Computer Standards & Interfaces 87 (2024) 103774

Contents lists available at ScienceDirect

Computer Standards & Interfaces

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

Detecting deviations in the code using architecture view-based drift analysis

Burak Uzun, Bedir Tekinerdogan *

Information Technology Group, Wageningen University and Research, Wageningen, the Netherlands

ARTICLE INFO

Keywords:
Software architecture reconstruction
Model-driven development
Architecture drift analysis
Architecture drift

ABSTRACT

Context: One of the key requirements for the code is conformance with the architecture. Architectural drift implies the diverging of the implemented code from the architecture design of the system. Manually checking the consistency between the implemented code and architecture can be intractable and cumbersome for large-scale systems.
Objective: This article proposes a holistic, automated architecture drift analysis approach that explicitly focuses on the adoption of architecture views. The approach builds on, complements, and enhances existing architecture conformance analysis methods that do not adopt a holistic approach or fail to address the architecture viewpoints.
Method: A model-driven development approach is adopted in which architecture views are represented as specifications of domain-specific languages. The code in its turn, is analyzed, and the architectural view specifications are reconstructed, which are then automatically checked with the corresponding architecture models.
Results: To illustrate the approach, we have applied a systematic case study research for an architecture drift analysis of the business-to-customer (B2C) system within a large-scale software company.
Conclusion: The case study research showed that divergences and absences of architectural elements could be detected in a cost-effective manner with the proposed approach.

1. Introduction

Software architecture design represents the gross level structure of the system and defines the systemic design decisions. The architecture design, together with the rationale of the design decision, is described in the architecture documentation that can be used as a guideline for the corresponding implementation. A well-documented architecture is crucial for supporting communication among stakeholders, for guiding and analysis of the design decisions, and for guiding the organizational processes [1–3].

Unfortunately, software systems are rarely static and must be adapted due to bug fixes or new requirements. If the code and/or the architecture are adapted separately, this leads to the so-called *architectural drift* problem [4–8], which refers to the discrepancy between the architecture description and the resulting implementation. Architectural drift can occur even during the initial implementation of the architecture due to a lack of knowledge about the architecture or stringent time-to-market constraints. This drift may directly lead to increased maintenance time and cost because the important systemic design decisions are not followed and lost. Eventually, this can result in a system where the difference between the code and the system's architecture is so large that a complete system re-implementation is required.

Manually checking the consistency between the implemented code and architecture can be cumbersome and intractable for large-scale systems [9]. Hence, automated architecture drift analysis has been proposed to automatically check the discrepancy between architecture and code. To support architecture drift analysis, a model of the code is usually reconstructed, which is then compared to a model of the architecture, after which the discrepancies are highlighted. Three different discrepancies can be distinguished: (1) missing architectural elements in the implemented code (absence), (2) extra defined architectural elements in the implemented code (divergence), and (3) the implemented code having the same architectural elements as the architecture (convergence).

Software architecture is typically modeled using so-called architecture views that represent the system from one or more stakeholders' perspectives. By separating the architecture views, the ubiquitous notion of the separation of concerns principle is applied, thereby

* Corresponding author.
E-mail address: bedir.tekinerdogan@wur.nl (B. Tekinerdogan).

https://doi.org/10.1016/j.csi.2023.103774
Received 5 March 2023; Received in revised form 5 May 2023; Accepted 15 July 2023
Available online 16 July 2023

supporting the understandability, maintainability, and complexity management of the overall system. Hence, to provide a proper architecture drift analysis, it is important to check whether the guidelines of all relevant architecture views have been addressed in the code.

Our approach builds on, complements, and enhances existing architecture drift analysis methods that do not adopt a holistic approach or fail to address architecture viewpoints explicitly. We adopt a model-driven development approach in which architecture views are represented as specifications of domain-specific languages. In this context, the given code is analyzed, and the architectural view specifications are reconstructed from the code. Using architecture reconstruction, the code is analyzed and the necessary views are provided. The architecture drift analysis then checks the deviations in the code with respect to the architecture views. For the architecture framework, we have used selected viewpoints from the views and beyond approach [2].

To illustrate the approach, we have applied a systematic case study research for an architecture drift analysis of the business-to-customer (B2C) system within a large-scale software company. The case study research showed that divergences and absences of architectural elements could be detected in a cost-effective manner with the proposed approach.

The remainder of the article is organized as follows. Section 2 provides the background on software architecture modeling, software architecture reconstruction, and architecture drift analysis. Section 3 presents the proposed architecture drift analysis method. Section 4 presents the derived metamodeling of architecture viewpoints. Section 5 describes the architecture viewpoint-oriented software architecture reconstruction method in detail. Section 6 presents the implementation and the corresponding tool of the presented drift analysis method. Section 7 presents the case study research to illustrate and validate the approach. Section 8 presents the discussion, Section 9 the related work, and finally, Section 10 concludes the article.

2. Background

Before we describe the overall method, we first describe the key elements that are integrated in the overall process, including software architecture modeling (Section 2.1), architecture reconstruction (Section 2.2), and architecture drift analysis (Section 2.3).

2.1. Software architecture modeling

A common practice for describing the architecture according to the stakeholders' concerns is to model different architectural views [2,3,10]. Fig. 1 shows the conceptual model related to architecture views. An architectural view represents a set of system elements and relations associated with them to support a particular concern. Usually, multiple architectural views are needed to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. Architectural views conform to viewpoints that represent the conventions for constructing and using a view. Having multiple views helps separate the concerns and support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. A comprehensive approach for modeling software architecture based on viewpoints is the Views and Beyond (V&B) approach. The V&B approach defines the following view categories: *Module view* category that is used for documenting a system's principal units of implementation. *Component and Connector* category that is used for documenting the system's units of execution. *Deployment View* category is used to document the relationships between a system's software and its development and execution environments. Viewpoints are defined as styles which are used to define views. Although the V&B approach has defined a predefined set of architectural styles, it is also possible to define new styles for particular concerns.

2.2. Software architecture reconstruction

Software architecture reconstruction (SAR) is a reverse engineering process in which the architectural structure of a software system is extracted from system entities such as code, log, and documentation. Fig. 2 shows a conceptual model for architecture reconstruction. SAR is often needed to derive missing or incomplete architecture documentation or to identify and manage architecture drift. SAR methods can be applied to derive a single abstract model or extract architecture views of the system [11,12]. Hereby, since manual handling of the architecture reconstruction process is usually time-consuming and costly, automatized methods and tools are proposed. The SAR process results in architecture documentation that includes a description of a set of architecture views for addressing stakeholder concerns.

2.3. Software architecture drift analysis

Within the architecture-driven development context, the code must be consistent with the architecture (and vice versa). In Table 1 we list the definitions for the terms that we use related to architecture drift analysis. In case the architecture elements and the corresponding design decisions are not correctly reflected in the code, then we can identify this as an architecture drift. The notion of drift also implies the dynamic behavior of the problem due to the bugs introduced in the code or the need to adapt the code for changing requirements [13]. When comparing the architecture elements with the code then we can identify three different scenarios. If the relations that are present in the architecture are also found in the implementation, then this is *convergent* relation. In case the architecture relation is not present in the implementation, then this is called an *absence* relation. Absence relations occur of course, during the initial development of the system in which the architecture is defined but the implementation is not ready yet. As such, in the early phases of the development these absence relations might be a lesser concern. Finally, if the implementation includes relation that is not present in the architecture, then this is called *divergence* relation. Architectural violations are due to absence or divergence relations.

An often-used architecture consistency approach is the reflexion modeling approach as proposed by Murphy et al. [13]. In principle, a reflexion model allows a software developer to view the structure of a

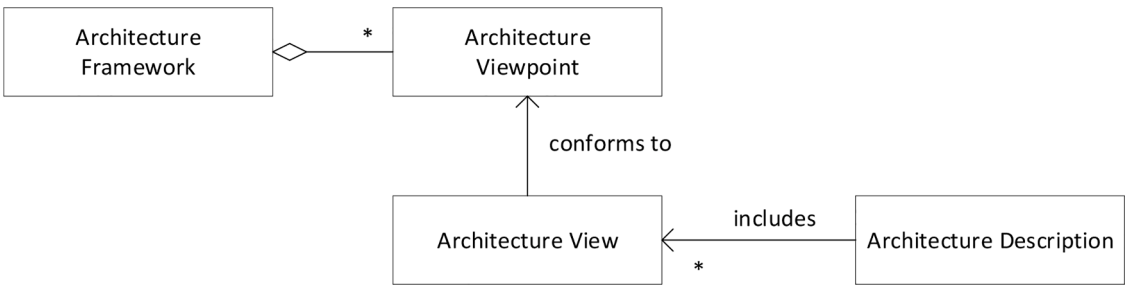

Fig. 1. Architecture Viewpoint Concepts and their relations (adopted from: [27]).

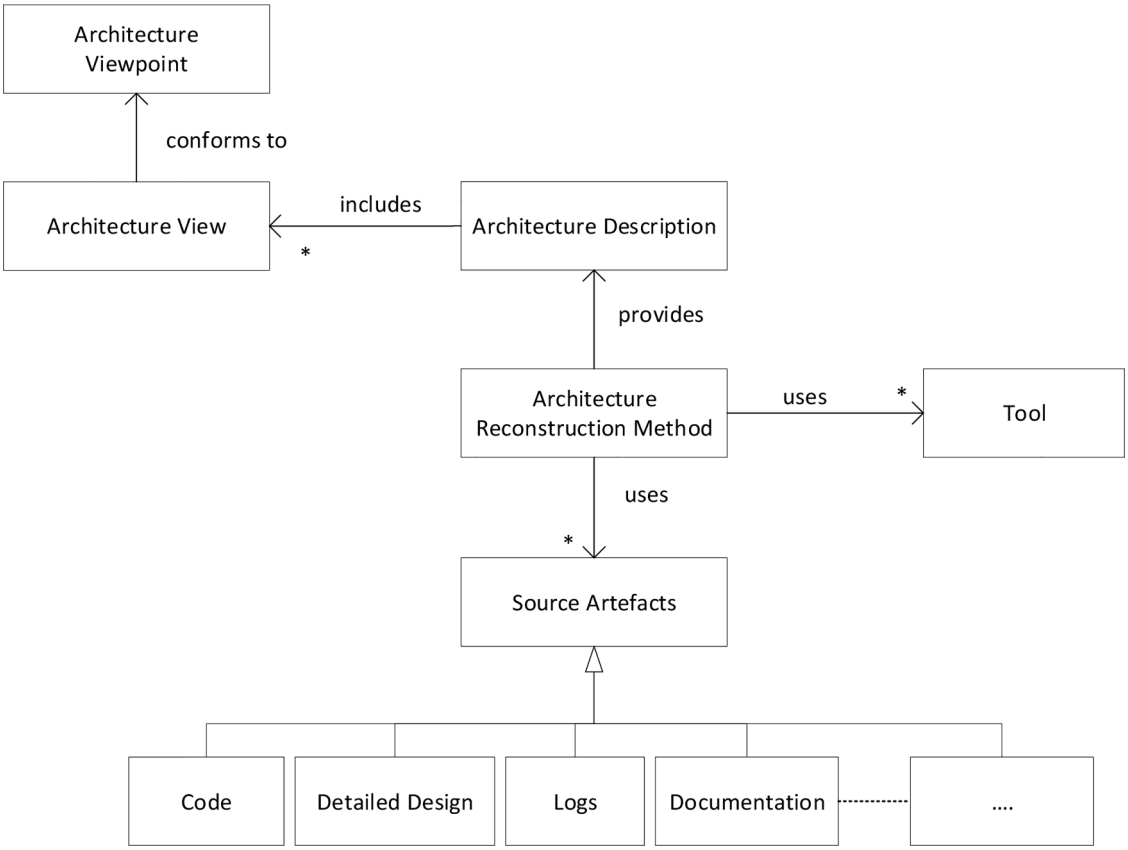

Fig. 2. Conceptual Model for Architecture Reconstruction.

Table 1
Adopted definitions related to architecture drift analysis.

Architectural Drift	is a phenomenon that occurs when the implemented code of a software system deviates from its intended architecture. This deviation can take the form of divergence or absence, as long as it does not violate the architectural constraints. Architectural drift can lead to inconsistencies and increased complexity in the system, negatively impacting its maintainability and evolvability.
Divergence	Divergence refers to the situation when the implemented code contains extra architectural elements that were not part of the system's intended architecture. These elements may not violate the architecture but may still introduce inconsistencies or complexities that were not anticipated in the design.
Absence	Absence occurs when the implemented code is missing architectural elements that were part of the system's intended architecture. This can lead to incomplete or incorrect implementations of the system's intended functionality and negatively impact the overall quality of the software.
Discrepancy	Discrepancies refer to the differences between the implemented code and the intended architecture of a software system. They can manifest as divergence or absence, and are indicative of architectural drift.
Convergence	Convergence represents the ideal state where the implemented code and the intended architecture are in alignment, with no discrepancies or deviations between them. Achieving convergence implies that the software system has been developed according to its architectural design and adheres to its constraints.

system's source through a chosen high-level (often architectural) view. To check the consistency between the architecture model and the code, an abstract model of the code is derived. The two models are then compared to each other with respect to earlier defined mapping rules between the code and the implementation. The results of the comparison are presented to the user through a *Reflexion Model*. Usually architecture drift analysis approaches that apply reflexion modeling include tools for modeling the architecture, modeling the mappings, deriving the abstract model from the source code, the consistency analysis checker, and the generator of the resulting reflexion model.

3. Viewpoint oriented software architecture drift analysis

In this section, we present our viewpoint-oriented software architecture drift analysis method. Fig. 3 presents the workflow model that defines the steps of the presented approach. Four different swimlanes are defined, each representing a distinct role: software architect, software developer, conformance tool, and reconstruction tool. The left part of the figure shows the role of the software architect who creates the architecture view models for the system. The right part of the figure illustrates the role of the software developer who implements the code based on the provided architecture view models. Both processes continue throughout the lifecycle of the software system, and thus, over time, architecture drift can occur. In many projects, the architecture drift is checked manually; however, this does not scale with the increasing size and complexity of software projects. Therefore, automated drift analysis is required, which is realized by a dedicated tool, as represented by the swimlane, Software Architecture Conformance Tool.

The tool expects two inputs: architecture models based on the various views, and the architecture models based on the implementation. Both inputs, the original architecture model specification and the extracted architecture models from the code, reflect architecture views. The software architecture reconstruction process extracts architectural elements from the code developed by software developers. The extracted architectural elements are then analyzed and formed into architecture components, which are subsequently derived into architecture view models. The models derived from architecture reconstruction and those provided by the software architect are compared against each other to

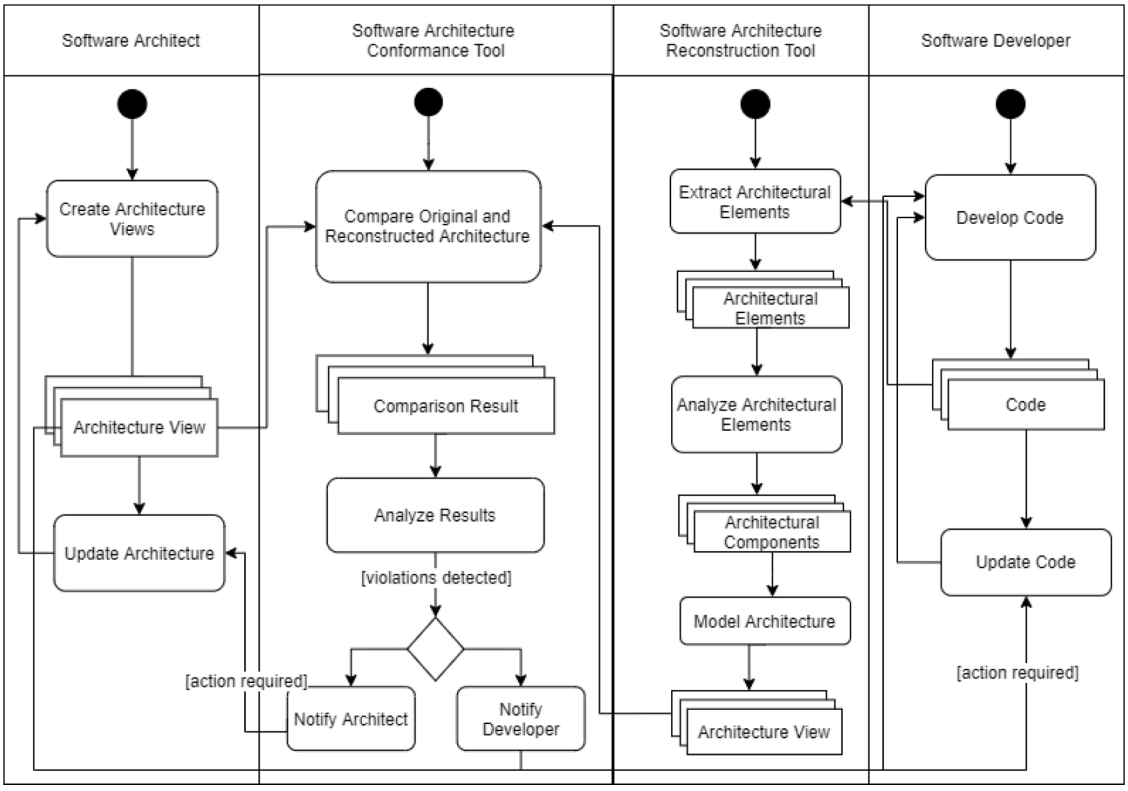

Fig. 3. Workflow Model for the adopted architecture drift analysis approach.

identify absences and divergences. It is important to note that the flows under conformance and reconstruction tools are automated.

We have implemented our tool for five architecture viewpoints, which include decomposition, shared data, uses, generalization, and layered. The subsequent sections of this study provide metamodels for the utilized architecture viewpoints and viewpoint-based approaches for architecture reconstruction and architecture drift analysis.

4. Metamodeling of architecture viewpoints

In order to model the architecture viewpoints, we have first defined the corresponding metamodels for the viewpoints and then mapped these to domain-specific languages using the Eclipse Ecore model [14–16]. The metamodels of the selected viewpoints are shown in Fig. 4. These metamodels are represented as Ecore models in the Eclipse IDE. Each of these metamodels has been developed after a thorough domain analysis of the corresponding viewpoints. For this, we have analyzed the viewpoints as discussed by Clements et al. [2], which provides a comprehensive approach for documenting software architectures using a broad set of viewpoints. In contrast to earlier viewpoint approaches, this approach provides a broader set of viewpoints and allows the introduction of new viewpoints. We have focused on the module viewpoints that define the architecture structure based on implementation units, that is, modules. The other two categories, the component & connector and the allocation viewpoints, do not focus on the implementation concerns and as such are less feasible for our purposes. As can be seen from the figure, we have used the following five views: decomposition view, shared data view, uses view, generalization view and layered view. Architecture models are defined as instances of these metamodels. Similarly, architectures extracted from the code (next section) is also defined as specifications of these models.

4.1. Decomposition view

Due to improvements in software development technology and the need for more complex requirements, the software industry is building larger and more complex software systems. One way to handle this complexity is well-adapted design paradigm called divide and conquer. The divide and conquer paradigm is basically breaking down complex problems into much smaller and manageable problems so that original problem can be solved with ease. One of the first actions taken by software architects while designing software systems is to derive decomposition view of the software system. Decomposition view visualizes the partition of code across different modules and submodules in a software system [2]. Decomposing a software system into smaller and cohesive parts is a great example of dividing and conquering paradigm. Fig. 4 presents metamodel for decomposition viewpoint in which model aggregates elements that are in type of module and subsystem. Element has a list property called subelements which references other elements for keeping submodules of a module.

4.2. Shared data view

Due to research in fields like big data and cloud engineering, database systems and their usages are getting more and more common. Nowadays, lots of software-intensive systems are integrated with some kind of persistent data stores which can be relational databases, filesystems, messaging queues and nonrelational databases. Since the presentation of interaction patterns between software systems and persistence data sources are an important aspect of software system design. Shared data view presents data sources along with their accessors in which accessors may have read or write interactions with these data sources. Shared data view is useful when data sources have multiple data accessors both reading and modifying the shared data [2]. Also, some additional data can be presented along with shared data view models such as restrictions for connections, access control authorizations, synchronization mechanisms and data properties. Shared data view model aggregates elements and attachments. Elements for shared data viewpoint can be repository and data accessor. These elements have two different attachments which are data write and data read relations.

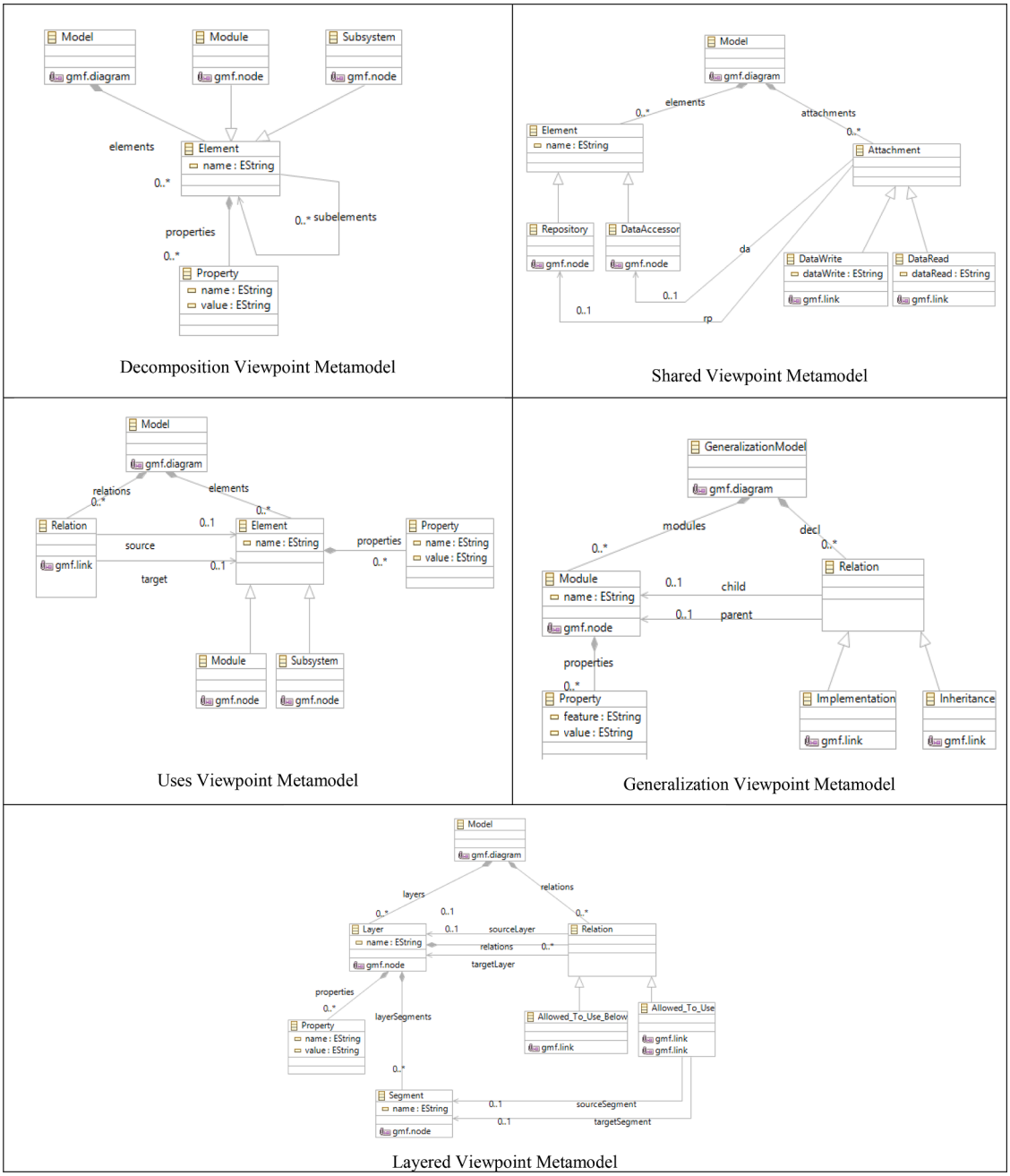

Fig. 4. Metamodels of the Selected Viewpoints.

4.3. Uses view

Generally, software systems are decomposed into little pieces due to the complexity in software systems which makes them more manageable and understandable for all involved stakeholders. Most of the time these decomposed pieces depend on each other for performing. Uses view presents usage dependencies between these pieces which are called modules. Usage dependencies occur whenever module's correctness depends on another module's correctness [2]. Uses view is insightful since the modules presented are the subset of the modules presented in decomposition view and it presents special type of relation between these modules. Also documenting uses view helps with incremental development and deployment [2]. Uses view model aggregates of relations and elements. Elements are modules or subsystems and relations hold two properties which are source and target elements.

4.4. Generalization view

By the nature of domain software system is modeling, it can be broken into pieces where some part of the system is more specialized version of another part. These specializations denote not just for difference but also commonality between parts. Generalization view presents special type of relation between modules called "is-a" relation [2]. This view is useful when extension and evolution of modules [2]. Modules in this case are classes or interfaces and generalization view presents inheritance or implementation relations between these elements. Inheritance relation exists between class and class or interface and interface elements. Moreover, implementation relation exists between class and interface elements. Generalization view model aggregates of modules and relations. Relations can be either implementation or inheritance and has two properties which are child and parent of that relation.

```java
public class ArchitectureReconstructor {
    private GeneralizationViewpointExtractor generalizationViewpointExtractor;
    private DecompositionViewpointExtractor decompositionViewpointExtractor;
    private UsesViewpointExtractor usesViewpointExtractor;
    private SharedDataViewpointExtractor sharedDataViewpointExtractor;
    private LayeredViewpointExtractor layeredViewpointExtractor;
    private HutnWriter hutnWriter;

    public ArchitectureReconstructor() {
        this.generalizationViewpointExtractor = new GeneralizationViewpointExtractor();
        this.decompositionViewpointExtractor = new DecompositionViewpointExtractor();
        this.usesViewpointExtractor = new UsesViewpointExtractor();
        this.sharedDataViewpointExtractor = new SharedDataViewpointExtractor();
        this.layeredViewpointExtractor = new LayeredViewpointExtractor(usesViewpointExtractor);
        hutnWriter = new HutnWriter();
    }

    public void reconstruct(String path){
        Generalization generalizationView = generalizationViewpointExtractor.extract(path);
        Decomposition decompositionView = decompositionViewpointExtractor.extract(path);
        Uses usesView = usesViewpointExtractor.extract(path);
        SharedData sharedDataView = sharedDataViewpointExtractor.extract(path);
        LayeredViewpoint layeredView = layeredViewpointExtractor.extract(path);
        hutnWriter.write(generalizationView, decompositionView, usesView, sharedDataView, layeredView);
    }
}
```

Fig. 5. Architecture Reconstruction Library Details.

4.5. Layered view

Software architecture evolved from copy and paste architecture to layered monolith and then to microservices. Both layered monolith and microservices divide software architecture units into layers of division where layers interact with each other for their correctness. The layered view is specialized version of the uses view where layers are cohesive group of modules that can interact in one-way direction [2]. Layered view model aggregates layers and relations. Relations can be either allowed to use below relation or allowed to use relation and hold two properties which are the source and target layer.

5. Viewpoint oriented software architecture reconstruction tool

We have developed a software architecture reconstruction tool that can extract architecture models from the code. The extracted architecture models represent the selected architecture views of the systems. We have created a library that can be plugged into projects that are programmed using Java as the programming language. The library depends on scripting the source code using Java reflection features. We have utilized an open-source project for Java runtime metadata analysis to fetch classes from the code under test [17]. Complete implementation for architecture reconstruction can be found on GitHub [14].

The developed architecture reconstruction method derives architecture view models from the code in four steps: pre-processing, extraction, analysis and model. We first pre-process the code by removing external dependencies and annotating code parts for repositories and its data accessors. Architecture elements are then extracted and mapped to architectural components. The resulting architecture is represented as Human-Usable Textual Notation (HUTN), which conforms to the Meta Object Facility (MOF) from the Object Management Group (OMG) for storing models in a human understandable format. Fig. 5 presents the part of the architecture reconstruction library. As can be seen from the figure, the ArchitectureReconstructor class has five fields that are responsible for extracting five different viewpoints. The results of the extractors are view models, which are then translated to HUTN models in reconstruct method.

Fig. 6 presents the simple usage of the library API. Architecture reconstructor object needs to be initialized and reconstruct method with the path of code under test should be executed.

Table 2 reports the pseudocodes for the architecture reconstruction code of the selected 5 viewpoints. We have defined an algorithm for each viewpoint and implemented them to reconstruct the corresponding architecture view model.

Fig. 7 shows, for example, the reconstruction of an architecture decomposition view from the code. Hereby, a system is illustrated that consists of three sub-modules X, Y and Z. The module X consists further of sub-modules X1 and X2, while the module Y consists of sub-modules Y1, Y2, and Y3. In a similar sense, the reconstructed architecture code views are easily extracted.

6. Viewpoint oriented architecture drift analysis tool

In the previous section, we have introduced the method to extract the software architecture models from the code. At this stage, we have both the reconstructed and original architecture ready for drift analysis. We used Epsilon Comparison Language (ECL) to compare different architecture models that conform to the same metamodel of the corresponding viewpoint. ECL is rule driven domain specific language for comparing different or same type of models [16]. Complete implementation can be found on github [18]. The following subsections elaborate on the implementation of the architecture drift analysis. Section 6.1 describes the implementation of the architecture drift analysis for each of the five viewpoints. Section 6.2 describes the results of the execution of the code. Finally, Section 6.3 presents the overall tool environment.

```java
ArchitectureReconstructor reconstructor = new ArchitectureReconstructor();
reconstructor.reconstruct( path: "case");
```

Fig. 6. Architecture Reconstruction API Usage.

Table 2
Pseudocodes for reconstructing view models per architecture viewpoint.

Viewpoint	Pseudocode

Decomposition	• Find all classes for given path
	• For each class
	  ○ Add package name to set
	  ○ Tokenize the package name by "." and add them to set
	• For each package name
	  ○ Find subelements that begins with package name
	  ○ Create a decomposition module object with name and subelements

Uses	• Find all classes for given path
	• For each class
	  ○ Create uses module
	  ○ Create uses relation for each declared field type
	  ○ Create uses relation for each method parameter types
	  ○ Create uses relation for each method return types

Generalization	• Find all classes for given path
	• For each class
	  ○ Create generalization module
	  ○ Create inheritance relation for each parent class
	  ○ Create implementation relation for each implemented interface

Layered	• Find all uses relations • Filter uses relations where no circular dependency exists
Shared Data	• Manually annotate data accessor classes with @Repo • Manually annotate data accessor methods with @Read and @Write • Create repositories for each repository used by data accessors • Create data accessors for each class annotated with @Repo • Create data read for each method annotated with @Read • Create data write for each method annotated with @Write

Table 3
Pseudocode for the architecture drift analysis.

Viewpoint	Comparison Pseudocode

Decomposition	• For each module
	  ○ Module names must match
	  ○ Modules subelements must match

Uses	• For each module
	  ○ Modules name must match
	• For each relation
	  ○ Relations source module and target module must match

Generalization	• For each module
	  ○ Modules name must match
	• For each implementation and inheritance relation
	  ○ Child and parent module of the relation must match

Layered	• For each module
	  ○ Modules name must match
	• For each allowed to use below relation
	  ○ Relations source module and target module must match

Shared Data	• For each repository
	  ○ Repository name must match
	• For each data accessor
	  ○ Data accessor name must match
	• For each data read
	  ○ Data read repository must match
	  ○ Data read accessors must match
	  ○ Data read qualifier must match
	• For each data write
	  ○ Data write repository must match
	  ○ Data write accessors must match
	  ○ Data write qualifier must match

6.1. Implementation of architecture drift analysis for each viewpoint

The drift analysis pseudocode for each viewpoint is primarily based on the elements and structure of the metamodels that we have presented in Fig. 4. In Table 3 we first report the corresponding derived pseudocodes for each architecture viewpoint. Each of the pseudocodes checks whether the required architecture elements from the corresponding viewpoints are whether present or absent in both the original architecture view and the reconstructed architecture view. The pseudocodes of Table 3 have all been implemented using ECL.

Fig. 8 shows the ECL code for comparing *decomposition views* as described in Table 3. A rule is defined to match the modules in the original and reconstructed decomposition view of an architecture. Matching is performed between original architecture(l) and reconstructed architecture(r). In alignment with the pseudocode and the metamodel, the rule compares the two models by checking whether the name of the modules as well as the subelements equality match.

Fig. 9 presents the ECL code for comparing reconstructed and original *uses views* of an architecture, which consists of two comparison rules for modules and relation elements. Reconstructed and original uses views of an architecture are compared by checking the equality in the module's name and relation's source and target modules.

```
decomposition {
    Model "decomposition"{
        elements :
            Module "S"{
                name :"s"
                subelements : Module "X", Module "Y",  Module "Z"
            },Module "X"{
                name : "s.x"
                subelements : Module "X1",  Module "X2"
            },Module "X1"{
                name : "s.x.x1"
            },Module "X2"{
                name : "s.x.x2"
            },Module "Y"{
                name : "s.y"
                subelements : Module "Y1",  Module "Y2",  Module "Y3"
            },Module "Y1"{
                name : "s.y.y1"
            },Module "Y2"{
                name : "s.y.y2"
            },Module "Y3"{
                name : "s.y.y3"
            },Module "Z"{
                name : "s.z"
            }
    }
}
```

Fig. 7. Example HUTN model for the decomposition view.

```
rule ModuleRule
    match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name and l.subelements.matches(r.subelements)
}
```

Fig. 8. ECL code for comparing decomposition views.

```
rule ModuleRule
    match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name
}
rule RelatationRule
    match l :  original!Relation with r : reconstructed!Relation {
    compare :  l.source.matches(r.source) and l.target.matches(r.target)
}
```

Fig. 9. Uses Viewpoint ECL Code.

Fig. 10 presents the ECL code for comparing reconstructed and original *generalization views* of an architecture. The comparison is performed on three elements of generalization viewpoint: module, implementation relation and inheritance relation. Module comparison checks for the equality between original and reconstructed views for the name property of modules. Both implementation and inheritance comparison check the equality of child and parent in corresponding relations.

Fig. 11 presents the ECL code for *layered views*. Comparisons are like the comparisons for the uses viewpoint. The only difference is the naming of the elements, whereas layer maps to the module and allowed to use below maps to relation rule. Layer comparison checks for the name equality and allowed to use below comparison checks matching the source and target layers of these restricted relations.

Fig. 12 presents the ECL code for *shared data views*. The set of rules as shown in the figure compares the reconstructed and original shared data views of an architecture. Two elements repository and data accessors are compared their name equality. Data read and write rules are compared by matching their data accessors, repositories and respective properties.

6.2. Results of the architecture drift analysis

The result of the execution of each ECL rule implementations in the previous sub-section is an array object that holds information about matches (convergence, absence and divergence) between the compared two view models. Matches contains information about what are being compared and if there are discrepancies or not. This object alone does not add much value, so we needed to extract matched elements (convergence), elements present in reconstructed view but not in original view (divergence) and elements present in original view but not present in reconstructed view (absence). Fig. 13 presents a code piece to extract this information from the mentioned data array. The first for loop statement extracts convergence relations between the original and reconstructed architecture. The second for loop statement extracts both divergence and absence relations between the original and reconstructed architecture.

Fig. 14 presents a sample output from decomposition ECL code execution. In this execution we used the sample model from Fig. 7 as an original decomposition view. We introduced a new module named K with its subelements K1 and K2 beneath module X and removed module Z completely from the view. In the figure, we can see that untouched module of X1, X2, Y, Y1, Y2 and Y3 are matched between the original and reconstructed view. However, modules X, Z and S did not match the modules in the reconstructed decomposition view. Moreover, modules S, X, K, K1 and K2 of reconstructed decomposition view did not match the ones in the original architecture.

6.3. Eclipse tool

Fig. 15 shows the eclipse workbench for software architecture drift analysis. The left part shows the package explorer in which we have partitioned our logic by each viewpoint. Each viewpoint consists of a metamodel file, executable ant build file, ECL file and two architecture view models in HUTN format. The section on the top right shows the editor support from the Eclipse Epsilon from framework. The lower right section presents the execution result of the ant build file, which prints out architecture deviations.

7. Case study design

To validate our architecture drift analysis approach, we have adopted the case study empirical evaluation protocol discussed by Runeson and Höst [19]. The protocol consists of the following steps: (1) case study design, (2) preparation for data collection, (3) execution with data collection on the studied case, (4) analysis of collected data (5)

```
rule ModuleRule
    match l : original!Module with r : reconstructed!Module {
    compare : l.name = r.name
}

rule ImplementationRule
    match l :  original!Implementation with r : reconstructed!Implementation {
    compare :  l.child.matches(r.child) and l.parent.matches(r.parent)
}

rule InheritanceRule
    match l :  original!Inheritance with r : reconstructed!Inheritance {
    compare :  l.child.matches(r.child) and l.parent.matches(r.parent)
}
```

Fig. 10. Generalization Viewpoint ECL Code.

```
rule LayerRule
    match l : original!Layer with r : reconstructed!Layer {
    compare : l.name = r.name
}

rule AllowedToUseBelowRule
    match l :  original!Allowed_To_Use_Below with r : reconstructed!Allowed_To_Use_Below {
    compare :  l.sourceLayer.matches(r.sourceLayer) and l.targetLayer.matches(r.targetLayer)
}
```

Fig. 11. Layered Viewpoint ECL Code.

```
rule RepositoryRule
    match l : original!Repository with r : reconstructed!Repository {
    compare : l.name = r.name
}
rule DataAccessorRule
    match l : original!DataAccessor with r : reconstructed!DataAccessor {
    compare : l.name = r.name
}
rule DataReadRule
    match l :  original!DataRead with r : reconstructed!DataRead {
    compare :  l.dataRead = r.dataRead and l.da.matches(r.da) and l.rp.matches(r.rp)
}

rule DataWriteRule
    match l :  original!DataWrite with r : reconstructed!DataWrite {
    compare :  l.dataWrite = r.dataWrite and l.da.matches(r.da) and l.rp.matches(r.rp)
}
```

Fig. 12. Shared Data Viewpoint ECL Code.

```
var matchedElements = new Set;
var originalUnmatchedElements = new Set;
var reconstructedUnmatchedElements = new Set;
for (matched in matchTrace.getMatches()) {
    if(matched.matching){
        matchedElements.add(matched.left.extract());
    }
}
for (matched in matchTrace.getMatches()){
    if(not matched.matching){
        var originalUnmatched = matched.left.extract();
        var reconstructedUnmatched = matched.right.extract();
        if(not matchedElements.contains(originalUnmatched)){
            originalUnmatchedElements.add(originalUnmatched);
        }
        if(not matchedElements.contains(reconstructedUnmatched)){
            reconstructedUnmatchedElements.add(reconstructedUnmatched);
        }
    }
}
```

Fig. 13. Conformance and Deviation Extracting ECL Code.

reporting. Table 4 reports the case study design steps for the selected case study.

The case study design approach is in the category of the applied research type. As such, the primary purpose is to understand the impact of adopting the architecture drift analysis approach within the real industrial context. One research question has been defined, which relates to how effective the adopted architecture drift analysis is. The effectiveness of the proposed approach is calculated by the number of divergences and absences discovered on the implementation of the system with respect to present divergences and absences. To this end, we have applied the fault injected real code approach to detect the inconsistencies. We have reconstructed relevant architecture views from the implementation and applied drift analysis using the described tool. We have applied exhaustive testing to trigger every flow path in Table 3 for each architecture view in an industrial case study. As shown in Table 4, our first-degree information sources are software developers, meetings and interviews. We also analyzed technical reports, technical documents along with the source code as second-degree information source. At last, we analyzed official documents provided by the company to other stakeholders as a third-degree information source. Table 4 also

```
[epsilon.ecl - DecompositionModelComparator.ecl] Matched Elements
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x.x2
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s.x.x1
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s.y
[epsilon.ecl - DecompositionModelComparator.ecl] 4. Module s.y.y1
[epsilon.ecl - DecompositionModelComparator.ecl] 5. Module s.y.y2
[epsilon.ecl - DecompositionModelComparator.ecl] 6. Module s.y.y3
[epsilon.ecl - DecompositionModelComparator.ecl] Unmatched elements from original architecture model
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s.z
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s
[epsilon.ecl - DecompositionModelComparator.ecl] Unmatched elements from reconstructed architecture model
[epsilon.ecl - DecompositionModelComparator.ecl] 1. Module s.x
[epsilon.ecl - DecompositionModelComparator.ecl] 2. Module s
[epsilon.ecl - DecompositionModelComparator.ecl] 3. Module s.x.k.k1
[epsilon.ecl - DecompositionModelComparator.ecl] 4. Module s.x.k
[epsilon.ecl - DecompositionModelComparator.ecl] 5. Module s.x.k.k2
BUILD SUCCESSFUL
Total time: 930 milliseconds
```

Fig. 14. Sample Output from Execution.

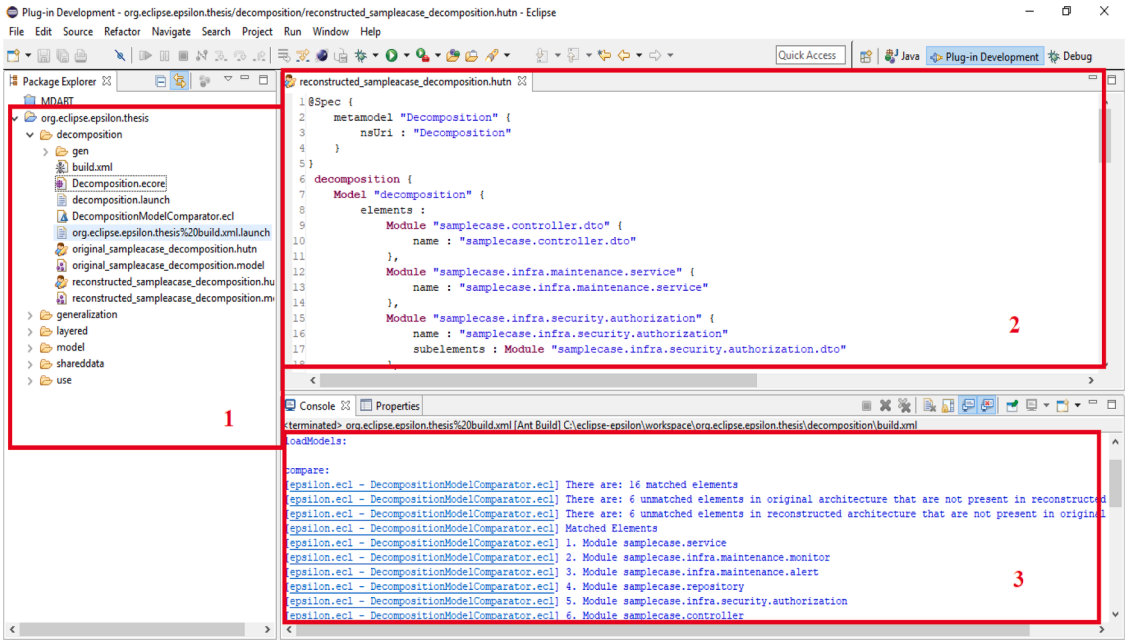

Fig. 15. Snapshot of the eclipse workbench showing the architecture drift analysis.

Table 4
Case study design.

Case Study Design Activity	Case Study
Goal	Assessing the effectiveness of the adapted architecture drift analysis approach
Research Questions	RQ1: How effective is the adopted architecture drift analysis approach and tool?
Background and source	Software Engineers (1st degree) Meetings and interviews (1st degree) Technical documents and reports (2nd degree) Source Code (2nd degree) Official documents (3rd degree)
Data Collection	Direct data collection through semi-structured interviews and meetings Independent data collection based on document analysis (the papers and technical reports) Indirect data collection based on source code analysis.
Data Analysis	Quantitative Data Analysis using Tables

reports the data collection methods from the information sources. We have conducted semi-structured interviews and meeting with the software developers. Then, we independently analyzed technical reports and official documents. Finally, the results of the previous steps have been reported.

7.1. Industrial case study: E-commerce

In this section, we outline the industrial case study utilized for validating our approach. Due to confidentiality constraints, the private company's name is withheld. Our approach has been applied within a real-world industrial context for e-commerce software. There are five primary e-commerce software categories: business-to-business (B2B),

business-to-customer (B2C), customer-to-business (C2B), customer-to-customer (C2C), and public administration. In this case, we focus on a B2C e-commerce system, which serves approximately eighteen million customers for monetary transactions, generating massive amounts of data related to customer transactions across various company ecosystems. We present the architecture view models for each viewpoint introduced earlier, using the original architecture models of the system for drift analysis, along with the deviated architecture models. The codebase, from which we derived the architecture models, consists of around forty thousand lines of code. Each team has a software architect who has a strong understanding of the problem domain and other systems within the company's ecosystem. The standard process for initiating a project within the company can be summarized in three steps: First, business analysts gather functional and non-functional requirements for the system. Second, the software architect assigned to the implementation team begins designing the high-level architecture. Lastly, software developers implement the requirements based on input from software architects and business analysts. Throughout this process, developers collaborate closely with the software architect to design the low-level architecture.

7.1.1. Shared data view

Fig. 16 presents the shared data view for the original architecture. It has one repository with add, delete, update and query methods on a single database called DB.

7.1.2. Decomposition view

Fig. 17 presents the decomposition view for the original architecture. There are nested packages inside each other and a total of twenty-two packages. As can be seen, form the figure *samplecase* package is divided into controller, service, util, repository and infra. Each of these packages is decomposed into smaller packages, forming the decomposition relation between the packages in a hierarchy, which can also be seen from the figure in detail.

7.1.3. Uses view

The uses view of the system is shown in Fig. 18. Here the arrows represent a uses relation that defines the dependency on the correct function of a module to the used module. As can be seen from the figure, there are 18 modules, and 24 uses relations between these modules. The most used module is infra.maintenance.log module, and the rest of the uses relations are distributed similarly between the modules.

7.1.4. Layered view

Fig. 19 shows the layered view of the original architecture. The controller is the highest layer in the architecture where it is allowed to use its sub-packages, service and the security packages from infrastructure. Moreover, the security layer has its own layering within where *infra.security.dto* package is the lowest layer in this view. We can also notice that *infra.maintenance.log* is used by three higher layers in

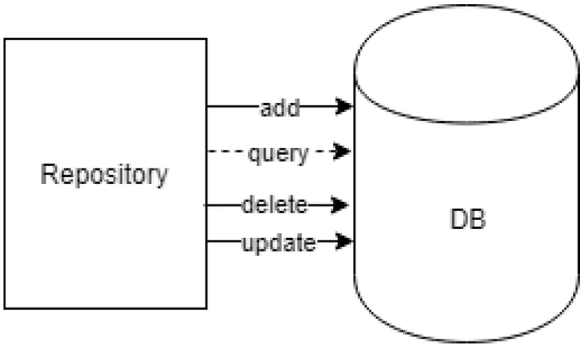

Fig. 16. Shared data view of adopted project infrastructure.

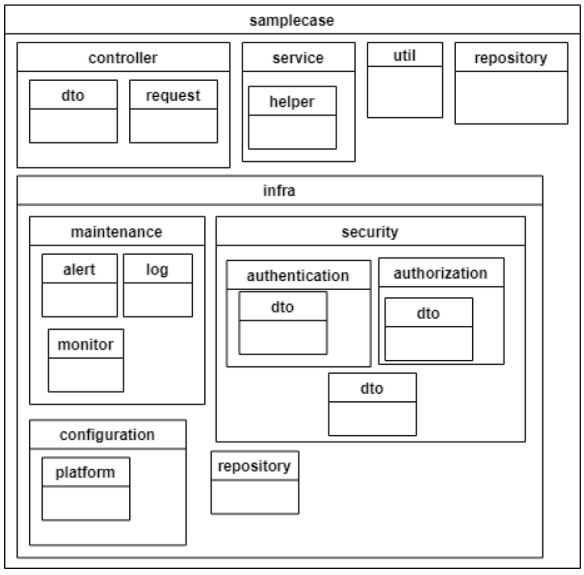

Fig. 17. Decomposition view of adopted project infrastructure.

different ranking which are: *controller, service* and *repository*. The *service* layer is allowed to use the *repository* and its sub package *helper*. This layer then allowed to use *util, infra.maintenance.alert* and *infra.security.authorization*.

7.1.5. Generalization view

In the system, we have also defined the generalization view to show the generalization specialization relations among the modules in the system. Fig. 20 presents a generalization view of the original architecture. In the figure, blue squares denote interfaces, black squares denote classes, dotted arrows denote the implementation relation and solid arrows denote extension relation. In the figure, three types of generalizations are shown including interface extensions, class extensions and interface implementations. Class can extend another class and class can implement an interface. However, interface can only extend another interface. As can be seen from the figure there are 21 classes, 6 interfaces, 9 inheritance relations and 9 implementation relations. Implementation relations are between classes and interface whereas extension relations are between the same type of entities (eg: class to class, interface to interface).

Based on the architecture documentation that consists of view descriptions, the system has been implemented. For the development and maintenance of the system, many developers and testers have been assigned who is responsible for the continuous maintenance and evolution of the system. For managing such a large system, it is important that the corresponding code is consistent with the architecture. Testing is carried out for different quality concerns. One important concern is also the alignment with the architectural design decisions.

7.2. How effective is the adopted VOSACAM approach?

In this section, we explain our exhaustive fault-based testing on our case to validate our adapted approach. Software testing is considered to be fault-based testing when the objective is to demonstrate an absence of predefined faults [20–22]. Therefore, we created a mutant copy of our case per architecture viewpoint according to criteria defined in Table 3. A mutant copy of a program is a program under test seeded with structural changes or bugs [23,24]. In our approach, we are executing ECL scripts on the reconstructed architecture (mutant copy) and original architecture of the case to detect and kill all mutants. Killing mutant is detecting the injected bug on our reconstructed architecture which implies that our approach is effective at finding real-life defects. In our

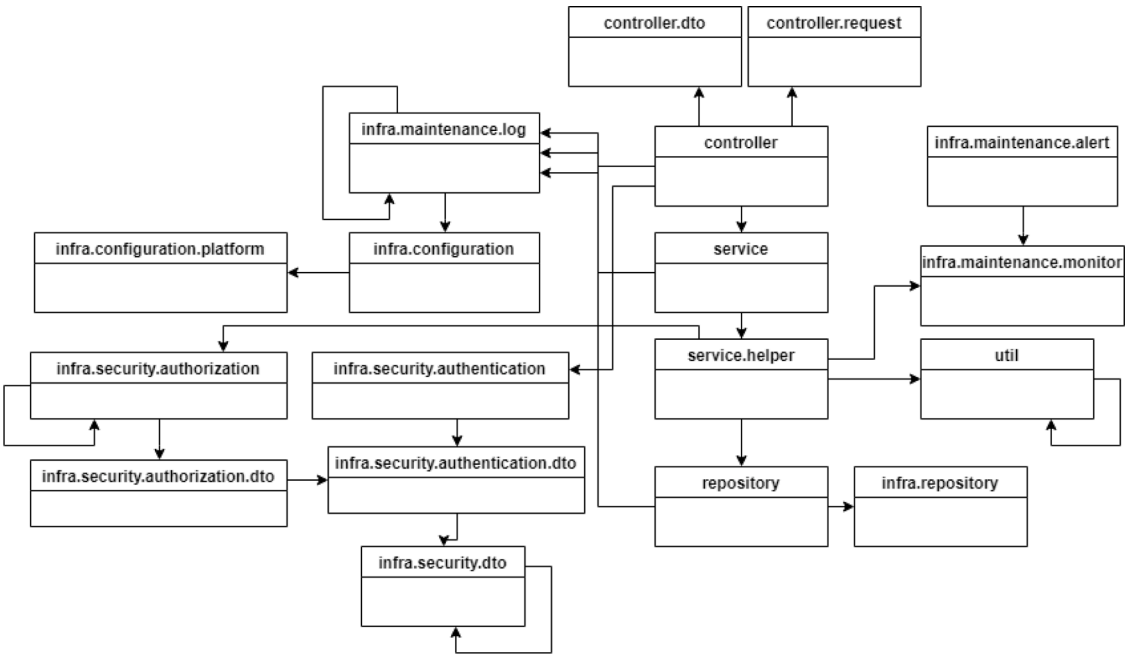

Fig. 18. Uses view of the adopted project infrastructure.

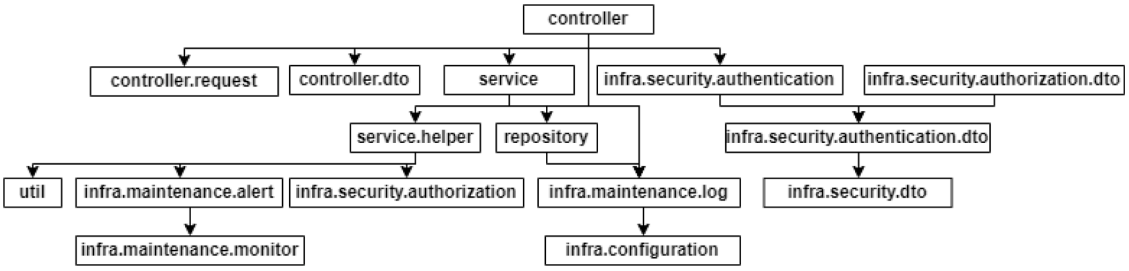

Fig. 19. Layered view of adopted project infrastructure.

experiments, we introduced absence and divergence of relations which corresponds to the removal and altering of a relation. Each viewpoint includes its own absence and divergence relations depending on the structure of the architecture viewpoint. Different strategies for mutation testing can be identified based on either first-order or high-order mutants. Jia and Harman introduced higher-order mutation testing in which mutation operators are applied more than once in their study [25]. For our testing purposes, this is inapplicable since our operators are conflicting with each other meaning that a relation cannot be removed and altered consecutively. Therefore, we applied first-order mutation testing.

Firstly, we generated the reconstructed architecture from the code using our novel software architecture reconstruction method presented in Section 5. These architecture view models are regarded as the original architecture of the system. Afterwards, we made changes to the system implementation in order to reconstruct the implemented architecture (mutant architecture) of the case study. Subsequently, we use our architecture drift analysis described in Section 6. The rest of this section provides mutants we created and the result of our approach per each viewpoint.

7.2.1. Shared data viewpoint

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for shared data viewpoint:

a) Add a new database called DB2
b) Add a new repository called repository2
c) Add both insert and search methods to repository2
d) Change add method type to data read
e) Change query method type to data write

Fig. 21 presents the execution result for shared data viewpoint drift analysis. We can see that the original architecture has two unmatched items, which are *query* and *add* methods, whose types were changed in the reconstructed architecture. Also, we can see six unmatched items from the reconstructed architecture. Four of them are newly added items and the remaining two of them were changed from the original architecture.

7.2.2. Decomposition viewpoint

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for the decomposition viewpoint:

a) Add a new service package under maintenance
b) Remove the dto package from the authentication package

We expect to detect unmatched dto and authentication packages from the original architecture. Furthermore, we also expect to detect newly added service packages and diverged maintenance packages at the reconstructed architecture. Fig. 22 presents the output of

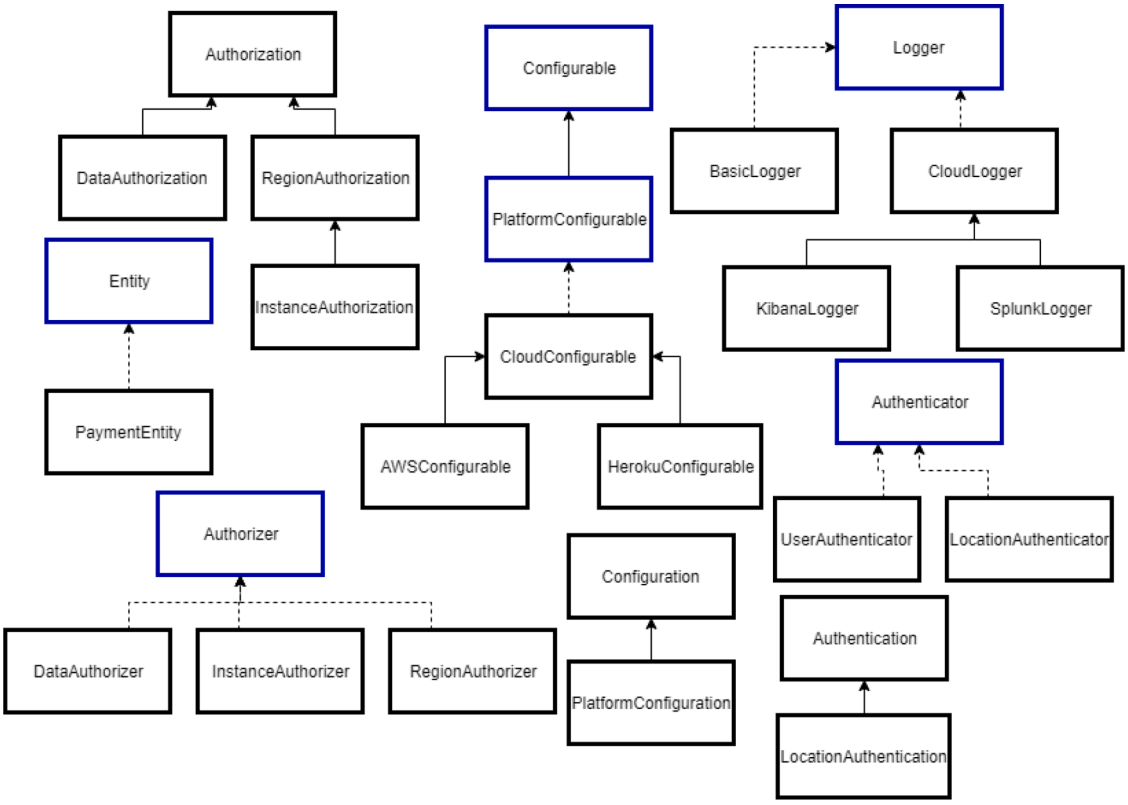

Fig. 20. Generalization view of adopted project.

```
Unmatched elements from original architecture model
1. query reads data using samplecase.infra.repository.Repository from the repository db
2. add writes data using samplecase.infra.repository.Repository from the repository db
Unmatched elements from reconstructed architecture model
1. Data Accessor samplecase.infra.repository.Repository2
2. insert writes data using samplecase.infra.repository.Repository2 from the repository db2
3. Repository db2
4. query writes data using samplecase.infra.repository.Repository from the repository db
5. add reads data using samplecase.infra.repository.Repository from the repository db
6. search reads data using samplecase.infra.repository.Repository2 from the repository db2
```

Fig. 21. Shared data architecture drift analysis result.

```
Unmatched elements from original architecture model
1. Module samplecase.infra.maintenance
2. Module samplecase
3. Module samplecase.infra.security.authentication
4. Module samplecase.infra.security.authentication.dto
5. Module samplecase.infra
6. Module samplecase.infra.security
Unmatched elements from reconstructed architecture model
1. Module samplecase.infra.maintenance
2. Module samplecase
3. Module samplecase.infra.security.authentication
4. Module samplecase.infra.maintenance.service
5. Module samplecase.infra
6. Module samplecase.infra.security
```

Fig. 22. Decomposition viewpoint drift analysis result.

architecture drift analysis between the original and the reconstructed architecture. We can see that removing dto package from under the authentication package caused nesting packages to diverge with respect to the original architecture. On the reconstructed architecture side, we can see a similar effect as dto package is not present, it causes nesting packages to diverge from the original architecture. Also, new service package is added under maintenance which triggered divergence both in original and reconstructed architectures. Maintenance in the original architecture was not matched with the reconstructed architecture. Whereas maintenance and service packages in the reconstructed architecture were not matched to the original architecture.

7.2.3. Generalization viewpoint

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for a generalization viewpoint:

a) Change type Authentication from class to interface
b) Add a new class EnvironmentConfiguration
c) Extend the EnvironmentConfiguration from Configuration class
d) Add a new interface ComplexLogger
e) Extend the ComplexLogger from Logger
f) Add a new class ConcreteLogger
g) Implement the Logger at ConcreteLogger
h) Remove the Entity interface

Fig. 23 shows the execution result of the drift analysis. Architecture Drift analysis shows that there are three divergences from the original architecture due to the removal of the Entity interface and changing type Authentication from class to interface. Also, there are seven new elements that are existing in the reconstructed architecture that does not exist in the original architecture. First, divergence relates to changing the type of Authentication in which the relationship between LocationAuthentication changes from inheritance to implementation. The second and third divergences are due to the introduction of ConcreteLogger and its implementation relation with Logger. Fourth and sixth divergences are due to introduction ComplexLogger and its inheritance relation with Logger. The fifth and seven divergences are due to the introduction of EnvironmentConfiguration and its inheritance relation with Configuration.

7.2.4. Uses viewpoint

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for uses viewpoint:

a) Add a new uses relation from infra.maintenance.monitor module to util module
b) Remove the relation between repository and infra.repository modules

Fig. 24 shows the execution result of architecture drift analysis for the uses viewpoint. As can be seen from the figure, architecture drift analysis detected one divergence (newly added relation) and one absence (removed relation) between architecture models.

7.2.5. Layered viewpoint

We have modified the implemented code as follows and run our architecture reconstruction method to extract the drifted software architecture model for layered viewpoint:

a) Remove the uses relation between infra.configuration and infra.maintenance.log modules
b) Add a new uses relation from repository module to controller module
c) Add a new uses relation from util module to service.helper module
d) Add a new uses relation from controller module to infra.security.authorization.dto

Fig. 25 presents the results of the architecture drift analysis execution result. We can see that due to the first three changes reconstructed architecture has seven absent allowed to use relation. Moreover, the last change caused divergence in the reconstructed architecture from the original architecture. We can see that the biggest impact was done by the second change in which we introduced a cyclic relation in layered relation. This change caused all layered relations between *controller, service* and *repository* to be violated.

Table 5 reports the execution result of our approach with respect to each viewpoint. The table consists of three sections which are divergence, absence and conformance relations. The first section gives an execution result for divergence relations for each architecture viewpoint. In a divergence relation, the reconstructed architecture has elements that are not present in the original architecture. The first column for this section shows the number of divergence relations after the fault injection process and the second column shows the number of divergence relations discovered by our tool. Moreover, the second section gives an execution result for absence relations for each architecture viewpoint. In an absence relation, the reconstructed architecture is missing architectural elements that are present in the original architecture. The first column for this section shows the number of absence relations after the fault injection process and the second column shows the number of faults discovered by our tool. Finally, the last section shows the conformance relations per architecture viewpoint. In a conformance relation, both the reconstructed and the original architectures share same architectural elements. The first column for this section shows the number of conformance relations after the fault injection process and the second column shows the number of conformance relations found. As can be inferred from the table, our approach has full coverage of both divergence and absence detection in architecture drift analysis.

8. Discussion

The contribution of this paper lies in the proposal and implementation of a viewpoint-oriented software architecture drift analysis method that specifically addresses the need for multiple architectural viewpoints when analyzing and maintaining complex software systems. The

```
Unmatched elements from original architecture model
1. samplecase.infra.security.authentication.dto.LocationAuthentication inherits samplecase.infra.security.authentication.dto.Authentication
2. Module samplecase.infra.repository.Entity
3. samplecase.repository.PaymentEntity implements samplecase.infra.repository.Entity
Unmatched elements from reconstructed architecture model
1. samplecase.infra.security.authentication.dto.LocationAuthentication implements samplecase.infra.security.authentication.dto.Authentication
2. Module samplecase.infra.maintenance.log.ConcreteLogger
3. samplecase.infra.maintenance.log.ConcreteLogger implements samplecase.infra.maintenance.log.Logger
4. Module samplecase.infra.maintenance.log.ComplexLogger
5. samplecase.infra.configuration.platform.EnvironmentConfiguration inherits samplecase.infra.configuration.Configuration
6. samplecase.infra.maintenance.log.ComplexLogger inherits samplecase.infra.maintenance.log.Logger
7. Module samplecase.infra.configuration.platform.EnvironmentConfiguration
```

Fig. 23. Generalization viewpoint architecture drift analysis result.

```
Unmatched elements from original architecture model
1. samplecase.repository uses samplecase.infra.repository
Unmatched elements from reconstructed architecture model
1. samplecase.infra.maintenance.monitor uses samplecase.util
```

Fig. 24. Uses viewpoint architecture drift analysis execution result.

```
There are: 28 matched elements
There are: 7 unmatched elements in original architecture that are not present in reconstructed architecture
There are: 1 unmatched elements in reconstructed architecture that are not present in original architecture
Unmatched elements from original architecture model
1. Layer samplecase.infra.configuration
2. Layer samplecase.util
3. samplecase.controller is allowed to use below samplecase.service
4. samplecase.service.helper is allowed to use below samplecase.util
5. samplecase.service.helper is allowed to use below samplecase.repository
6. samplecase.infra.maintenance.log is allowed to use below samplecase.infra.configuration
7. samplecase.service is allowed to use below samplecase.repository
Unmatched elements from reconstructed architecture model
1. samplecase.controller is allowed to use below samplecase.infra.security.authorization.dto
```

Fig. 25. Layered viewpoint architecture drift analysis execution result.

Table 5
Execution result of our approach.

Viewpoint	Divergences		Absences		Conformances	
	# of Divergences Present	# of Divergences Found	# of Absences Present	# of Absences Found	# of Conformances Present	# of Conformances Found
Generalization Viewpoint	7	7	3	3	70	70
Decomposition Viewpoint	6	6	6	6	16	16
Layered Viewpoint	1	1	7	7	28	28
Uses Viewpoint	1	1	1	1	59	59
Shared Data Viewpoint	6	6	2	2	4	4

motivation for incorporating multiple viewpoints stems from the fact that they provide a better representation of the system from various stakeholders' perspectives, thereby facilitating the understanding, maintainability, and complexity management of the overall system. While an integral architectural model provides a unified view of the software architecture, it may not sufficiently capture the unique concerns and perspectives of different stakeholders. The use of multiple viewpoints allows for a more comprehensive analysis of the architecture by explicitly addressing the specific concerns of each stakeholder. This enables a more targeted and efficient architecture drift analysis, as it allows the identification and resolution of discrepancies in the areas that matter most to each stakeholder. Other proposals may indeed use different architectural models as needed; however, our approach emphasizes the importance of adopting a viewpoint-oriented method that explicitly caters to the diverse concerns of stakeholders, which may not be adequately addressed in other approaches. The introduction of software architecture reconstruction and analysis tools is crucial for our viewpoint-oriented approach. While there are existing tools for reverse engineering code into architectural models or drift analysis, they may not be designed to support multiple architectural viewpoints or tailored to the specific concerns of various stakeholders. Our approach and its corresponding tooling are specifically developed to handle the unique requirements of a viewpoint-oriented architecture drift analysis.

Our architecture drift analysis approach has been evaluated with respect to effectiveness. We have conducted an industrial case study to assess the effectiveness of our approach using a case study protocol defined by Runeson and Höst [26]. While the system used in the case study is an actual industrial system, the use of mutants was chosen as a means to simulate possible deviations and evaluate the effectiveness of the architecture drift analysis approach in a controlled setting. This allowed us to systematically evaluate the performance of the proposed approach in detecting these inconsistencies. Although the analysis was not performed during the actual development of the system; however, the case study still provides valuable insights into the effectiveness of the architecture drift analysis approach when applied to a real-world system. The industrial context is reflected in the system's complexity and scale, as well as the involvement of software developers, meetings, interviews, and analysis of technical reports and documents related to the system. Based on this evaluation, we can state that the approach was effective in detecting deviations between the original and final architecture of the software system which are absence and divergence relations. It executes successfully on an industrial software system and presents plausible results. We used a syntactic approach as many conformance analysis approaches did. The approach could be further extended with semantic-based approaches (e.g. using ontologies). We consider this as a our future work.

Similar to any case study research, our study also has some validity threats. Internal validity refers to the casual relation between treatment and outcome. While assessing the effectiveness of the suggested approach, we have applied formal fault-based testing and reported the results in an isolated environment that no external variable can affect. One can argue that injected violations are not real-world system faults but faults our approach aimed to detect do not change in real-world systems. Nevertheless, our experiments are executed on real industrial case. External validity refers to concern of generalizing the results of a scientific study. In our case study, we have applied our suggested approach on B2C e-commerce system. However, architecture deviation problem is not related to specific domain but characteristic problem which might occur when software is developed. Also, our approach is executed under one case study but what really matters in our context is

not the number of case studies but the size and complexity of the systems. In our approach, we have adopted five different viewpoints, defined the metamodels and domain-specific languages for these, implemented the viewpoint-oriented software architecture reconstruction method and implemented viewpoint-oriented software architecture drift analysis. The approach is generic enough and could be generalized for different viewpoints and different software system in any domain. As stated before, the system that is being developed can be a complex system, and hence, we could state that we have adopted a sufficiently representative system to support the external validity. Obviously, in practice, even larger systems could be identified, but the properties of architecture code consistency will be similar. In our future research, we will focus on further applications of the approach.

The topic of this paper, focusing on detecting and analyzing architectural drift, has implications for the enforcement and compliance of architectural standards and guidelines in the context of software development. Architectural standards are crucial for maintaining software quality, ensuring consistency across different systems, and facilitating communication among stakeholders. By providing a rigorous method to automatically identify discrepancies between the designed and implemented architectures, our approach contributes to the adherence and enforcement of established architectural standards in the software development process. Furthermore, our adoption of widely-accepted architecture view modeling notations and methodologies reinforces our commitment to integrating our approach with current industry practices and de facto standards. Thus, we believe that our research findings are not only interesting but also directly relevant to the application of standards in the development of computer systems, making our work well-suited for the journal's scope and audience

9. Related work

This study builds on our earlier study [27] which focuses on architecture conformance analysis using a model-based testing approach for checking the consistency between architectural models and the code. The main objective of this earlier work was to automatically derive test cases from architectural views to check the architectural constraints in the code, assuming the architecture is correct and the code needs to align with it. In contrast, this study presents a viewpoint-oriented software architecture drift analysis method that specifically targets divergence relations, emphasizing the importance of architectural viewpoints in addressing the discrepancies between the architecture description and the code. This approach provides an integrated, stakeholder-focused analysis to ensure that architecture drift analysis is effective and relevant for all parties involved. In summary, while the earlier paper focuses on architecture conformance analysis using a model-based testing approach, the new paper presents a viewpoint-oriented software architecture drift analysis method that specifically targets divergence relations and emphasizes the importance of architectural viewpoints. This new approach offers a more comprehensive and stakeholder-focused analysis, with practical applicability and relevance in real-world scenarios.

Our earlier study [28] focuses on addressing the architectural drift problem by introducing a notation based on design structure matrices. The main contribution of the earlier work is the introduction of design structure reflexion matrices (DSRMs) as a complementary and succinct representation of the architecture and code, supporting qualitative and quantitative analysis and refactoring. The focus, the problem statement and the adopted approach is thus different.

Several studies have been proposed for architecture drift analysis or also called architecture conformance analysis. Architecture conformance analysis can both be driven in the code level or architecture level. Code-level architecture conformance analysis is where software architecture models are transformed into code-level tests which are then executed against code under test. Architecture level conformance analysis is where software architecture models are reconstructed and executed against architecture under test. In our previous work [11] we have conducted a domain-driven analysis of architecture reconstruction methods in order to create a feature model and generic business process model from 17 studies that we have selected. We have developed a novel reconstruction method based on the outcome of the synthesis of the identified primary studies. In [27], we have adopted a model-based testing approach for deriving test cases to be used for architecture conformance analysis. However, as stated before, in that study we focused on generating test cases that can be used to check the compliance with the architecture and thus the absence relations. In this study however we focus on the deviations in the code that are not represented in the architecture, thus divergence relations.

Architecture reconstruction has been addressed in several studies. In [29] also a software architecture reconstruction method based on architecture viewpoints is presented. The method uses natural language processing and string parsing to reconstruct UML architecture models. In [14] an approach is presented that uses source files names in clustering algorithms for reconstructing architecture models from the result of the applied algorithm. In [30] a method is presented similar to [29] which is based on architecture viewpoints using clustering algorithms and generating customer architecture models that are conforming to metamodel defined in the study.

Various studies propose different approaches for architecture conformance analysis. In [14], the authors propose a tool called SCHOLIA that extracts runtime object graph (ROG) to derive ownership object graph (OOG), which represents the static hierarchical relations within software systems from the code. This study utilizes annotated code and executes conformance analysis between the generated model and existing model automatically. The tool has been evaluated using a case study. In study [19], the authors propose a conformance analysis tool called jRMTool utilizing reflexion modeling. The corresponding tool is evaluated for an internally used application in an organization. Reflexion model (RM) is created from the code and pre-defined architecture model. RM model is then analyzed for detecting divergence and absence relations for conformance analysis. Study [11], focuses on the architecture constraint analysis to detect whether original architecture and resulting architecture conforms to each other or there are deviations in this relation. Authors propose tool called dclcheck that executes architectural constraints defined by software architect against the code to detect any violation. Detected violations means that there is an erosion in the software architecture which leads to nonconformance between original architecture and the code. Study [31] proposes a tool called ArchRuby to detect architectural violations in the code similar to study [11]. Architectural rules are defined and executed against source code to detect architectural erosions which leads to nonconformance between original architecture and the resulting code. RM model is created from the rule executions and analyzed for detecting divergence and absences relations. Study [21] presents a tool called LISA to check software systems against the reference architecture defined. At first tool reconstructs architecture models from the code in LISA model format and then apply conformance analysis. Study [32] proposes yet another tool called ConQAT in which graph-based model is reconstructed from code and then if same mappings are checked for existence between original architecture and reconstructed architecture model. Study [33] proposes a tool extends static analysis platform Magellan. Firstly, tool reconstructs software architecture from the code by static analysis of files. Then predefined constraints are executed on the reconstructed architecture to detect deviations in the architecture. Study [34] presents a tool called SAVE, which reconstructs architecture models from code in Data model format. Then reconstructed architecture and original system architecture are checked for absence and divergence relations. Study [25] proposes ExplorViz tool, which is a web-based architecture model visualization tool. Authors argue that manual conformance analysis can be performed using tool by checking architecture model and then analyzing code. Study [35] proposes a tool called SARTE, which takes input of software architecture specifications as form of finite state

Table 6
Overview of existing architecture conformance analysis and architecture reconstruction method.

Tool / Conformance Analysis Feature	Architecture Reconstruction Applied	Conformance Analysis Execution Level	Checked Relations	Architecture View	Architecture Model Type	Execution	Case Study Protocol Applied
VOSACAM	Yes	Architecture Level	Divergence Absence	Decomposition Shared Data Uses Generalization Layered	HUTN	Automated	Yes
MDABT	No	Code Level	Divergence	Decomposition Shared Data Uses Generalization Layered	HUTN	Automated	Yes
SCHOLIA	Yes	Architecture Level	Divergence Absence	Generalization Uses	OOG	Semi-automated	No
jRMTool	Yes	Architecture Level	Divergence Absence	Uses	RM	Semi-automated	No
dclcheck	No	Code Level	Divergence Absence	Uses Layered Generalization	NA	Automated	No
ArchRuby	No	Code Level	Divergence Absence	Uses	RM	Automated	No
LISA	Yes	Architecture Level	Divergence Absence	Decomposition Layered Uses Generalization	LISA	Automated	Yes
ConQAT	Yes	Architecture Level	Absence	Decomposition Uses Layered	Graph	Automated	No
Magellan Ext	Yes	Architecture Level	Divergence Absence	Decomposition Uses	LogEn	Automated	No
SAVE	Yes	Architecture Level	Divergence Absence	Decomposition Uses	Data	Semi-Automated	No
ExplorViz	No	NA	Divergence Absence	NA	NA	Manual	No
SARTE	No	Code Level	Absence	Uses Decomposition	FSP	Semi-Automated	No
CHARMY	No	Code Level	Absence	Uses Decomposition	Charmy Specs	Manual	No

process model (FSP), test criteria. Tool generates test cases that are mapping architectural constraints to source code level. Study [36] presents a tool called CHARMY which transforms original architecture of software systems into code level tests. Tests are executed against the source code manually to detect the violations against the original architecture.

Table 6 reports the summary of the related work for architecture conformance analysis. The first row represents the method that we have proposed in the article, while the others represent the related approaches. We have focused on seven different features to characterize these studies. The first column shows whether the indicated approach uses architecture reconstruction or not. It appears that not all studies depend on architecture reconstruction; these methods map architectural specifications to code level testing. Conformance analysis is both done at the architecture level and code level. The table also reports that most of the studies checked absence and divergence relations together and only four studies checked for only one relation. We can also derive from the table that there is also no unified architecture model type for carrying out architectural conformance analysis. The most used architecture model types are RM and HUTN models. None of the studies explicitly focus the notion of viewpoints, but we still derived and interpreted which architecture viewpoint conformance analysis that have been used in the conformance analysis. We can see that uses, decomposition and layered viewpoints are the mostly used viewpoints. Furthermore, we can see most of the studies are using automated tools but there are some tools that are not fully automated and use manual checking. The studies that were checked do not evaluate the approach using a formal case study research protocol. Given this result, we can conclude that our approach is both complementary to the existing approaches and novel in the sense that it provides an integrated viewpoint oriented approach that is validated within an industrial context.

To sum up, our approach offers several key differences and main contributions when compared to the related work presented:

- Focus on divergence relations: While our previous study [27] and other approaches primarily concentrate on the absence relations, our current approach specifically addresses the deviations in the code that are not represented in the architecture, thus targeting divergence relations.
- Viewpoint-oriented method: Unlike other studies, our approach explicitly focuses on the notion of architectural viewpoints. This enables a more comprehensive and stakeholder-focused analysis, ensuring that the architecture drift analysis is relevant and effective for all parties involved.
- Integrated and validated within an industrial context: Our method is both complementary to existing approaches and novel, as it offers an integrated viewpoint-oriented approach that has been validated within an industrial context. This demonstrates the practical applicability and relevance of our approach to real-world software systems.
- Automated analysis: Most of the studies in the related work use either manual or semi-automated checking, while our approach emphasizes automated analysis for efficiency and scalability, particularly for large and complex software projects.
- Formal case study research protocol: Our approach is one of the few that has been evaluated using a formal case study research protocol, which strengthens the reliability and validity of our findings.

10. Conclusion

Software systems are seldom static, frequently requiring adaptations to address bugs or accommodate new requirements. The architectural drift problem, prevalent in many software projects, signifies the discrepancy between the architectural description and the resulting implementation [14]. Addressing the architectural drift problem to ensure alignment between the code and the architecture is crucial for guiding and managing software projects. In this article, we introduce a comprehensive, model-driven architecture conformance analysis approach to assess the consistency between architectural views and their

corresponding reconstructed views derived from the code. We have employed five distinct viewpoints, each with their respective meta-models and domain-specific languages. For every viewpoint, we have implemented a software architecture reconstruction method, a viewpoint-oriented software architecture conformance analysis, and a supporting toolset. We have applied our approach and tools to a case study involving a commercial company. In conclusion, our method serves as a complementary addition to existing approaches, offering a novel, integrated, and viewpoint-oriented solution that has been validated in an industrial context. Both the approach and the toolset are now in use by the company featured in the case study. In our future work, we aim to apply our approach and tools to various other case studies.

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.

Data availability

No data was used for the research described in the article.

References

[1] L. Bass, I. Weber, L. Zhu, DevOps: A software Architect's Perspective, Addison-Wesley Professional, 2015.
[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond. Second Edition, Addison-Wesley, Reading, MA, 2011.
[3] B. Tekinerdogan, Software Architecture, in: T. Gonzalez, J.L. Díaz Herrera (Eds.), Computer Science Handbook, Second Edition, Volume I: Computer Science and Software Engineering, Taylor and Francis, 2014.
[4] M. Abi-Antoun, J. Aldrich, Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations, in: Proc. of the 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2009.
[5] A. Bucchiarone, D. Di Ruscio, I. Malavolta, P. Pelliccione, M. Tivoli, Towards a model-driven infrastructure for runtime integration, validation and execution of formal tools, Sci. Comput. Program. 144 (2017) 57–84.
[6] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2013.
[7] S. Lee, J.C. Carver, An empirical study of architectural decay in open-source software, Inf. Softw. Technol. 84 (2017) 85–101.
[8] J. Perez, B. Rumpe, A comparison of static architecture compliance checking approaches, Softw. Syst. Model. 17 (4) (2018) 1105–1132.
[9] M. Plösch, R. Weinreich, C. Körner, A method for continuous code quality management using static analysis, J. Syst. Softw. 133 (2017) 275–294.
[10] E. Demirli, B. Tekinerdogan, Software Language Engineering of Architectural Viewpoints, in: Proc. of the 5th European Conf. on Software Architecture (ECSA 2011), 2011, pp. 336–343.
[11] R. Terra, M.T. Valente, A dependency constraint language to manage object-oriented software architectures, Softw.: Pract. Exper. 39 (12) (2009) 1073–1094.
[12] B. Uzun, B. Tekinerdogan, Domain-driven Analysis of Architecture Reconstruction methods. Model Management and Analytics for Large Scale Systems, Academic Press, 2020, pp. 67–84.
[13] G.C. Murphy, D. Notkin, K.J. Sullivan, Software reflexion models: bridging the gap between design and implementation, IEEE Trans. Softw. Eng. 27 (4) (2001) 364–380, 2001.
[14] Architecture Reconstructor, https://github.com/burakuzn/ArchitectureReconstructor, last accessed on May 2023.
[15] B. Tekinerdogan, E. Demirli, Evaluation Framework for Software Architecture Viewpoint Languages, in: Proc. of Ninth International ACM Sigsoft Conference on the Quality of Software Architectures Conference (QoSA 2013), Vancouver, Canada, 2013, pp. 89–98. June 17-21.
[16] Epsilon Comparison Language, https://www.eclipse.org/epsilon/doc/ecl/, last accessed on May 2023.
[17] Java runtime metadata analysis, https://github.com/ronmamo/reflections, last accessed on May 2020.
[18] Architecture Model Differentiator, https://github.com/burakuzn/ArchitectureModelDifferentiator, last accessed on May 2023.
[19] J. Rosik, A. Le Gear, J. Buckley, M. Ali Babar, An industrial case study of architecture conformance, in: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 80–89.
[20] L.J. Morell, A theory of fault-based testing, IEEE Trans. Softw. Eng. 16 (8) (1990) 844–857.
[21] R. Weinreich, G. Buchgeher, Automatic reference architecture conformance checking for soa-based software systems, in: 2014 IEEE/IFIP Conference on Software Architecture, IEEE, 2014, pp. 95–104.
[22] D.C. Marinescu, A.R. Leitner, B.E. Petzke, Architecture conformance testing using model checking and data mining, Softw. Qual. J. 25 (1) (2017) 313–341.
[23] D.M. Rafi, K. Moses, K. Petersen, M.V. Mäntylä, Benefits and limitations of automated software testing: systematic literature review and practitioner survey, in: Proceedings of the 7th InternationalWorkshop on Automation of Software Test (AST); Zurich, Switzerland, 2012.
[24] Y. Jia, M. Harman, Higher order mutation testing, Inf Softw Technol 51 (10) (2009) 1379–1393.
[25] F. Fittkau, P. Stelzer, W. Hasselbring, Live visualization of large software landscapes for ensuring architecture conformance, in: Proceedings of the 2014 European Conference on Software Architecture Workshops, 2014, pp. 1–4.
[26] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. J. 14 (2) (Dec. 2008) 131–164.
[27] B. Uzun, B. Tekinerdogan, Architecture conformance analysis using model-based testing: a case study approach, Software: Pract. Exper. 49 (3) (2019) 423–448.
[28] B. Tekinerdogan, Architectural drift analysis using architecture reflexion viewpoint and design structure reflexion matrices. Software Quality Assurance, Morgan Kaufmann, Boston, 2016, pp. 221–236.
[29] A. van Deursen, et al., Symphony: view-driven software architecture reconstruction, in: Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), IEEE, 2004.
[30] G. El Boussaidi, et al., Reconstructing architectural views from legacy systems, in: 2012 19th Working Conference on Reverse Engineering, IEEE, 2012.
[31] S. Miranda, E. Rodrigues, M. Valente, R. Terra, Architecture conformance checking in dynamically typed languages, J. Obj. Technol. 15 (3) (2016) 1. -1.
[32] F. Deissenboeck, L. Heinemann, B. Hummel, E. Juergens, Flexible architecture conformance assessment with ConQAT, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering 2, IEEE, 2010, pp. 247–250.
[33] M. Eichberg, S. Kloppenburg, K. Klose, M. Mezini, Defining and continuous checking of structural program dependencies, in: Proceedings of the 30th international conference on Software engineering, 2008, pp. 391–400.
[34] J. Knodel, D. Popescu, A comparison of static architecture compliance checking approaches, in: 2007 Working IEEE/IFIP conference on software architecture (WICSA'07), IEEE, 2007, p. 12. -12.
[35] H. Muccini, M. Dias, D.J. Richardson, Systematic testing of software architectures in the C2 style, in: International Conference on Fundamental Approaches to Software Engineering, Springer, Berlin, Heidelberg, 2004, pp. 295–309.
[36] A. Bucchiarone, H. Muccini, P. Pelliccione, P. Pierini, Model-Checking plus Testing: from Software Architecture Analysis to Code Testing, in: International Conference on Formal Techniques for Networked and Distributed Systems, Springer, Berlin, Heidelberg, 2004, pp. 351–365.