes the size change before and after evolution.

Our approach is implemented on CDG. However, most open source programs do not provide documentation about its architecture, so in

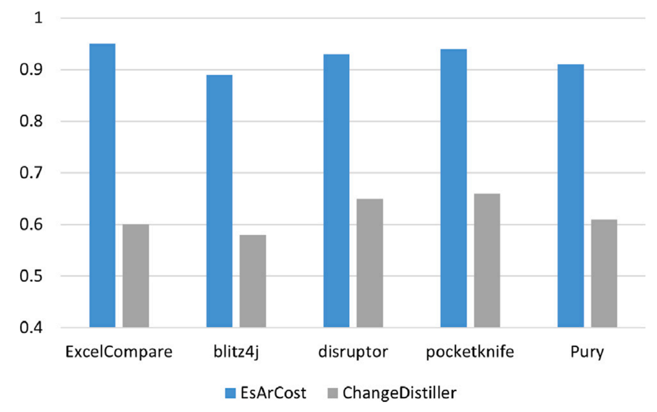


Fig. 6. The comparison of precision between EsArCost and ChangeDistiller.

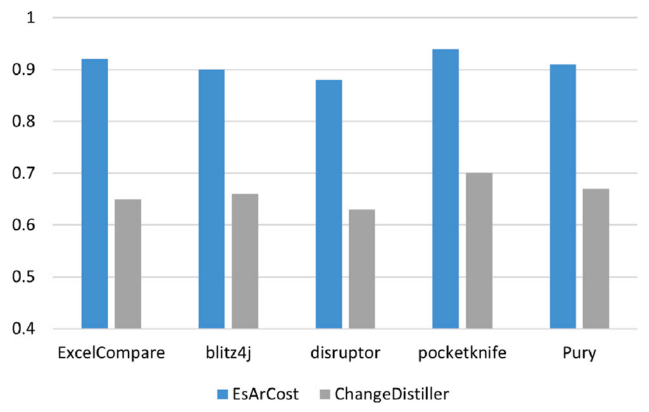


Fig. 7. The comparison of recall between EsArCost and ChangeDistiller.

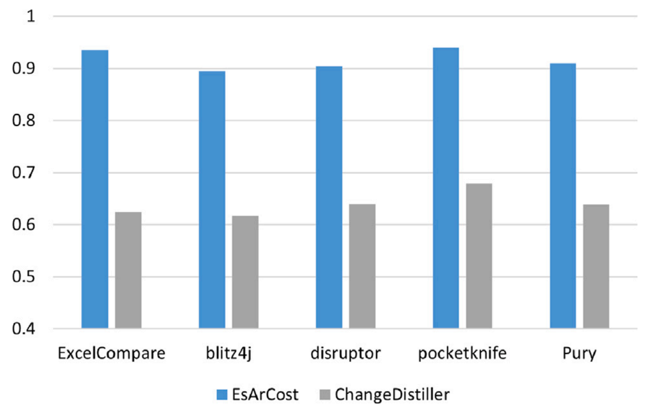


Fig. 8. The comparison of F-score between EsArCost and ChangeDistiller.

our evaluation, we obtain the architecture information and CDG using a software architecture recovery method (Kong et al., 2018) which has higher efficiency and effectiveness. And we analyze architecture erosion as in Section 4 and adopt the same weights.

5.1. RQ1: Accuracy of locating architecture erosion points

The process of locating erosion points is as follows: First, detecting changed pairs; Second, analyzing which changed pairs are the reasons for decreasing architecture qualities. So, the effectiveness of locating erosion points depends on whether EsArCost can detect all changed pairs and whether all changed pairs related to architecture quality degradation can be extracted.

In the terms of detecting changed pairs, we use precision, recall and F-score as the evaluation indicators to evaluate the accuracy of the detecting results.

The precision is used to measure the percentage of actual changes in the detection result. If the precision is low, then there are many wrong items in the detection result. The calculation formula of the precision is shown in Eq. (9), where D represents the set of detected changes, A represents the set of the actual changes, and $common(A, D)$ represents the same set of changes in D and A, $|A|$ is the number of elements of the set A.

$$Precision = \frac{common(A, D)}{|A|} \tag{9}$$

Recall refers to the percentage of detected actual change in all actual change. The low recall indicates that there are many undetected change operations. The calculation formula is shown in Eq. (10).

$$Recall = \frac{common(A, D)}{|D|} \tag{10}$$

F-score is the harmonic average of precision and recall. It indicates the comprehensive capability of change detection. The calculation formula is shown in Eq. (11), where P is the precision and R is the recall.

$$F - score = 2 * \frac{R * P}{R + P} \tag{11}$$

We compare the efficiency between the change detection method of EsArCost and ChangeDistiller (Gall et al., 2009) which is one of the state-of-the-art AST-based change detection tools.

The comparison of precision between EsArCost and ChangeDistiller is shown in Fig. 6. The average precision of EsArCost and ChangeDistiller are 92.4% and 62.0%, that is the correctness of the detection result of EsArCost is higher than that of ChangeDistiller.

The comparison of recall between EsArCost and ChangeDistiller is shown in Fig. 7. The average recall of EsArCost and ChangeDistiller are 91.0% and 66.2%, that is EsArCost detected more actual changes than ChangeDistiller.

The comparison of F-score between EsArCost and ChangeDistiller is shown in Fig. 8. The average F-score of EsArCost and ChangeDistiller are 91.7% and 64.0%, that is the overall accuracy of EsArCost is higher than that of ChangeDistiller.

A change is generated by the evolution between two statements, so the accuracy of locating the changed pair is equal to that of change detection, that is, the overall accuracy of detecting changed pairs is 91.7%. In a word, EsArCost can effectively detect change pairs.

In terms of analyzing which changed pairs are the reasons for decreasing architecture qualities, the architecture qualities are measured based on some internal attributes, such as size, dependency, node, dependence intensity, etc. The changed pair also contains information about internal attributes, so our approach extracts the erosion point from the changed pair set according to which changed pair is related to the internal attributes which are related to the decreasing architecture qualities. In a word, our approach can effectively locate which changed pairs are the erosion point.

5.2. RQ2: Effectiveness of estimating repair costs

A changed code was identified as an erosion point because it changed the architecture and the change caused a degradation in architecture qualities. So if we want to repair an erosion point, we should modify the code to reduce or eliminate the modification of the architecture by the erosion point while guaranteeing that the functionality remains unchanged, that is, code change causes architecture erosion, then we repair architecture erosion by changing code.

Lines of code (LOC) is an important indicator for estimating software costs, so we take how many LOC needs to be modified for changing architecture as the repair cost. Our previous work (Wang et al.,

Table 2
The information about erosion points.

No.	Program	Erosion point	*S_File*	*S_Component*	*S_Other*	Estimated costs	Actual costs
1	blitz4j	NFLockFreeLogger.java, 167	6	0	47	25	20
2		NFLockFreeLogger.java, 216	5	0	203	103	65
3		NFLockFreeLogger.java, 224	26	0	6	8	10
4	disruptor	ExceptionHandlerWrapper.java, 33	12	230	378	260	152
5		BatchEventProcessor.java, 283	42	125	180	136	86
6	ExcelCompare	CellPos.java, 40	24	244	291	224	143
7	pocketknife	BuilderProcessor.java, 150	63	132	293	199	122
8		MethodBinding.java, 15	11	72	18	33	30
9		FooSerializer.java, 8	17	28	0	12	15
10		StringSerializer.java, 8	17	2	0	4	6
11		IntentMethodBinding.java, 48	5	71	358	201	120
12	Pury	StartProfilingAspectTest.java, 66	6	0	6	4	10
13		MethodProfilingAspectTest.java, 143	3	0	326	164	130
14		MethodProfilingAspect.java, 117	32	5	37	26	22
15		SplashActivity.java, 62	11	0	248	126	99

2020) proposes a co-evolutionary method to co-change code based on new architecture, so in this paper, we use the method to calculate how many LOC needs to be modified when we modify architecture.

We list the information about some erosion points in Table 2. The second column is the program name, the third column is the location of the erosion point, the fourth column is the number of slicing statements in the same file *S_File*, the fifth column is the number of slicing statements in the same component but in different files *S_Component*, the sixth column is the number of slicing statements in different components *S_Other*, the seventh and the eighth columns are the estimated costs calculated by our approach and the actual costs, respectively. For example, the No. 1 erosion point comes from *blitz4j*, and it is in the NFLockFreeLogger.java file and locates on line 216. There are 5 slicing statements in NFLockFreeLogger.java and 47 slicing statements in other components. The estimated cost is 47, and the actual cost is 25 code lines.

According to the table, we find that: (i) For erosion points locate in the same program, they have different estimated costs and actual costs. For example, No. 7 to No. 11 locate in the same program, but because the corresponding component functions of them are different, so the slicing statements are different, and the estimate costs are different, (ii) For erosion points located in the same file, they have different estimated costs and actual costs. For example, No. 1 to No. 3 locate in the same file, but because they have different functions, so the slicing statements are different, and the estimated costs are different. In a word, although the statements are in the same program or even in the same file, their functions may be different, so the slices of them and the repair cost are different, and we should estimate the repair cost based on the specifics of each erosion point.

According to Table 2, we draw the scatter plot of #slice, estimated costs and actual costs, where #slice is the number of statements contained in the slice. According to the figure, we can find that there is a relatively obvious correlation between them, that is, although the estimated cost is a relative value, we can use the estimated cost to presume which erosion point needs the least repair cost.

According to Fig. 9 we find that: (i) When #slice is large, the difference between the estimated cost and the actual cost is less than the difference between the estimated costs and #slice, that is because most of the statements that need to be co-modified are those associated with dependencies. When no file dependencies or component dependencies are involved, the possibility of co-evolution is less; (ii) When #slice is small, The values of these three indicators are almost identical, that is because although the statements are not related to dependency, they are closely associated with the erosion point, so they all need be co-modified. In a word, the closely associated statements and the related dependency statements are more likely to be co-modified.

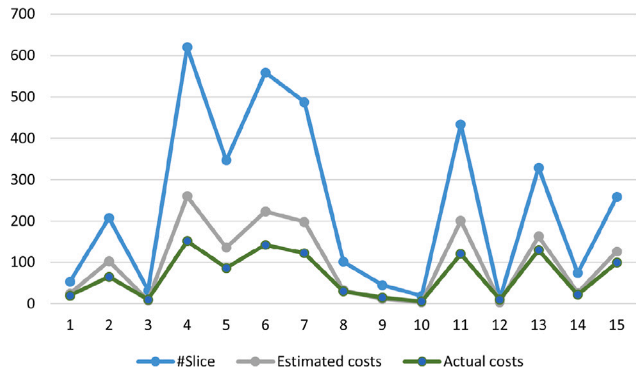


Fig. 9. The correlation between #slice, estimated costs and actual costs.

Table 3
The time of detecting changes.

Program	Constructing MAT(s)	Matching time (s)
common-project	15.572	20.707
hdfs-project	20.588	36.982
mapreduce-project	13.374	18.28
tools	6.281	7.428
yarn-project	21.017	38.32

5.3. RQ3: Efficiency of EsArCost

The time consumed by EsArCost mainly consists of two parts, locating erosion points and calculating slices, so we evaluate the efficiency of EsArCost based on the above parts.

The time consumed in locating erosion points mainly depends on the efficiency of detecting changes. The time of detecting changes depends on two parts, the construction time of MAT and the matching time. The time of detecting changes in large-scale programs is shown in Table 3. As the table shows, the large program has 40K code lines, and our approach can detect changes in 1 min, so the time consumed is acceptable.

We use EsArCost and ChangeDistiller to perform change detection on different open-source projects and summarize the time consumed. ChangeDistiller only detect changes between files, so we input pairs of changed files according to the recorded information in GitHub to calculate the sum of the time consumed. The time consumed is shown in Fig. 10.

As shown in Fig. 10, the time consumed of EsArCost is better than that of ChangeDistiller, which shows that the change detection method

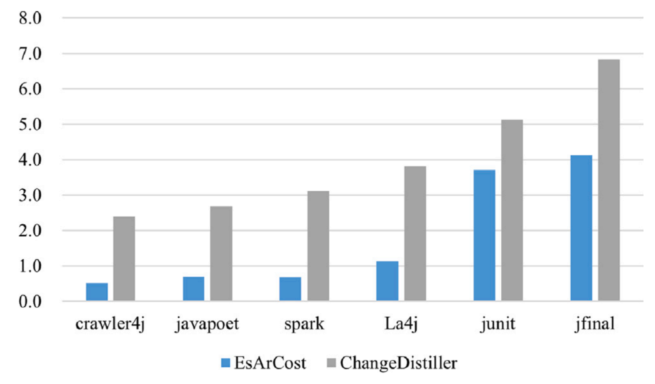


Fig. 10. The comparison of time consumed between EsArCost and ChangeDistiller.

of EsArCost has higher efficiency compared to the traditional AST-based change detection method.

In terms of performance overhead, the change detection method based on AST matching is weaker than that based on textual matching. The reason is that the minimum analysis granularity of the former is the fine-grained code change of the statement level and variable level, while the analysis granularity of the latter is the text line, that is, the granularity of matching objects of the AST-based matching method is finer than that of textual matching.

The AST-based matching process of the traditional change detection method based on AST adopts Chawathe's structured information difference analysis algorithm (Chawathe et al., 1996). That is to match the similarity of leaf nodes and non-leaf nodes from bottom to top, and finally obtain the matching set.

By analyzing the source code of the evolution software, it can be found that, between two adjacent versions, the unchanged source code always occupies the majority, and the changed source code only occupies a small part. The top-down matching process can quickly match the isomorphic subtrees between two MATs, and it greatly reduces the number of matching nodes of the bottom-up matching process. At the same time, because a large number of anchor matching nodes fix the entire frames of the two trees, the bottom-up matching process of the remaining nodes can achieve local matching within the container, thereby avoiding the global matching efficiency problem. Therefore, the temporal performance of the algorithm is better than the AST matching algorithm.

Further analysis of the time complexity of the two matching algorithms, the other $n = max(|T_1|, |T_2|)$, the time complexity of the AST matching algorithm is $O(N^2)$. For the two-step MPAT matching algorithm in this paper, we first consider the worst case. When the top-down anchor matching process does not find any isomorphic nodes, the algorithm degenerates into a traditional AST matching algorithm. The worst-case time complexity is $O(N^2)$. Fortunately, this situation is almost impossible, and the top-down anchor matching process can often quickly detect the part of the isomorphic node cluster with a higher proportion of nodes in the two trees. Therefore, the average time complexity is far below $O(N^2)$.

In a word, the experimental results show that the change detection method of EsArCost has higher efficiency than ChangeDistiller for these projects, and compared with the AST-based matching method, the complexity of our algorithm is better.

The time of calculating slices depends on two parts, the time of constructing SDG and the execution time of the slicing algorithm. If a statement is in a central component, it may have more related statements than other statements, so the difference in slice size is very large. At the same time, the corresponding time of executing the slicing algorithm is different. To obtain a relatively objective result, we randomly select ten statements as slicing criteria and summarize their time consumed. The summarization time of calculating slices is shown in Table 4, where the first column is the program name, the second column denotes the time consumed in constructing SDG, the third column denotes the average size of slices, and the fourth column is the average time of executing the slicing algorithm.

Table 4
The time of calculating slices.

Program	Constructing SDG(s)	Scale (LOC)	Executing algorithm(s)
common-project	6176.008	1039	0.16
hdfs-project	3262.908	6526	0.40
mapreduce-project	1048.705	3513	0.24
tools	1841.107	7682	0.77
yarn-project	11 191.672	8460	0.74

6. Related work

In this paper, we propose an approach to resolve the following questions. First, how to detect architecture changes; Second, how to locate architecture erosion points; Third, how to estimate repair costs of architecture erosion.

To the best of our knowledge, none of the existing approaches addresses all the above issues, so we analyze related work from the above three aspects.

In the aspect of detecting architecture changes, the change detection method we use is an extension of our previous work (Wang et al., 2019). The change detection method can be divided into two categories, which are the change detection method based on text matching and the change detection method based on AST (abstract syntax tree) matching. Some version control systems are implemented based on the first method, such as SVN, Unix diff and Ldiff (Canfora et al., 2009). However, the file name and the file path are matched before statements, and no information about the syntactic structure is explicitly stored, so it cannot identify renaming (Maletic and Collard, 2004). The second method converts the source program into ASTs (Philippsen and Philippsen, 2016; Jiang et al., 2017), and calculates the edit scripts of two ASTs using the set of matched nodes. However, it cannot analyze overall programs.

In the aspect of locating the erosion points, most research analyzes architecture erosion based on one certain version. There are two main types of indicators, the inconsistency between the requirements and the actual implementation and the improper architecture pattern. Medvidovic N (2003) use inconsistency as the indicator. Zhang et al. (2011) detect architecture erosion of architectural patterns by the design decision. Herold et al. (Herold and Rausch, 2013) detect architecture erosion of architecture patterns by the common ontology. However, the above effects of architecture erosion all threaten the architecture quality, that is, avoiding the effects on software quality is the ultimate goal. So, in our method, we use the architecture quality as the indicator to detect architecture erosion.

In the aspect of estimating repair costs, most research focuses on estimating software-related costs, rather than software architecture. However, there is a significant difference between software and software architecture which causes the estimating methods of software is not applicable to software architecture. The difference is that software architecture is the high abstraction level of software, in the process of layer-by-layer abstraction, the details related to statements that are not important for the architectural level are ignored. Thus, for the overall quality of the software, a statement needs to be modified, but for the software architecture, the information related to the statement is not reflected at the architectural level, so it is not even an erosion point.

In addition to the above aspects, our approach estimates the repair cost of architecture erosion, and it can be used to provide the repair priority when the total cost is limited.

7. Threats to validity

Construct validity. In our example, we set the maintenance difficulty factors for the same file, the same component, and the overall program based on experience, and these weights are not universal, and setting weights based on historical data is a more scientific approach. But in this paper, we focus on proposing the method of estimating repair costs, and our weights can also be used to estimate the approximate repair costs, because they are consistent with the actual situation. We quantify the erosion degree based on quantified quality attributes, so the calculation of our approach needs quantified information, such as structural information about the software architecture, information about the properties of code entities, etc. In future work, we can set the weights by mining historical data, then the estimated results can be approximately equal to the actual cost.

Internal validity. In this paper, we refer to the methods for calculating replaceability and modifiability, and there may be some different approaches or disagreements about how to calculate them. But the focus of our approach is on how to find an erosion point and estimate its repair cost based on measurement formulas, and our approach can also be applied to calculate architecture erosion and estimate repair costs based on other architecture qualities.

External validity. In our example, we calculate architecture erosion based on replaceability and modifiability, and they are measured by quantitative methods. This type of measurement method has good generality and objectivity. However, how our approach locates erosion points and estimates erosion costs based on a qualitative architectural measurement method needs further investigation.

8. Conclusion

In this paper, we propose an approach called EsArCost. To the best of our knowledge, EsArCost is the first approach proposed for estimating the repair cost of architecture erosion. In our approach, we estimate repair cost based on statements, because unreasonable code changes cause architecture erosion, and the number of code lines is the important factor in determining repair cost. When the repair cost is limited, developers can repair some erosion points which need less repair cost.

The erosion degree describes the overall erosion of the architecture, and it does not represent the severity of each erosion point, so as of our future work, based on EsArCost, we plan to quantitative the severity of the erosion point, and the severity as the benefit from repairing erosion points, then we establish the cost–benefit of repairing erosion points.

CRediT authorship contribution statement

Tong Wang: Conceptualization, Methodology, Software, Writing – original draft. **BiXin Li:** Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported partially by the National Natural Science Foundation of China (No. 62302012), the University Natural Science Research Project of Anhui Province, China (No. 2023AH051117), and the Natural Science Foundation of Anhui Province, China (No. 2108085QF263).

References

Allian, A.P., Sena, B., Nakagawa, E.Y., 2019. Evaluating variability at the software architecture level: An overview. In: Proceedings of the 34th ACM/SIGAPP symposium on applied computing. pp. 2354–2361.

Behnamghader, P., Le, D.M., Garcia, J., Link, D., Shahbazian, A., Medvidovic, N., 2017. A large-scale study of architectural evolution in open-source software systems. Empir. Softw. Eng. 22 (3), 1146–1193.

Bhat, M., Shumaiev, K., Hohenstein, U., Biesdorf, A., Matthes, F., 2020. The evolution of architectural decision making as a key focus area of software architecture research: A semi-systematic literature study. In: 2020 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 69–80.

Breivold, H.P., Crnkovic, I., Larsson, M., 2012. Software architecture evolution through evolvability analysis. J. Syst. Softw. 85 (11), 2574–2592.

Canfora, G., Cerulo, L., Penta, M.D., 2009. Ldiff:An enhanced line differencing tool. In: IEEE International Conference on Software Engineering. pp. 595–598.

Chatzipetrou, P., 2019. Software cost estimation: A state-of-the-art statistical and visualization approach for missing data. Int. J. Serv. Sci. Manag. Eng. Technol. 10 (3), 14–31.

Chawathe, S.S., Rajaraman, A., Garciamolina, H., Widom, J., 1996. Change detection in hierarchically structured information. Acm Sigmod Rec. 25 (2), 493–504.

Chhabra, J.K., Aggarwal, K., Singh, Y., 2003. Code and data spatial complexity: two important software understandability measures. Inf. Softw. Technol. 45 (8), 539–546.

Gall, H.C., Fluri, B., Pinzger, M., 2009. Change analysis with evolizer and ChangeDistiller. IEEE Softw. 26 (1), 26–33.

Harrison, R., Counsell, S.J., Nithi, R.V., 1998. An evaluation of the MOOD set of object-oriented software metrics. IEEE Trans. Softw. Eng. 24 (6), 491–496.

Herold, S., Rausch, A., 2013. Complementing model-driven development for the detection of software architecture erosion. In: Proceedings of the 5th International Workshop on Modeling in Software Engineering. pp. 24–30.

Hohpe, G., Ozkaya, I., Zdun, U., Zimmermann, O., 2016. The software architect's role in the digital age. IEEE Softw. 33 (6), 30–39.

Huang, J., Li, Y.F., Keung, J.W., Yu, Y.T., Chan, W., 2017. An empirical analysis of three-stage data-preprocessing for analogy-based software effort estimation on the ISBSG data. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 442–449.

Jiang, Q., Peng, X., Wang, H., Xing, Z., Zhao, W., 2017. Understanding systematic and collaborative code changes by mining evolutionary trajectory patterns. J. Softw. Evol. Process 29 (3), e1840.

Kadri, S., Aouag, S., Hedjazi, D., 2020. Multi-view model-driven projection to facilitate the control of the evolution and quality of the architecture. Int. J. Softw. Innov. (IJSI) 8 (4), 21–39.

Kadri, S., Aouag, S., Hedjazi, D., 2021. MS-QuAAF: A generic evaluation framework for monitoring software architecture quality. Inf. Softw. Technol. 140, 106713.

Knieke, C., Rausch, A., Schindler, M., 2021. Tackling software architecture erosion: Joint architecture and implementation repairing by a knowledge-based approach. In: 2021 IEEE/ACM International Workshop on Automated Program Repair. APR, IEEE, pp. 19–20.

Kong, X., Li, B., Wang, L., Wu, W., 2018. Directory-based dependency processing for software architecture recovery. IEEE Access 6, 52321–52335.

Kuan, S., 2017. Factors on software effort estimation. Int. J. Softw. Eng. Appl. 8 (1), 23–32.

Li, R., Liang, P., Soliman, M., Avgeriou, P., 2022a. Understanding software architecture erosion: A systematic mapping study. J. Softw. Evol. Process 34 (3), e2423.

Li, R., Soliman, M., Liang, P., Avgeriou, P., 2022b. Symptoms of architecture erosion in code reviews: a study of two OpenStack projects. In: 2022 IEEE 19th International Conference on Software Architecture. ICSA, IEEE, pp. 24–35.

Lulu, W., Bixin, L., Xianglong, K., 2020. Type slicing: An accurate object oriented slicing based on sub-statement level dependence graph. Inf. Softw. Technol. 127, 106369.

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidović, N., Kroeger, R., 2017. Measuring the impact of code dependencies on software architecture recovery techniques. IEEE Trans. Softw. Eng. 44 (2), 159–181.

Maletic, J.I., Collard, M.L., 2004. Supporting source code difference analysis. In: IEEE International Conference on Software Maintenance. pp. 210–219.

Medvidovic N, G.P., 2003. Stemming architectural erosion by coupling architectural discovery and recovery. In: Proceedings of the 2nd International Software Requirements to Architectures Workshop. pp. 61–68.

Mitra, D., Arora, M., Rakhra, M., Kumar, C.R., Reddy, M.L., Reddy, S.P.K., Kumar, C., Shabaz, M., 2021. A hybrid framework to control software architecture erosion for addressing maintenance issues. Ann. Romanian Soc. Cell Biol. 2974–2989.

Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C., Harman, M., 2017. Are developers aware of the architectural impact of their changes? In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 95–105.

Philippsen, M., Philippsen, M., 2016. Move-optimized source code tree differencing. In: ACM/IEEE International Conference on Automated Software Engineering. pp. 660–671.

Pospieszny, P., 2017. Software estimation: Towards prescriptive analytics. In: Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement. pp. 221–226.

Pospieszny, P., Czarnacka-Chrobot, B., Kobylinski, A., 2018. An effective approach for software project effort and duration estimation with machine learning algorithms. J. Syst. Softw. 137, 184–196.

Wang, T., Li, B., Zhu, L., 2020. A co-evolutionary method between architecture and code.. In: International Conference on Software Engineering and Knowledge Engineering. pp. 147–152.

Wang, T., Wang, D., Zhou, Y., Li, B., 2019. Software multiple-level change detection based on two-step MPAT matching. In: The 26th International Conference on Software Analysis, Evolution and Reengineering. pp. 4–14.

Zhang, L., Sun, Y., Song, H., Chauvel, F., Mei, H., 2011. Detecting architecture erosion by design decision of architectural pattern. In: SEKE. pp. 758–763.

Tong Wang is a Doctoral Researcher at Anhui University of Technology. She received the Ph.D. degree in software engineering from Southeast University. Her main research focuses on software maintenance with AI, particularly in the software architecture evolution.

Bixin Li is a full Professor at the Southeast University, Nanjing, China. He received the Ph.D. degree in computer science from Nanjing University. His main research focuses on software evolution and maintenance.