An improved mapping method for automated consistency check between software architecture and source code

Fangwei Chen
Beihang University
Beijing, China
chiefeweight@sina.com

Li Zhang
Beihang University
Beijing, China
lily@buaa.edu.cn

Xiaoli Lian
Beihang University
Beijing, China
lianxiaoli@buaa.edu.cn

Abstract—In daily software development, inconsistencies between architecture and code inevitably occur with the continuous contribution, even under model-driven development which can trace between design and code. Many methods have been proposed for consistency checking, but most require huge human efforts on establishing the mappings between architectural and code elements. Besides, the multi-layered architecture and code increases the difficulties in inconsistency detection, while existing algorithms do not handle this well. Thus, we propose an improved mapping method for automated consistency check between software architecture and Java implementation, with the premises that initial tracing between architecture and code are established. To be specific, during software evolution, our method can automatically re-establish the mappings between architecture and code using initial tracing information. Then, with detailed inconsistency check rules, we detect the inconsistencies heuristically. Experiments with two projects show our method's high effectiveness with more than 98% of recall and 96% of precision.

Index Terms—consistency checking, software architecture, Java implementation

I. Introduction

As the scale of software is becoming larger, and the architecture is becoming more complex, many projects are often faced with the problem of inconsistencies between the architecture design and code [30], which is called the architectural drift [23]. This is true for both model-driven development and other traditional software development processes. In a typical model-driven software development process, the project's architecture design is described using Domain-Specific Language (DSL) which will be automatically transformed into code skeletons [7]–[10]. Developers need to manually add some code into the code skeletons [33]. Usually new classes, packages or even new relationships not defined in the architecture design will be added, which introduce inconsistencies [30]. Furthermore, even after the project is completed, both the architecture design and source code would evolve [34], and once the other was not synchronized with the evolved one in time, inconsistencies appear. As a result, the maintenance tasks become complicated and costly [31]. Other traditional software development processes will be more likely to introduce inconsistencies since there are no models guiding the development.

So, the inconsistency detection is critical. Currently, the related research strands mainly follow the similar consistency check steps, as shown in Fig.1: first, extract the elements and relationships from architecture design and code and record them in a structure (called "the analysis graph"). Second, establish the mapping between the elements in these two extracted structures manually or semi-automatically. Third, check the consistency based on the mapping. The mapping method determines the approach's feasibility and performance mostly, where the MDA-focused approaches excel by taking advantage of the model transformation for tracing from code elements to architectural compositions [3]. Furthermore, the Java implementation are convenient for establishing the initial mapping since the structure and elements in the architecture design and the code are relatively similar to each other comparing with other languages. However, there are still some limitations to be improved: (1) Since the structure of architecture design and code are usually multi-layered, the hierarchy change of architectural or code elements should be taken into account to cover more inconsistencies. Currently there are several algorithms to address the hierarchical mapping establishment, e.g., MQAttract [4], CountAttract [12], etc. These algorithms can effectively deal with some of the mapping situations, while there still exist some scenarios that need to be covered, such as the mapping problems caused by the complex dependency between elements' children. (2) The relationship inconsistencies that are checked by most of the current approaches need to be refined into more detailed relationship inconsistencies, in order to improve the specificity of inconsistency identification.

Therefore, the main goal of this paper is to improve the current approaches' performance and capability by improving the mapping establishment. In this study, we propose an improved mapping method for automatically identifying the inconsistencies between architecture model and Java implementation on the basis that initial mapping

Funding for this work has been provided by the National Science Foundation of China Grant No. 61672078 and 61732019.

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00021

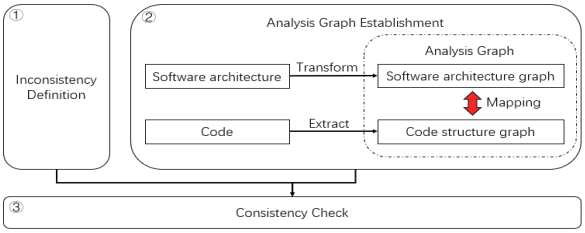

Fig. 1: Process of the Reflexion Model Technique

between the architecture and code are established. For the initial mapping establishment, we use a predefined Architecture Model Description Language (AMDL) that extends the UML to describe the elements and relationships in an architecture design, and then we generate Java code as long as the comments keeping the simple tracing information from the architecture design elements to code elements (including packages, classes and interfaces). After software evolvement, we establish an analysis graph with the high-level graph extracted from the AMDL model, the low-level graph extracted from Java implementation, and the mappings established automatically from the model-driven generated code comments. Then for the remaining unmapped elements, we propose the Improved CountAttract Mapping algorithm which heuristically establishes the mappings based on the already mapped elements. When all mappings are established, based on the inconsistency check rules, we perform the consistency checking based on the analysis graph.

The rest of this paper is structured as follows. In Section II we provide a brief review of the consistency checking approaches based on RM and our work's background. In Section III we describe our automated and hierarchical consistency checking approach. To evaluate the capability and performance of our approach, we introduce the experimental setup, present the results, and analyze them to evaluate how does our approach perform in Section IV. Finally, in Section V we conclude our work and point out the future work.

II. The Related Work and Background

To address the architectural drift problem which will impact the maintenance of system as well as the software evolution, many consistency checking approaches between architecture and code have been proposed. We briefly review the current studies on consistency checking, and then figure out what can be improved. The background of our work is also introduced.

A. The Related Work

Technically, current major consistency checking approaches can be categorized into the following types [3], [24]:

(1) Manual consistency check. Manually evaluate the factors that may cause inconsistencies between architecture design and code [3]. The efficiency of this type of approaches is very low, and the check results are error-prone.
(2) Dependency Structure Matrix [25] based consistency check.The consistency check between architecture design and code can be transformed into the comparison between two matrices, which can record the class, package and their relationships. Thus once the matrices are built, the consistency check can be performed automatically.
(3) Reflexion Model based consistency check. Reflexion Model (RM) was first proposed by Murphy et al. [1]. The prerequisite of this approach is the existence of the architecture design described with a modeling language, and the source code. Then, a structure used for mapping the software architecture and the code should be established [16]. The consistency check can be automatically done based on the mapping and inconsistency check rules.

It is argued that the Reflexion Model technique is very effective in consistency check [5], [25]. Thus, most consistency checking approaches are based upon the Reflexion Model technique [1], such as Ali et al. [35] and Buckley et al. [36], [37], along with tools like JITTAC [19], [27], SAVELife [26], and ARTOS [28], etc. However, many approaches still need the manual mapping establishment [6]. It is very challenging to build the mapping during software evolution for three primary reasons: 1) the code changes may be in multiple forms (such as the addition, deletion, name changing, and the hierarchy changing); 2) the relationships and structures of classes in both architecture design and code are always complex; 3) the hints about the mappings between architecture design and code are limited.

To address the mapping establishment problem, MQAttract [4] and CountAttract [12] exceed many other algorithms, especially in the hierarchical mapping establishment. Through the following analysis, we choose the CountAttract algorithm and improve it to establish the mappings between composited components:

1) Clustering algorithms are considered as an effective way dealing with component and package mappings

during the reflexion process [13], [14]. The CountAttract algorithm is an effective clustering algorithm.
2) Many algorithms cluster all elements in one iteration [14], which may impact the mapping accuracy. The CountAttract algorithm can be improved to address this problem.
3) Current CountAttract algorithm [11], [12] calculates the Attraction (We call the possibility of mapping low-level elements and high-level elements as Attraction) only based on the dependency between unmapped and mapped low-level elements [15]. It does not consider the dependency of each element's children, which may cause a mapping problem in the following scenario: we have already mapped a low-level element (e.g., a package) to a high-level element (e.g., a component), and the majority of dependencies between this package and other packages belong to very few children of this package (i.e., one or two classes). When these children are moved from the origin package to a new package, the origin package's attraction to its mapped component will decrease while the new package's attraction will increase, which may cause the calculated mapping to change due to the attraction calculation. However, neither do the hierarchy of packages in the code nor the hierarchy of components in the architecture design change, so the actual mapping should not change either.

Based on these reasons, we propose an Improved CountAttract Mapping, an automatic and incremental mapping algorithm by improving the existing methods using the existing mappings and heuristic information in the model-driven software development. Our proposed algorithm will be detailedly explained later.

B. Background

UML component diagram and class diagram are often used for architecture designing, but each diagram cannot cover all aspects of the architecture, for example, the component diagram demonstrates the module division of the system but lacks of details, while the class diagram shows the details of one module but lacks of external dependencies. Therefore, in our previous work, we defined an architecture modeling language Architecture Model Description Language (AMDL) which contains several UML elements essential to the architecture design, such as system, component, class and interface. The meta-model of AMDL is shown in Fig.2. We also designed to transform the architecture model into Java code skeletons.

We built a prototype to support all essential steps of model-driven software development, including the architecture design, the Java code skeleton generation with comments for tracing between high-level design element and low-level code implementation.

III. Our Approach

We propose a heuristic hierarchical consistency checking approach by improving the mapping algorithm between the software architecture and the Java implementation, based on the premise that initial tracing between design and code are established. The input of our approach consists of three parts: (1) an architecture design in the form of AMDL which combines the UML component diagram and class diagram; (2) the source code developed from the code skeleton that is generated from the architecture design; (3) the automatically created mappings between the architectural elements and the generated code elements. The output is a list of inconsistencies.

Following the Reflexion Model [1] theory, the procedure of our approach is three-folds: First, we extract the overview of the architecture design (specified using AMDL) as the high-level analysis graph, and the overview of the code as the low-level analysis graph. Second, we establish the complete analysis graph for consistency check by building the mappings between elements in these two graphs. Since the code skeleton records the tracing information from architectural elements, mappings between these elements can be automatically established, leaving all manually changed elements to be mapped. We analyzed that the following manual development may cause architectural inconsistency: the addition, deletion, name or hierarchy change of an element, and the relationship change between elements. Thus, we propose an improved mapping method for these manually changed elements based on the clues provided by the mapped elements. Third, once all architectural and code elements are mapped, the consistency check can be easily done through performing inconsistency identification.

We introduce our approach in the following parts. In Section III-A we define the check rules for inconsistency identification. In Section III-B we introduce the automatic analysis graph establishing method using the traces generated automatically during MDA and our proposed heuristics. In this part, we propose an improved CountAttract algorithm to complete the analysis graph. In Section III-C we explain the hierarchical consistency checking process.

A. Inconsistency Check Rules

The inconsistency check rules are critical for the consistency check. In this section we define the check rules based on traditional checking rules [16], [30], [32], which often have two parts: absence and divergence. An absence inconsistency is defined as an architecture element or relationship does not have a corresponding reference in code, while a divergence inconsistency is defined as a code element or relationship does not have a corresponding reference in architecture design. However, these rules cannot distinguish some specific changes like the code element's addition, deletion and hierarchical change, so they need to be specified for architect and practitioners' favor in detail. Based on traditional checking rules, we define

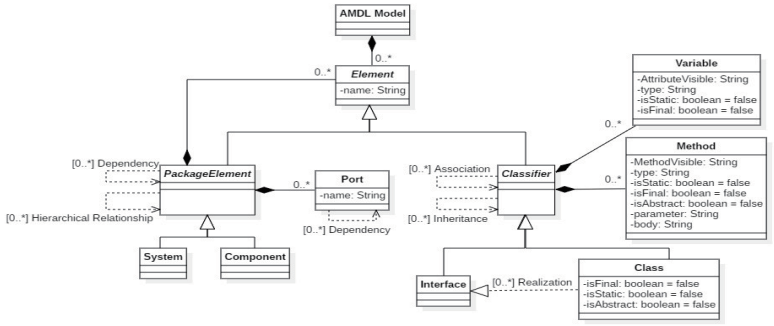

Fig. 2: AMDL Meta Model

all inconsistencies that we mean to check in two parts: element inconsistencies and relationship inconsistencies.

In this paper, to be clear, we call architecture element as high-level element, and call code element as low-level element, so as relationships. A high-level element may represent a system, a component, a class or an interface in the architecture design. A low-level element may represent a package, a class or an interface in the code. All these definitions are listed in Table. I:

1) Element Inconsistency Check Rules: Each of these inconsistencies corresponds to an element changing scenario: element absence, element divergence, low-level element's name changing, low-level element's hierarchy changing, and the changing both on low-level element's name and hierarchy. We detailedly explain each inconsistency check rule in Table.II.

2) Relationship Inconsistency Check Rules: These inconsistencies each corresponds to a relationship changing scenario: relationship absence, and relationship divergence, as shown in Table.III.

B. Analysis Graph Establishing Method

In order to automatically check the consistency between the architecture design and the code, taking into account the complexity and difference of elements and relationships in both the architecture design and the codes, most consistency checking approaches extracted an analysis graph. The analysis graph is a middle structure that contains not only the structural information of both architecture design and code, but also their mappings. An example of an analysis graph is shown in Fig.3. Once the analysis graph is completed, the consistency checking can be easily done according to the mapping. In this section, we firstly introduce our analysis graph. Then we explain the analysis graph establishing method, including elements and relationships extraction and automated mapping.

1) Analysis Graph Composition: Our analysis graph contains all elements from the architecture design and the code, and all types of relationships between elements, as listed in Table. I.

2) Analysis Graph Establishment: Once the composition of the analysis graph is determined, the analysis graph can be established from the architecture design and the code. Our analysis graph establishment method can be divided into two parts: elements and relationships extraction, and automated reflexion model construction based on initial mapping.

a) Elements and relationships extraction: This step is used for extracting all elements and relationships from architecture design and code. We propose two algorithms for extraction from these artifacts: architecture design parse using Dom4j, and reverse engineering from source code using JavaParser and MoDisco.

In our previous tool, the architecture design is recorded in XML file. We extract the architecture design elements and relationships in the architecture design file, and record each high-level element's type, name and parent, and each relationship's source and target elements, All architectural elements and relationships are transformed into analysis graph elements respectively.

We use JavaParser and MoDisco to extract classes, interfaces, packages and their relationships from the code. For each code element, we record its type, name and parent, and all types of relationships related with this element. For each package, we also record its children. All code elements and relationships are transformed into its corresponding elements and relationships in the analysis graph.

b) Improved CountAttract Mapping for automatic mapping establishment based on existing mapping: After all elements and relationships are extracted, there should be mappings between architecture and code elements. Based on the reasons explained in Section II-A, we propose the Improved CountAttract Mapping, an automatic and incremental mapping algorithm by improving the existing methods using existing mappings and heuristic information obtained during the model-driven software development.

In order to introduce our algorithm clearly, we use relational algebra to describe all elements and their relationships in the analysis graph. HG (High-level Graph)

TABLE I: High-level and Low-level Definitions

Analysis Graph Composition	Description
High-level element	system, component, class and interface
Low-level element	package, class and interface
Relationships of element	Association, Inheritance and Realization are used to analyze the relationship between classes or interfaces. Dependency is used to analyze the relationship between system and component in the architecture design, and package in the code. Hierarchical Relationship is used to demonstrate the hierarchy between architectural elements or code elements. Mapping is used to map the high-level elements and their corresponding low-level elements.

TABLE II: Element Inconsistency Check Rules

Inconsistency	Description
Element absence	a high-level element cannot be mapped to any low-level element
Element divergence	a low-level element cannot be mapped to any high-level element
Low-level element's name change	a high-level element and its mapped low-level element which is at the same layer have different element names. By saying "at the same layer", we mean that the high-level element can be mapped to a low-level element, and both their direct parent nodes can be mapped, too.
Low-level element's hierarchy change	a high-level element and its mapped low-level element which has the same element name are at different layers. By saying "at different layers", we mean that the architecture element can be mapped to a low-level element, but their direct parent nodes cannot be mapped.
Low-level element's name and hierarchy change	a high-level element and its mapped low-level element have different names, and they are at different layers

TABLE III: Relationship Inconsistency Check Rules

Inconsistency	Description
Relationship absence	a high-level relationship cannot be mapped to any low-level relationship
Relationship divergence	a low-level relationship cannot be mapped to any high-level relationship

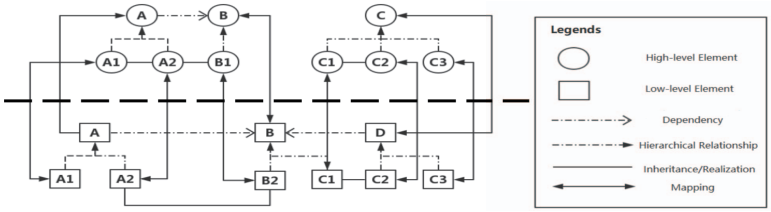

Fig. 3: An Example of Analysis Graph

represents architecture design which is composed of high-level elements (represented as E_H) and their relationships (represented as R_H) in the analysis graph. LG (Low-level Graph) represents code which is composed of low-level elements (represented as E_L) and their relationships (represented as R_L) in the analysis graph. These symbols are listed in (1):

$$\begin{aligned} HG &= (E_H, R_H) \quad & LG &= (E_L, R_L) \\ E_H &= \{h_1, ..., h_n\} \quad & E_L &= \{l_1, ..., l_n\} \\ R_H &= E_H \times E_H \quad & R_L &= E_L \times E_L \end{aligned} \tag{1}$$

$P_H(E_H)$ represents the hierarchy between high-level elements, as shown in (2). $P_H(h_1) = h_2$ means that the high-level element h_2 is the direct parent of high-level element h_1. Similarly, we use $P_L(E_L)$ to represent the hierarchy between low-level elements. Equation (2) can be used iteratively several times, for example, $P_H(P_H(h_1)) = h_3$ means that the high-level element h_3 is the parent of the parent of high-level element h_1. We simply mark this as $P_H^2(h_1) = h_3$. The same is true with the low-level equation.

$$\begin{aligned} P_H(E_H) &: E_H \to E_H \quad & P_H^n(E_H) &: E_H \to E_H \quad & n \in \mathbb{N}_+ \\ P_L(E_L) &: E_L \to E_L \quad & P_L^n(E_L) &: E_L \to E_L \quad & n \in \mathbb{N}_+ \end{aligned} \tag{2}$$

$Map(E_L)$ represents the mapping between high-level elements and low-level elements, for example, $Map(l_1) = h_1$ represents that the low-level element l_1 is mapped to the high level element h_1. This relation is shown in (3).

$$Map(E_L) : E_L \to E_H \tag{3}$$

Based on the relational algebra above, we can explain our algorithm for constructing the mappings with two steps: firstly, we try to map a low-level element to a high-level element using the high-level element's children's information. If the information are not enough to make mapping decisions, then secondly, we improve the CountAttract algorithm to calculate the possibility of mapping the unmapped low-level elements and the high-level elements by heuristically using the information of the children of high-level and low-level elements, and decide

the mappings based on calculation result. Our improved algorithm can deal with the scenario we mentioned before where the current CountAttract algorithm may cause problems.

At the first step of our algorithm, we use inner-similarity to map low-level elements to high-level elements. The inner-similarity between a low-level element and a high-level element is defined as the ratio of high-level element's children which are mapped to a low-level element's child to all children of the high-level element. When the ratio is higher than a predetermined threshold, we believe that a mapping should be established between these two elements. Equation (4), (5) and (6) show the first step of out algorithm.

$$\eta(l_i, h_j) = \frac{|Children_{mapped}(h_j, l_i)|}{|Children(h_j)|} \tag{4}$$

$$\begin{aligned} Children_{mapped}(h_j, l_i) =& \{h \mid C_H(h, h_j) \wedge \\ & \exists l_k \in E_L, C_L(l_k, l_i) \\ & \text{and } Map(l_k) = h\} \\ Children(h_j) =& \{h \mid C_H(h, h_j)\} \\ C_H(h_i, h_j) \text{ is true } \iff & \exists n \in \mathtt{N}_+, P_H^n(h_i) = h_j \\ C_L(l_i, l_j) \text{ is true } \iff & \exists n \in \mathtt{N}_+, P_L^n(l_i) = l_j \end{aligned} \tag{5}$$

$$Map(l_i) = \begin{cases} h_j, & \eta(l_i, h_j) > \alpha \\ l_i \text{ is not mapped} & \eta(l_i, h_j) \leqslant \alpha \end{cases} \tag{6}$$

In (4), $\eta(l_i, h_j)$ represents the inner-similarity of the low-level element l_i and the high-level element h_j, which is the ratio of can-be-mapped children (represented as the number of elements in $Children_{mapped}(h_j, l_i)$) and all children of h_j (represented as the number of elements in $Children(h_j)$) . We count the children of h_j using $C_H(h_i, h_j)$ as defined in (5), and pick those which can be mapped to a child of l_i using $Map(E_L)$ as defined in (3). Equation (6) shows how to make a mapping decision between l_i and h_j based on $\eta(l_i, h_j)$. First, we should choose a threshold value $\alpha \in [0, 1]$. ω means the similarity between l_i and h_j, the closer to 1 means the more possibility that l_i should be mapped to h_j. According to previous studies [13], we set $\alpha = 0.5$. If $\eta(l_i, h_j)$ is greater than ω, then the mapping between l_i and h_j will be established. Otherwise, the mapping will not be established, then l_i will be labeled as not mapped and be put into the unmapped set represented as U. When all low-level elements in E_L are checked in the first step, we will turn into the second step if U is not empty.

At the second step of our algorithm, we improve the current CountAttract algorithm to map all low-level elements in the U to high-level elements. First, we calculate the Attraction value for each unmapped low-level element l_i and the high-level element h_j. This generates several candidate low-level elements which possibly should be mapped to h_j. Then, we automatically choose a low-level element among all candidates, and establish a mapping with h_j. The second step of our algorithm will not stop until all low-level elements are mapped to high-level elements.

Equation (7) and (8) show how to calculate the Attraction value of low-level element l_i and high-level element h_j (represented as $A(l_i, h_j)$).

$$\begin{aligned} A(l_i, h_j) =& \sum_{\forall l_j : C_L(l_j, l_i)} \sigma_1(l_j, h_j) \\ &+ \sum_{\forall l_j : \{l_i, l_j\} \in R_L \wedge Map(l_j) \in E_H} \sigma_2(l_j, h_j) \end{aligned} \tag{7}$$

$$\begin{aligned} \sigma_1(l_j, h_j) =& \begin{cases} 1, & Map(l_j) \in E_H \wedge C(Map(l_j), h_j) \\ 0, & \text{otherwise} \end{cases} \\ \sigma_2(l_j, h_j) =& \begin{cases} 1, & \{Map(l_j), h_j\} \in R_H \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{8}$$

The Attraction value of low-level element l_i and high-level element h_j is composed of two parts. The first part is the weighted sum of l_i's children (maybe the children of children, checked by C_L) which can be mapped to one of h_j's children (maybe the children of children, checked by C_H). The second part is the weighted sum of l_i's associated low-level elements which can be mapped to one of h_j's associated high-level elements.

We calculate the Attraction value for each unmapped low-level element l_i and the high-level element h_j, and use (9) to generate two candidate sets [13], $candSet_1(h_j)$ and $candSet_2(h_j)$. Then, we automatically choose one candidate low-level element in these two candidate sets, and map it to h_j.

$$\begin{aligned} candSet_1(h_j) &= \{l \mid A(l, h_j) \geqslant \overline{X(h_j)}\} \\ candSet_2(h_j) &= \{l \mid A(l, h_j) \geqslant \overline{X(h_j)} + sd(h_j)\} \\ \overline{X(h_j)} &= \frac{1}{|U|} \sum_{l \in U} A(l, h_j) \\ sd(h_j) &= \sqrt{\frac{1}{|U|} \sum_{l \in U} (A(l, h_j) - \overline{X(h_j)})^2} \end{aligned} \tag{9}$$

Each candidate in $candSet_1(h_j)$ has the Attraction value no less than the average of $UmappedSet$'s elements' Attraction values. Each candidate in $candSet_2(h_j)$ has the Attraction value no less than the average plus the standard deviation of $UmappedSet$'s elements' Attraction values. We believe that the larger of a candidate's Attraction value is, the more possible that a mapping should be established between this candidate and the high-level element h_j. To choose a candidate automatically and appropriately, we apply the following strategies:

(1) If there is only one candidate in $candSet_2(h_j)$, then establish a mapping between this candidate and h_j.

(2) If there are no candidates or more than one candidates in $candSet_2(h_j)$, then if it is the first time trying to map low-level elements with the high-level element h_j, stop mapping with h_j and try to establish mappings with other high-level elements which have not been tried with. When all other high-level elements have at least tried for one time, then try to establish mapping with h_j for the second time.
(3) If it is the second time trying to establish mapping with h_j, then recalculate $candSet_1(h_j)$ and $candSet_2(h_j)$. If there are no candidates in $candSet_2(h_j)$ this time, then choose the one with the largest Attraction value in $candSet_1(h_j)$ as the candidate. If there are one or more candidates in $candSet_2(h_j)$, then choose the one with the largest Attraction value in $candSet_2(h_j)$ as the candidate. Then, establish a mapping between the chosen candidate and h_j.

With the heuristic algorithm above, we can automatically establish the analysis graph between HG and LG as defined in (1).

C. Hierarchical consistency checking

After the analysis graph is established, we check the inconsistencies between architecture design and code according to the inconsistency check rules in Section III-A. Since the inconsistency check rules consist of two parts, our hierarchical consistency check also contains two parts: element consistency check and relationship consistency check.

First, we detect the element inconsistencies, including "element absence", "element divergence", "low-level element's name changing", "low-level element's hierarchy changing" and "low-level element's name and hierarchy changing".

Second, we detect the relationship inconsistencies based on the element consistency check result, including "relationship absence" and "relationship divergence". The relationship consistency check follows these steps:

1) All relationships in the architecture design that come from or target to high-level elements which have the "element absence" inconsistencies will be considered to have the "relationship absence" inconsistencies.
2) All relationships in the code that come from or target to low-level elements which have the "element divergence" inconsistencies will be considered to have the "relationship divergence" inconsistencies.
3) For each of the rest relationships in the architecture and in the code, firstly find the relationship's origin source element and origin target element, and record the mapping elements for each origin elements. If the mapped elements do not have the same relationship type or direction as the origin elements do, then this relationship will be considered as "relationship absence" if the origin elements are high-level elements, otherwise it will be considered as "relationship divergence" if the origin elements are low-level elements.

IV. Experimental Setups and Results

Given the goal of this study in Section I, we define the following research questions, and the evaluation experiment. We use our prototype to perform all experiments. In our prototype, the consistency checking module can check consistency between the architecture design and the codes evolved from the generated ones.

A. Research Questions

To evaluate our approach, we define two research questions.

RQ1: How is the capability of our approach?

To figure out the difference between our approach and others on detecting inconsistencies, approach automation and hierarchical-supportiveness, we should compare with various consistency checking approaches and tools. This includes major approaches such as JITTAC [19], and other related works.

RQ2: How is the performance of our approach compared with the state-of-arts?

To evaluate the precision and recall of our approach, and figure out to what extent has our approach improved the state-of-arts, we should conduct a series of experiments on different projects, and compare the results with other major approaches, in order to analyse the performance of our approach thoroughly.

B. Addressing the RQ1

1) Experiment Design: To evaluate the capability of our approach, we compare it with other existing consistency checking approaches from the respective of MDA-focused or not, automated or not, be able to check the hierarchical consistency or not, and supporting checking the detailed inconsistency or not (related with the elements and four kinds of relationships). We achieve this comparative analysis through a mini-sized survey by focusing on the above four aspects.

2) Experiment Results and Analysis: Table IV lists all referred approaches and the aspects we compare with. The column of "MDA" shows that most current approaches do not support model-driven software development, except for our approach and Biehl et al. [16]. However, Biehl's approach only considers architecture design drawn in class diagrams, thus only checks the inconsistency of classes and interfaces. Our approach is based on an extensible architecture design model, which contains class, interface, system and component. So, our approach can cover more specific relationships and inconsistencies. Finally, our approach implements the automated checking very well and the only human efforts need to be done are the architecture design before coding.

We also compare the tools that support consistency checking with our prototype, and the results are listed in Table V.

TABLE IV: Consistency Checking Approach Comparison

Approaches	Inconsistency Checking				MDA	Automated	Hierarchical
	Association Between Classes and Interfaces	Inheritance and Realization Between Classes and Interfaces	Dependency Between Systems and Components	Element Inconsistency			
Our Approach	✓	✓	✓	✓	✓	✓	✓
JITTAC [19]	×	×	✓	×	×	×	✓
Biehl's [16]	✓	×	×	×	✓	✓	×
Dependency Structure Matrix [20]	✓	×	✓	×	×	×	✓
SAVELife	×	×	✓	×	×	×	✓
ARTOS	×	×	✓	×	×	×	✓
Knodel's [17]	×	×	✓	×	×	×	✓
Haitzer's [18]	✓	×	✓	Partly ✓	×	×	✓

TABLE V: Consistency Check Tools Comparison

Tool	Architecture Description	Architecture Element Variety	License
Our Prototype	AMDL	✓	-
Structure101	Wireframe, Hierarchical description	×	Industrial
Sonar J	Hierarchical description	×	Industrial
ConQAT	Wireframe	×	Open-source
Lattix LDM [20]	Wireframe	×	Industrial
Macker	Wireframe	×	Open-source
XRadar	Wireframe	×	Open-source
Classycle [21]	Wireframe, Hierarchical description	×	Open-source
ReflexML [22]	Component Diagram	✓	Open-source

In Table V, we do the comparison from the aspects of architecture description, supporting the variety of architecture elements, and license. Most of the open-source tools use wireframe to describe the architecture design, so that their abilities to describe various elements such as interface or port are limited. Only our prototype and ReflexML [22] support various elements in the architecture description, while our's has rich semantics and detailed information of all types of elements using AMDL.

C. Addressing the RQ2

1) Experiment Design: To evaluate the performance of our approach, we choose Beihang Tongyan and ASM as two industrial cases according to the following reasons: First, the mapping is a tedious and error-prone work for human, so we can't use manual mapping results for comparison with the automated mapping results. Thus, we need to find a more feasible way. Second, the final goal of our method is to improve the consistency checking job. We argue that if the our mapping is correct, then the inconsistencies found by our method will be acknowledged by developers since they will map the same elements and find the same inconsistency. Otherwise, if our mapping is incorrect, then the detected inconsistency will not be acknowledged by developers. So, the performance of our approach can be evaluated through consistency check, which is more feasible for human efforts and more practical for software development. In this way, the results can be easily compared with other approaches.

Due to the above reasons, for each case, we use our prototype to establish the architecture design. And according to the nature of the case systems, we set different scenarios, including software evolution or inconsistency injection. Then we run our algorithm to check consistency for each scenario, and recruited volunteers or developers to verify the results. The validated results are evaluated using measurements like precision and recall.

In case that each approach may perform differently on a specific scenario, a detailed analysis between the rationales of our approach and others are given after each result, aiming at explaining the performance theoretically. The compared rationales include the mapping methodology, element inconsistency types and relationship inconsistency types.

a) Subject Projects: Beihang Tongyan project is developed by our colleagues, so the developers can be consulted for evaluation. Its architecture has been refactored from version 1.0 to version 2.0 which conforms to its architecture design better, thus we use our prototype to build version 2.0's architecture, and detect the inconsistency between version 1.0's code and the built architecture. Two developers are invited to decide whether they can agree with the inconsistencies found by our tool. The automated detecting results are also checked by an external reviewer from a different research institution in addition to the project's developers.

Since some types of inconsistencies may not be included in the Beihang Tongyan, we select an open-source project ASM [1] and inject some inconsistencies manually for evaluation. ASM was selected because there are sufficient documentations online to explain its architecture design so that we can build its architecture in our prototype. We recruited two volunteers (denoted as P1 and P2) to inject any inconsistency they want into ASM's code. Each

[1]https://asm.ow2.io/

volunteer injects 30 inconsistencies, as listed in Table VI. After the inconsistency injection, we let them check each other's injected inconsistencies to avoid bias.

Since our approach mainly improves the current RM-based consistency checking approaches, we perform three other popular approaches for comparison, including RM-based approaches and others: JITTAC [19], Biehl's approach [16] and the Dependency Structure Matrix approach [20]. JITTAC [19] and the Dependency Structure Matrix approach [20] need the predefined mappings between the elements in software design and source code, so we manually establish the mappings between components in design and packages in source code.

b) Measurements: We use our approach to check consistency in both cases automatically, and validate the results through verification of developers and volunteers. Based on the automatic consistency check results and manual verifications, we use precision and recall to measure the performance of the approaches.

In the consistency checking results, we record the number of those which are totally agreed by developers and volunteers as $cNum$. By saying "totally agreed", we mean that the manual check report the same result as our approach does. The number of identified inconsistencies which are basically agreed are recorded as $bNum$, which means that the result is truly an inconsistency, but it is not detailed enough. These results may be located into a deeper level of the architecture design or code if they are done manually. The number of inconsistencies which are disagreed by developers or volunteers are recorded as $dNum$. Finally, we record the number of inconsistencies that developers or volunteers believe to be but our approach can't find out as $nfNum$.

Using the marks defined above, we can define the precision and recall in (10).

$$\begin{aligned} Precision &= \frac{cNum}{cNum + bNum + dNum + nfNum} \\ Recall &= \frac{cNum + bNum}{cNum + bNum + dNum + nfNum} \end{aligned} \tag{10}$$

2) Experiment Results and Analysis for RQ2:

a) Experiment Results on Beihang Tongyan: The results of each inconsistency type are listed in Table VII. The results are verified by two developers. Since they both have the similar understanding of Beihang Tongyan's two versions' architectures and codes, their evaluation towards our check results are all the same. Thus, we give the overall analyzing results in Table VIII.

In Table VIII, we can see that our approach finds 301 inconsistencies, while 277 are totally agreed by developers. These results show that our approach can reach the precision of 90.82% and the recall of 95.74%, both higher than all other three approaches.

The 15 inconsistencies which are basically agreed by developers are due to the package merging. During the code evolution, 15 packages merge into one package. According the Attraction calculation between high-level and low-level elements, our analysis graph establishing method (see Section III-B) can map the new package in the code to all 15 high-level elements in the architecture design which are mapped to the original 15 packages in the old version of code. Thus, there are 15 "low-level element's hierarchy changing" inconsistencies as defined in III-A1. These 15 results are basically agreed by developers. However, they point out that it could be clearer if we define this type of inconsistency as "low-level element's merging".

There are nine "relationship divergence" inconsistencies which are disagreed by developers. This is because of the merging of packages cause some element's hierarchy to change, therefore some relationships' source elements or target elements differ from the original mapped elements. We do not have the "low-level element's merging" inconsistency checking, so this type of inconsistency is reported inaccurately.

There are four inconsistencies that we do not find out. These are four function changes which do not cause the change of the relationship between classes or interfaces. Since our approach cares more about the architecture consistency, we do not take code changes that are too much detailed into consideration.

We also briefly analyzed the results of other approaches in order to find out the reasons that their results differ from ours. JITTAC [19] finds 119 inconsistencies. JITTAC [19] maps the Dependency between systems and components in the architecture design to the Dependency between packages in the code. These mappings are very accurate in consistency checking, so all 119 inconsistencies found by JITTAC [19] are totally agreed by developers. JITTAC [19] does not check the Realization or Dependency between classes and interfaces, or the element change, so 182 inconsistencies are not found by JITTAC [19]. Biehl's approach [16] finds 109 inconsistencies. This approach only checks the Dependency between classes and interfaces, while the Inheritance and Realization between classes or interfaces, the Dependency between components and packages, or the element change are not checked. Thus, 196 inconsistencies are not found by Biehl's approach [16]. The Dependency Structure Matrix approach [20] finds 195 inconsistencies, where 186 are totally agreed by developers. The Dependency Structure Matrix approach [20] checks Association between classes and interfaces and Dependency between components and packages, which makes its precision and recall higher than JITTAC [19] and Biehl's approach [16]. However, 9 "relationship divergence" inconsistencies are disagreed by developers, which is the same case with our approach. The Dependency Structure Matrix approach [20] does not check the Inheritance and Realization between classes or interfaces, or the element change, so 110 inconsistencies are not found.

TABLE VI: Injected Inconsistencies

Volunteer	Element Inconsistency					Relationship Inconsistency								Total
	EI1	EI2	EI3	EI4	EI5	RI1	RI2	RI3	RI4	RI5	RI6	RI7	RI8	
P1's Injection	3	4	3	1	1	2	3	2	5	1	3	1	1	30
P2's Injection	5	3	3	7	2	0	3	0	1	1	1	1	3	30

Meanings of EI1 - EI5:

1) EI1: Element absence
2) EI2: Element divergence
3) EI3: Low-level element's name changing
4) EI4: Low-level element's hierarchy changing
5) EI5: Low-level element's name and hierarchy changing

Meanings of RI1 - RI8:

1) RI1: Association absence
2) RI2: Association divergence
3) RI3: Inheritance absence
4) RI4: Inheritance divergence
5) RI5: Realization absence
6) RI6: Realization divergence
7) RI7: Dependency absence
8) RI8: Dependency divergence

TABLE VII: Consistency Check Results of Beihang Tongyan

Approach	Element Inconsistency					Relationship Inconsistency							
	EI1	EI2	EI3	EI4	EI5	RI1	RI2	RI3	RI4	RI5	RI6	RI7	RI8
Our Approach	47	0	0	54	3	117	2	1	0	1	0	55	21
JITTAC [19]	0	0	0	0	0	0	0	0	0	0	0	117	2
Biehl's approach [16]	0	0	0	0	0	107	2	0	0	0	0	0	0
Dependency Structure Matrix [20]	0	0	0	0	0	117	2	0	0	0	0	55	21

Note: EI1 - EI5 and RI1 - RI8 are the same as Table VI

TABLE VIII: Analysis of Consistency Check Results of Beihang Tongyan

Approach	Inconsistency	Developers' Verification				Precision	Recall
		$cNum$	$bNum$	$dNum$	$nfNum$		
Our Approach	301	277	15	9	4	90.82%	95.74%
JITTAC [19]	119	119	0	0	182	39.01%	39.01%
Biehl's approach [16]	109	109	0	0	196	35.73%	35.73%
Dependency Structure Matrix [20]	195	186	0	9	110	60.98%	60.98%

b) Experiment Results on ASM: We use our approach to check the consistency between ASM's architecture design and two volunteer's injected code, and confirm the results with each volunteer respectively. The results for each volunteer are shown in Table IX and Table X.

For P1, we find 73 inconsistencies. 72 are totally agreed by P1. The one inconsistency which is not found is the change of methods and attributes within a class, which does not cause any change of relationships between classes or interfaces. This type of change is not detected in our method, same as the reason we explained in the consistency check results of Beihang Tongyan. JITTAC [19] finds 11 dependency divergences between packages, which are all totally agreed by P1. Both Biehl's approach [16] and the Dependency Structure Matrix approach [20] find the same association absences and association divergences in the total of 17, since they mainly check the relationship change between classes and interfaces.

For P2, we find 98 inconsistencies. 95 findings are totally agreed by P2. Two findings are basically agreed. One is the "low-level element's name changing" inconsistency which is caused by the moving of all elements from one package to another package. The other one is the "element divergence" inconsistency that is caused by the class addition in a package. One finding is not agreed by P2, which is a "dependency divergence" inconsistency caused by the element merging. The rejection reason is that the check result should give the "low-level element's merging" inconsistency. JITTAC [19] finds dependency absences and dependency divergences in the total of 46. Biehl's approach [16] finds association absences and association divergences in the total of 45. The Dependency Structure Matrix approach [20] finds association absences, association divergences, dependency absences and dependency divergences in the total of 53. These results are all totally agreed by P2. However, element inconsistencies, the inheritance inconsistencies and the realization inconsistencies are not accurately checked by any of these three approaches.

To sum up the comparison between our approach and other three approaches, we can find that:

- The Improved CountAttract Mapping algorithm did improve the mapping establishment in the RM technique, which results in better consistency check performance. Our approach reaches higher precision and recall than other three approaches due to the explicit inconsistency checking, the usage of initial mapping and automatic reflexion model method, and the consideration of both element change and relationship change during software evolution. However, we do not consider the merging or division of elements, which has caused some check results that are not agreed by developers and volunteers.
- JITTAC [19] performs well in the consistency checking of relationships between components and pack-

TABLE IX: Analysis of Consistency Check Results of ASM for P1

Approach	Inconsistency	Volunteer's Verification				Precision	Recall
		$cNum$	$bNum$	$dNum$	$nfNum$		
Our Approach	73	72	0	0	1	98.63%	98.63%
JITTAC [19]	11	11	0	0	62	15.06%	15.06%
Biehl's approach [16]	17	17	0	0	56	23.28%	23.28%
Dependency Structure Matrix [20]	17	17	0	0	56	23.28%	23.28%

TABLE X: Analysis of Consistency Check Results of ASM for P2

Approach	Inconsistency	Volunteer's Verification				Precision	Recall
		$cNum$	$bNum$	$dNum$	$nfNum$		
Our Approach	98	95	2	1	0	96.94%	98.98%
JITTAC [19]	46	46	0	0	52	46.94%	46.94%
Biehl's approach [16]	45	45	0	0	53	45.92%	45.92%
Dependency Structure Matrix [20]	53	53	0	0	45	54.08%	54.08%

ages, but it cannot check the relationships between classes and interfaces. Biehl's approach [16] supports the automatic consistency check of associations between classes and interfaces, but it lacks of the check of dependency changing. The Dependency Structure Matrix approach [20] reaches high precision and recall due to its ability to check the association between classes and interfaces, and the dependency between components and packages. However, all of the three approaches do not support the detailed consistency check of low-level elements, including the low-level element's name changing and hierarchy changing.

D. Threats to Validity

According to the taxonomy proposed by Wohlin et al. [29], there are three potential threats of validity that may apply to our work: threats to construct and internal validity, threats to external validity.

1) Threats to construct and internal validity: To reinforce the construct validity and internal validity of our study, we conducted experiments on two projects. One of the subject projects, Beihang Tongyan, contains the inconsistencies caused during the system evolution. Each automated inconsistency checking result on this project was checked independently by its two developers, to reduce the impact of subjectivity. To ensure our approach can detect as many types of inconsistencies as possible for different software, we let two volunteers independently inject any types of inconsistencies into the other open-source project, ASM, to do further evaluation. To alleviate the bias of subjective evolution on the automated checking results, we also ask the two volunteers to check each other's injected inconsistencies. Anyway, since the order in which other approaches are compared may affect the evaluation results, we plan to perform more case studies using random comparison order.

2) Threats to external validity: We evaluated our approach on two projects to cover different development scenarios that may cause inconsistencies. Especially, in the experiment on ASM, we allow volunteers to inject any types and numbers of inconsistencies, thus ensures our work's external validity. In principle, our work is applicable for any Java projects, as long as the architecture design is similar to UML like AMDL, and the initial mapping between source code and architectural elements are established. Anyway, to mitigate any potential threats to external validity futher, we plan to perform more experiments on larger projects which may cover more types of inconsistencies. We hope that this can help make our results more applicable in practice.

V. Conclusion

Inconsistencies between architecture and source code occur inevitably during the software maintenance. Although lots of good work has been done to identify the inconsistencies, challenges still exist. Most work requires huge manual annotations on the mappings between the software design and source code. Others which use some algorithms for automation, however, may miss some inconsistency types which are really helpful for the synchronization. In addition, the hierarchy between the elements in design and/or code increases the difficulties of mapping establishment. To solve these problems, we proposed a method to automatically identify the hierarchical inconsistencies between architecture design and Java implementation. We firstly define the finer-grained inconsistency types. Then we build analysis graph of software design and the source code by proposing the Improved CountAttract Mapping algorithm for hierarchical mapping. Finally, we detect the inconsistency based on rules.

We use an industrial project and an open-source project to evaluate the capability and performance of our method. The ratio of detected inconsistencies acknowledged by developers reflects our method's performance. For the industrial project, 95.74% detected inconsistencies are acknowledged by developers, and 90.82% are totally agreed by them. For the open-source project, more than 98% detected inconsistencies are acknowledged by two of our volunteers (98.63% and 98.98% respectively), and more than 96% are totally agreed by them (98.63% and 96.94% respectively). To support all of the work mentioned in this study, we improved our previous prototype. Any interested academic researchers can contact us for using this tool.

For future work, we will promote our approach to address the low-level element's merging inconsistencies, and perform additional experiments on larger projects (even non-MDA ones) in order to explore the approach's robustness after adding more types of inconsistencies. We will also do more comparisons between our approach and the combination of other approaches, in order to figure out what scenarios can benefit the most from our approach, and inspire better consistency check approaches.

Acknowledgment

Funding for this work has been provided by the National Science Foundation of China Grant No. 61672078 and 61732019.

References

[1] Murphy, Gail C., David Notkin, and Kevin Sullivan. "Software reflexion models: Bridging the gap between source and high-level models." Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering. 1995.

[2] Koschke, Rainer, and Daniel Simon. "Hierarchical Reflexion Models." WCRE. Vol. 3. 2003.

[3] Ali, Nour, et al. "Architecture consistency: State of the practice, challenges and requirements." Empirical Software Engineering 23.1 (2018): 224-258.

[4] Bittencourt, Roberto Almeida, et al. "Improving automated mapping in reflexion models using information retrieval techniques." 2010 17th Working Conference on Reverse Engineering. IEEE, 2010.

[5] Rosik, Jacek, et al. "Assessing architectural drift in commercial software development: a case study." Software: Practice and Experience 41.1 (2011): 63-86.

[6] Caracciolo, Andrea, Mircea Filip Lungu, and Oscar Nierstrasz. "A unified approach to architecture conformance checking." 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, 2015.

[7] Whittle, Jon, John Hutchinson, and Mark Rouncefield. "The state of practice in model-driven engineering." IEEE software 31.3 (2013): 79-85.

[8] Stevens, Perdita. "Bidirectional model transformations in QVT: semantic issues and open questions." Software & Systems Modeling 9.1 (2010): 7.

[9] Langhammer, Michael. Automated Coevolution of Source Code and Software Architecture Models. Vol. 23. KIT Scientific Publishing, 2019.

[10] Brambilla, Marco, Jordi Cabot, and Manuel Wimmer. "Model-driven software engineering in practice." Synthesis lectures on software engineering 3.1 (2017): 1-207.

[11] Christl, Andreas, Rainer Koschke, and Margaret-Anne Storey. "Automated clustering to support the reflexion method." Information and Software Technology 49.3 (2007): 255-274.

[12] Olsson, Tobias, Morgan Ericsson, and Anna Wingkvist. "Semi-automatic mapping of source code using naive Bayes." Proceedings of the 13th European Conference on Software Architecture-Volume 2. 2019.

[13] Christl, Andreas, Rainer Koschke, and Margaret-Anne Storey. "Automated clustering to support the reflexion method." Information and Software Technology 49.3 (2007): 255-274.

[14] Said, Wasim, Jochen Quante, and Rainer Koschke. "Reflexion models for state machine extraction and verification." 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018.

[15] Koschke, Rainer, et al. "Extending the reflexion method for consolidating software variants into product lines." Software Quality Journal 17.4 (2009): 331-366.

[16] Biehl, Matthias, and Welf Löwe. "Automated architecture consistency checking for model driven software development." International Conference on the Quality of Software Architectures. Springer, Berlin, Heidelberg, 2009.

[17] Knodel, Jens, et al. "Architecture compliance checking-experiences from successful technology transfer to industry." 2008 12th European conference on software maintenance and reengineering. IEEE, 2008.

[18] Haitzer, Thomas, and Uwe Zdun. "DSL-based support for semi-automated architectural component model abstraction throughout the software lifecycle." Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures. 2012.

[19] Buckley, Jim, et al. "JITTAC: a just-in-time tool for architectural consistency." 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.

[20] Sangal, Neeraj, et al. "Using dependency models to manage complex software architecture." Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2005.

[21] Elmer, Franz-Josef. "Classycle: Analysing Tools for Java Class and Package Dependencies." How Classycle works (2012).

[22] Adersberger, Josef, and Michael Philippsen. "ReflexML: UML-based architecture-to-code traceability and consistency checking." European Conference on Software Architecture. Springer, Berlin, Heidelberg, 2011.

[23] Schmidt, Douglas C. "Model-driven engineering." COMPUTER-IEEE COMPUTER SOCIETY- 39.2 (2006): 25.

[24] Brunet, Joao, et al. "Five years of software architecture checking: A case study of Eclipse." IEEE Software 32.5 (2014): 30-36.

[25] Passos, Leonardo, et al. "Static architecture-conformance checking: An illustrative overview." IEEE software 27.5 (2009): 82-89.

[26] Knodel, Jens, et al. Sustainable structures in software implementations by live compliance checking. Fraunhofer Verlag, 2011.

[27] Pruijt, Leo, et al. "The accuracy of dependency analysis in static architecture compliance checking." Software: practice and Experience 47.2 (2017): 273-309.

[28] Çilden, Evren, and Halit Oğuztüzün. "A Reflexion Model Based Architecture Conformance Analysis Toolkit for OSGi-Compliant Applications." 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017.

[29] Wohlin, Claes, et al. Experimentation in software engineering. Springer Science & Business Media, 2012.

[30] Perry, Dewayne E., and Alexander L. Wolf. "Foundations for the study of software architecture." ACM SIGSOFT Software engineering notes 17.4 (1992): 40-52.

[31] Tran, John B., et al. "Architectural repair of open source software." Proceedings IWPC 2000. 8th International Workshop on Program Comprehension. IEEE, 2000.

[32] Herrmannsdoerfer, Markus, Sander D. Vermolen, and Guido Wachsmuth. "An extensive catalog of operators for the coupled evolution of metamodels and models." International Conference on Software Language Engineering. Springer, Berlin, Heidelberg, 2010.

[33] Mentis, Anakreon, and Panagiotis Katsaros. "Model checking and code generation for transaction processing software." Concurrency and Computation: Practice and Experience 24.7 (2012): 711-722.

[34] Knodel, Jens, et al. "Static evaluation of software architectures." Conference on Software Maintenance and Reengineering (CSMR'06). IEEE, 2006.

[35] Ali, Nour, Jacek Rosik, and Jim Buckley. "Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study." Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures. 2012.

[36] Herold, Sebastian, and Jim Buckley. "Feature-oriented reflexion modelling." Proceedings of the 2015 European Conference on Software Architecture Workshops. 2015.

[37] Buckley, Jim, et al. "Real-time reflexion modelling in architecture reconciliation: A multi case study." Information and Software Technology 61 (2015): 107-123.