Heuristics for Discovering Architectural Violations

Cristiano Maffort*, Marco Tulio Valente*, Mariza Bigonha*,
Nicolas Anquetil†, André Hora†
*Department of Computer Science, UFMG, Brazil
†RMoD Project-Team, INRIA, Lille Nord Europe, France
Email: {maffort,mtov,mariza}@dcc.ufmg.br, {nicolas.anquetil,andre.hora}@inria.fr

***Abstract*—Software architecture conformance is a key software quality control activity that aims to reveal the progressive gap normally observed between concrete and planned software architectures. In this paper, we present ArchLint, a lightweight approach for architecture conformance based on a combination of static and historical source code analysis. For this purpose, ArchLint relies on four heuristics for detecting both absences and divergences in source code based architectures. We applied ArchLint in an industrial-strength system and as a result we detected 119 architectural violations, with an overall precision of 46.7% and a recall of 96.2%, for divergences. We also evaluated ArchLint with four open-source systems, used in an independent study on reflexion models. In this second study, ArchLint achieved precision results ranging from 57.1% to 89.4%.**

***Index Terms*—Software architecture conformance; Static analysis; Mining software repositories.**

I. Introduction

Software architecture conformance is a key software quality control activity that aims to reveal the progressive gap normally observed between concrete and planned software architectures [1], [2]. More specifically, this activity aims to expose statements, expressions or declarations in the source code that do not match the constraints imposed by the planned architecture. The ultimate goal is to prevent the accumulation of such incorrect implementation decisions and therefore to avoid the phenomena known as architectural drift or erosion [3].

There are two main techniques for architecture conformance: reflexion models and domain-specific languages [4]. Reflexion models compare a high-level model manually created by the architect with a concrete model, extracted automatically from the source code [5]. As a result, reflexion models can reveal two kinds of architectural anomalies: absences (relations prescribed by the high-level model that are not present in the concrete model) and divergences (relations not prescribed by the high-level model, but that are present in the concrete model). Alternatively, domain-specific languages with focus on architecture conformance provide means for architects to express in a customized syntax the constraints defined by the planned architecture [6]–[8].

However, the application of the current techniques for architecture conformance may require a considerable effort. For example, reflexion models may require successive refinements in the high-level model to reveal the whole spectrum of absences and divergences that can be present in the source code of large and extensively maintained systems [9], [10]. On the other hand, domain-specific languages may require the extensive and detailed definition of constraints.

This paper presents ArchLint, an approach that combines static and historical source code analysis techniques in order to provide a lightweight alternative for architecture conformance. ArchLint requires two inputs on the system under analysis: a high-level component specification and the history of revisions. Without requiring further refinements in this model, ArchLint supports four heuristics to discover suspicious dependencies in the source code, i.e., dependencies that may denote divergences or absences. The common assumption behind the proposed heuristics is that dependencies denoting architectural violations—at least in systems not facing a massive erosion process—are *rare events in the space-time domain*, i.e., they appear in a small number of classes and they are frequently removed during the evolution of the systems.

We report two case studies with ArchLint. In a first study, we applied the solution in an industrial-strength information system. Our goal was to conduct a sensitivity study, in order to discover the best combination of values for the thresholds required by the proposed heuristics. As a result, we detected 119 violations in the first evaluated system, with a precision of 46.7% and a recall of 96.2%, for divergences. In a second study, we evaluated four open-source systems used in an independent study on reflexion models. For the systems with architectural violations, ArchLint achieved precision results ranging from 57.1% to 89.4% and recall ranging from 16.2% to 93.1%. Such results are similar to the ones produced by general-purpose and widely used static analysis tools, such as FindBugs [11] and PMD [12]. In fact, ArchLint can be seen as an attempt to elevate to an architectural level the warnings raised by popular static analysis tools.

The remainder of this paper is divided into eight sections. In Section II, we present an overview of the proposed approach for architecture conformance. Sections III and IV present the heuristics for absences and divergences, respectively. Section V describes the internal architecture of a prototype tool supporting our approach. Sections VI describe an analysis of sensitivity, considering a real-world information systems. Section VII reports an evaluation with four open-source systems. Section VIII presents related work. Section IX concludes with ArchLint's contributions and limitations. It also outlines future research lines and improvements.

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

II. Proposed Approach

Figure 1 illustrates our approach for detecting architectural violations. Basically, it relies on two types of input information on the target system: (a) history of versions; (b) high-level component specification. We consider that the classes of a system are statically organized in modules (or packages, in Java terms) and that modules are logically grouped in coarse-grained structures, called components. The component model includes information on the names of the components and a mapping from modules to components, using regular expressions (a complete example is provided in Section VI-A). From this, ArchLint identifies suspicious dependencies (or lack of) in source code by relying on frequency hypotheses and past corrections made on these dependencies. ArchLint considers all static dependencies possibly established between classes, including dependencies due to method calls, variable declarations, inheritance, exceptions, etc.

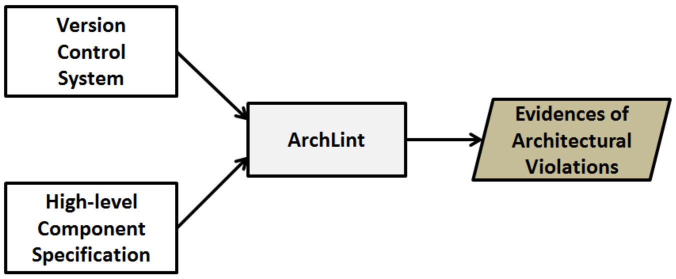

Version Control System
High-level Component Specification
ArchLint
Evidences of Architectural Violations

Fig. 1. Proposed approach

We do not make efforts in inferring the high-level components automatically because it is usually straightforward for architects to provide this representation. When the architects are not available (e.g., in the case of open-source systems), a high-level decomposition in major subsystems is often included in the developer's documentation or can be retrieved by inspecting the package structure. In fact, as described in Section VII, we applied ArchLint to four open-source systems, reusing high-level models independently defined by other researchers from information available in the system's documentation.

In the following sections, we motivate and describe the heuristics followed by ArchLint to detect evidences of absences (Section III) and divergences (Section IV). The heuristics were proposed after our experience on discovering architectural violations in a real-world information system [6], different from the system we evaluate on Section VI.

III. Detecting Absences

An absence is a violation due to a dependency defined by the planned architecture, but that does *not exist* in the source code [1], [5]. For example, suppose an architecture that requires that classes located in a $View$ component must extend a class called $ViewFrame$. In this case, an absence is counted for each class in the $View$ not following this rule.

To detect absences, we initially search for classes denoting minorities at the level of components, regarding a given dependency. Assuming that absences are an exceptional property in classes, minorities have more chances to represent architectural violations. Moreover, we rely on the history of versions to mine for dependencies dep introduced in classes originally created without dep. The underlying assumption in this case is that absences are usually detected and fixed. Moreover, the goal is to reinforce the evidences collected in the previous step by checking whether the classes originally created with the architecture violation under analysis (i.e., absence of dep) have later fixed this violation (i.e., have introduced dep).

Figure 2 illustrates the proposed heuristic. In this figure, class C_2 has an absence regarding $TargetClass$ because: (a) C_2 is the only class in the component cp that does not depend on $TargetClass$; (b) a typical evolution pattern among the classes in cp is to introduce a dependency with $TargetClass$, as observed in classes C_1, C_4, and C_5.

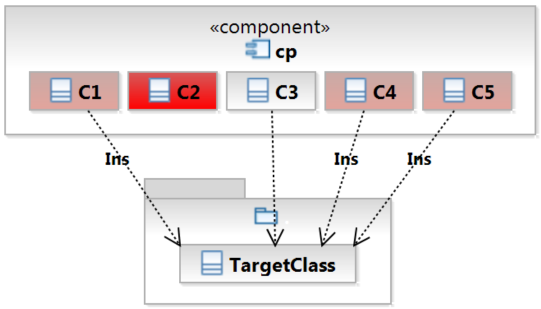


Fig. 2. Example of absence ($C2$ does not depend on $TargetClass$)

Formal Definition: Our heuristic for detecting absences, relies on two definitions. First, the *Dependency Scattering Rate*—denoted by $DepScaRate(c, cp)$—is the number of classes in the component cp that depend on a target class c divided by the total number of classes in cp. Second, the *Dependency Insertion Rate*—denoted by $DepInsRate(c, cp)$—is the number of classes in the component cp originally created without a dependency with a target class c but that have this dependency in the last version of the system under analysis divided by the total number of classes in cp originally created without a dependency with c.

The candidates for absences in a given component cp are defined as follows:

$$Absences(cp) = \{\ (x, c) \mid comp(x) = cp\ \wedge$$
$$\neg depends(x, c)\ \wedge\ DepScaRate(c, cp) \geq A_{sca}\ \wedge$$
$$DepInsRate(c, cp) \geq A_{ins}\}$$

According to this definition, an absence is a pair (x, c) where x is a class located in cp that does not depend on the target class c, when most of the classes in cp have this dependency. Moreover, several classes in cp created without this dependency evolved to depend on c. The parameters A_{sca} and A_{ins} define the thresholds for dependency scattering and insertion.

IV. Detecting Divergences

A divergence is a violation due to a dependency that is not allowed by the planned architecture, but that *exists* in the source code [1], [5]. Our approach includes three heuristics for detecting divergences, as described next.

A. *Heuristic #1*

Essentially, this heuristic targets a common pattern of divergences: the use of frameworks and APIs by unauthorized components [6], [13]. For example, enterprise software architectures commonly define that object-relational mapping frameworks must only be accessed by components in the persistence layer. Therefore, this constraint explicitly authorizes the use of an external framework, but only in well-defined components.

The heuristic initially prescribes that the search for divergences must be restricted to dependencies present in a small number of the classes of a given component (according to a given threshold, described next). However, although this is a necessary condition for divergences, it is not sufficient to characterize such violations. For this reason, the heuristic includes two extra conditions. First, it establishes that the dependency must have been removed several times from the high-level component under analysis (i.e., along the component's evolution, it has been detected as a violation and the system has been refactored to fix the violation; but it has been introduced again, possibly by another developer in another package or class that is part of the component). Second, the heuristic also searches for components where the dependency under analysis is extensively found (i.e., components that behave as "heavy-users" of the target module). The assumption is that it is common to have coarse-grained and lexically related groups of classes that according to the architecture must only be accessed by classes in well-delimited components.

Figure 3 illustrates the proposed heuristic. In this figure, class C_2 presents a divergence regarding $TargetModule$ because: (a) C_2 is the only class in component cp_1 that depends on $TargetModule$; (b) many classes in cp_1 (like C_1, C_4, and C_5) have in the past established and then removed a dependency with $TargetModule$; (c) most of the dependencies with $TargetModule$ come from another component cp_2 (i.e., cp_2 is a "heavy-user" of $TargetModule$).

Formal Definition: This heuristic relies on two new definitions. First, the *Dependency Deletion Rate* of a component *cp* regarding a target module m—denoted by *DepDelRate*(m, cp)—is the number of classes in *cp* that have established a dependency in the past with a class in m but do not have this dependency anymore in the current version divided by the total number of classes in *cp* that have established a dependency with module m. Second, the predicate $heavyUser(cp, m)$ checks whether most classes in component *cp* depend on classes located in the module m (as defined by a threshold D_{hvy}).

The candidates for divergences in a component cp_1 are defined as follows:

$$Div_1(cp_1) = \{\ (x, c) \mid comp(x) = cp_1\ \wedge\ mod(c) = m\ \wedge$$
$$depends(x, c)\ \wedge\ DepScaRate(c, cp_1) \leq D_{sca}\ \wedge$$
$$DepDelRate(m, cp_1) \geq D_{del}\ \wedge$$

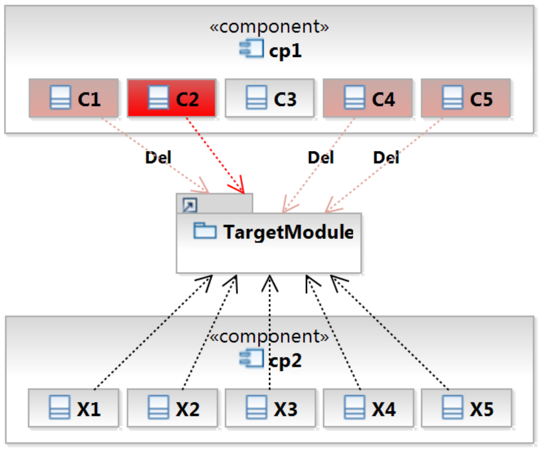


Fig. 3. Example of divergence ($C2$ depends on $TargetModule$)

$$heavyUser(cp_2, m)\ \wedge\ cp_1 \neq cp_2\}$$

According to this definition, a divergence is a pair (x, c), where x is a class located in the component cp_1 that depends on a target class c, located in a module m, when most of the classes in cp_1 do not have this dependency (as defined by the scattering rate inferior to a minimal threshold D_{sca}). Moreover, the definition requires that several classes in the component under evaluation must have removed the dependencies with m in the past, as defined by a threshold D_{del}. Finally, there is another component cp_2 with a heavy-user behavior with respect to module m.

B. *Heuristic #2*

As in the previous case, this second heuristic restricts the analysis to dependencies originating from a small number of the classes of a given component and to dependencies that have been removed in the past. However, it has two important differences regarding the first heuristic: (a) it is based on fine-grained dependencies, i.e., dependencies to a specific target class and not to whole modules; (b) it does not require the existence of a heavy-user for the dependency under analysis.

Figure 4 illustrates the proposed heuristic. In this figure, class C_2 has a divergence regarding $TargetClass$ because: (a) C_2 is the only class in the component *cp* that depends on $TargetClass$; (b) a common evolution pattern among the classes in *cp* is to remove a dependency with $TargetClass$, as observed in the history of the classes C_1, C_4, and C_5.

This heuristic has been proposed to detect two possible sources of divergences: (a) the occasional use of frameworks not authorized by the planned architecture (e.g., a system that in particular locations relies directly on SQL statements instead of using the object-relational mapping framework prescribed by the architecture) [6]; (b) the use of incorrect abstractions provided by an authorized framework (e.g., a system that in particular locations relies on inheritance instead of annotations when accessing a framework that provides both forms of reuse, although the architecture authorizes only the

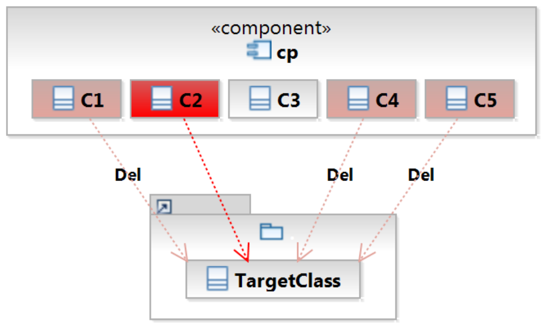


Fig. 4. Example of divergence ($C2$ depends on $TargetClass$)

later).

Formal Definition: This heuristic relies on the *Dependency Deletion Rate*, defined when formalizing the first heuristic. However, for this second heuristic we count deletions regarding a target class c (and not a module m). The heuristic is formalized as follows:

$$Div_2(cp) = \{\ (x, c) \mid comp(x) = cp\ \wedge$$
$$depends(x, c)\ \wedge\ DepScaRate(c, cp) \leq D_{sca}\ \wedge$$
$$DepDelRate(c, cp_2) \geq D_{del}\ \}$$

According to this definition, a divergence is a pair (x, c), where x is a class located in the component cp that depends on a target class c, when most of the classes in cp do not have this dependency (as defined by the threshold D_{sca}). Moreover, several classes in the component under evaluation removed the dependencies with c in the past.

C. Heuristic #3

This heuristic is based on the assumption that a common consequence of divergences is the creation of asymmetrical cycles between components. More specifically, as illustrated in Figure 5, our goal with this heuristic is to search for pairs of components cp_1 and cp_2 where most references are from cp_2 to cp_1, but there is also a representative number of references in the reverse direction. The underlying assumption is that in the original architecture such components have been designed to communicate unidirectionally and the dependencies in the "wrong direction" are in fact architectural violations (and not exceptions authorized by the architecture for example for performance reasons). This heuristic is particularly useful to detect *back-call* violations, a typical violation in layered architectures that happens when a lower layer relies on services implemented by upper layers.

Formal Definition: To evaluate the third heuristic for divergences, we assume that $rf(cp_1, cp_2)$ denotes the number of references to classes in cp_2 originating from classes in cp_1. We also define the *Dependency Direction Weight* between components cp_1 and cp_2 as follows:

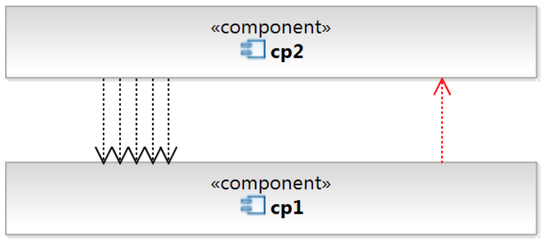


Fig. 5. Divergences due to asymmetrical cycles

$$DepDirWeight(cp_1, cp_2) = \frac{rf(cp_1, cp_2)}{rf(cp_1, cp_2) + rf(cp_2, cp_1)}$$

Using this definition, the heuristic is formalized as follows:

$$Div_3(cp_1) = \{\ (c_1, c_2) \mid depends(c_1, c_2)\ \wedge$$
$$comp(c_1) = cp_1\ \wedge\ comp(c_2) = cp_2\ \wedge\ cp_1 \neq cp_2\ \wedge$$
$$D_{dir} \leqslant DepDirWeight(cp_1, cp_2) < 0.5\}$$

According to this formalization, divergences are pairs of classes (c_1, c_2) where c_1 is a class in cp_1 (the component under analysis) that depends to a class c_2 in cp_2 and the dependencies from cp_1 to cp_2 meet the following conditions: (a) they are not exceptions, because they are greater than a minimal threshold D_{dir}; (b) they are not dominant, because there are more dependencies in the reverse direction, as specified by a dependency direction weight less than 0.5.

V. Tool Support

We have implemented a prototype tool that supports the ArchLint approach for detecting architectural violations. As represented in Figure 6, ArchLint's implementation follows a pipeline architectural pattern with three main components:

- The *Code Extractor* module is responsible for extracting the source code of all versions of the system under evaluation. Currently, our prototype provides access only to `svn` repositories.

- The *Dependency Extractor* is responsible for creating a dependency model describing the structural relations available in each source code version considered in the evaluation. Essentially, this model is a directed graph, whose nodes are classes and the edges are the dependencies. To extract the dependencies from the source code, we use VerveineJ[1], a Java parser that exports dependency relations in the format for modeling static information defined by the Moose platform for software analysis[2]. We modified VerveineJ to store this information in a relational database, in order to facilitate queries over the collected data.

[1]https://gforge.inria.fr/projects/verveinej.
[2]http://www.moosetechnology.org.

- The *Architectural Violations Detector* module implements the heuristics described in Sections III and IV. Basically, the heuristics are implemented as SQL queries.

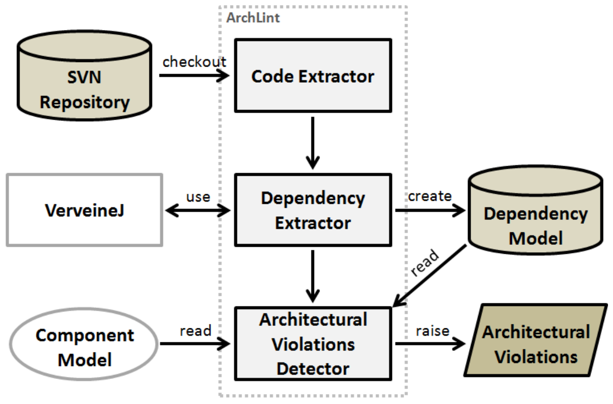


Fig. 6. ArchLint architecture

VI. First Study - Sensitivity Analysis

To start evaluating our approach, we conducted a first study using a real-world information system. Our goal with this first study is twofold: (a) to analyze the sensitivity of our approach to changes in the values of the thresholds used by the proposed heuristics; (b) to determine a set of thresholds to be used in the second case study reported in the paper.

A. Methodology

In this first study we use ArchLint to detect violations in the architecture of an EJB-based information system used by a major Brazilian university, which for confidentiality reasons we will just call SGA. The system includes functionalities for human resource management, finance and accounting management, and material management, among others. In the study, we considered 4,923 revisions (all available revisions), stored in a `svn` repository, from March, 2009 to September, 2011. After parsing these revisions, ArchLint generated a dependency model with more than 29 million relations, which have been stored in a relational database with 4.5 GB. The first extracted version was considered as the baseline for calculating the functions *DepInsRate* and *DepDelRate*, described in Sections III and IV. The last revision considered in the study has 1,852 classes and interfaces, comprising around 127 KLOC.

To apply ArchLint, we initially asked SGA's senior architect to define the high-level component model. After a brief explanation on the purpose and characteristics of this model, the architect suggested the following components:

- $ManagedBean$: bridge between user interface and business-related components.
- $IService$: facade for the service layer.
- $ServiceLayer$: core business processes automated by the system.
- $IPersistence$: facade for the persistence layer.
- $PersistenceLayer$: implementation of persistence.
- $BusinessEntity$: domain types used in the system (such as Professor, Student, etc).

Table I shows the number of packages and classes in the high-level components defined by the SGA's architect. As can be observed, the proposed components are coarse-grained structures, ranging from components with 16 packages and 304 classes (*ManagedBean*) to components with 21 packages and 357 classes (*BusinessEntity*). The table also shows the regular expressions proposed by the architect to define the packages in each component. We can observe that most expressions are simple, usually selecting packages that have common names or prefixes.

TABLE I
High-level components in the SGA system

Component	Packages	Classes	Regular Expression
ManagedBean	16	304	br.sga*.managedbeans*
IService	17	308	br.sga*.ejb.facade*
ServiceLayer	17	311	br.sga*.ejb.local*
IPersistence	18	315	br.sga*.dao* <excludes> br.sga*.dao.jpa*
Persistence	17	309	br.sga*.dao.jpa*
BusinessEntity	21	357	br.sga*.domain*

Using as input the regular expressions specifying the components, we executed ArchLint to trigger evidences of architectural violations in the SGA system. In fact, we executed the tool several times, varying the thresholds required by the heuristics proposed in Sections III and IV. Basically, the intention was to check how our results are affected by setting different values for such thresholds. We asked the architect to carefully examine the triggered violations and to classify them as true or false positives. Since the architect has a complete domain of SGA's architecture and implementation, he is the right expert to play an oracle role in our study.

In the case of divergences, to measure recall we asked the architect to inspect and classify all dependencies established between the classes in the last version considered in the evaluation (which was a not simple task, but at least it was manageable by setting several filters in a spreadsheet with all existing dependencies).

B. Detecting Absences

Figures 7(a) and 7(b) show the number of true absences (true positives) and false absences (false negatives) raised by ArchLint. As reported in Section III, the detection of absences relies on two thresholds: A_{sca} (percentage of classes the dependency is scattered on) and A_{ins} (percentage of classes that inserted the dependency in the past). Figure 7(a) shows the results when we have set up A_{sca} as 0.90 and ranged A_{ins} from 0.35 to 0.95 (in intervals of 10%). In Figure 7(b), we fixed A_{sca} in 0.95 and ranged A_{ins} in the same way.

As presented in the figures, there is a major reduction in the number of true positives when we vary A_{ins}. In Figure 7(a), for example, we have achieved 43 true absences for $A_{ins} = 0.35$ and only two true results for $A_{ins} = 0.95$. On the other hand, there is also an important reduction in the number of true positives when we compare both figures. For example, after increasing A_{sca} to 0.95, we achieved only 28 true positives for $A_{ins} = 0.35$.

Figure 7(c) shows the precision for each combination of thresholds. As can be observed, the precision has varied from 15.4% to 57.1%. The best precision was achieved for $A_{sca} = 0.95$ and $A_{ins} = 0.35$.

C. Detecting Divergences

Heuristic #1: As reported in Section IV-A, this heuristic depends on two thresholds: D_{sca} (percentage of classes the dependency is scattered on) and D_{del} (percentage of classes that removed the dependency in the past). Figures 8(a), 8(b), and 8(c) show the results after ranging D_{sca} from 0.15 to 0.05 and D_{del} from 0.55 to 0.95. As can be checked, we counted exactly nine true positives in each combination of thresholds we have tested. On the other hand, after setting $D_{sca} = 0.05$, we were able to reduce to zero the number of false negatives.

Figure 9 presents the precision results, which have varied from 36% (for $D_{sca} = 0.15$ and $D_{del} = 0.55$) to 100% (for $D_{sca} = 0.05$ and $D_{del} \geqslant 0.55$).

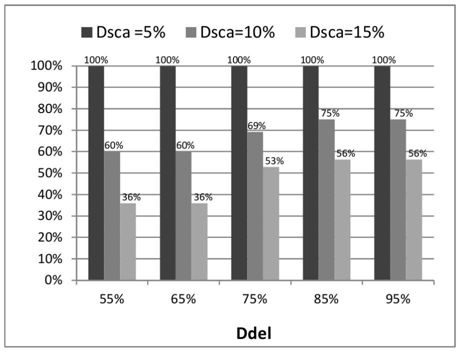


Fig. 9. Precision for the heuristic #1

Heuristic #2: As in the previous heuristic, this second heuristic relies on D_{sca} and D_{del} thresholds. Figure 10(a) shows the results achieved when we set up D_{sca} as 0.10 and ranged D_{del} from 0.55 to 0.95 (in intervals of 10%). In Figure 10(b), we fixed D_{sca} in 0.05 and ranged D_{del} in the same way. We observed a major decrease in the number of false positives when we varied D_{del} values (from 132 to 13 false positives, as presented in Figure 10(a)). On the other hand, changes in D_{sca} have not impacted the number of true positive—which are exactly the same in both figures.

Figure 10(c) presents the precision for each combination of thresholds we have tested. The precision has varied from 18.4% (for $D_{sca} = 0.10$ and $D_{del} = 0.55$) to 63.9% (for $D_{sca} = 0.10$ and $D_{del} = 0.95$).

Heuristic #3: This heuristic relies on the D_{dir} threshold, which defines the minimal percentage of dependencies that characterizes an asymmetrical cycle. Figure 11 shows that the heuristic started to generate true results for $D_{dir} \geqslant 0.05$. More specifically, we have counted exactly the same 50 true positives and zero false negatives for D_{dir} equals to 0.05 and 0.15.

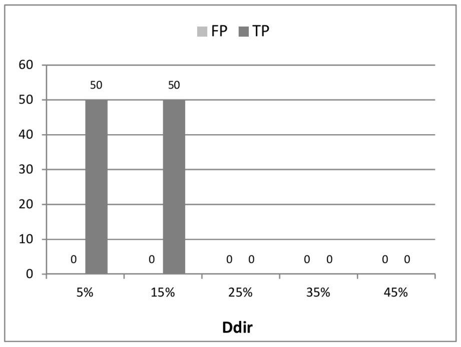


Fig. 11. Divergences detected by heuristic #3

D. Combining the Heuristics

In the previous subsections, we reported the results achieved by the heuristics individually, by considering different threshold values. Based on this sensitivity study, we decided to establish and to consider in the remaining of this paper the following combination of thresholds:

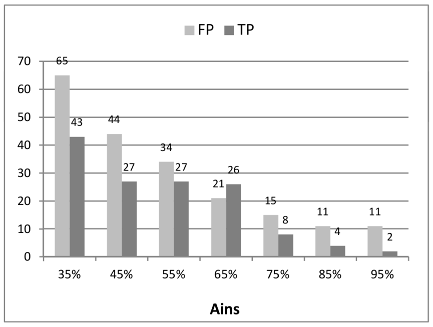
- Absences: $A_{sca} = 0.9$ and $A_{ins} = 0.35$.
- Divergences: $D_{sca} = 0.05$, $D_{del} = 0.65$, $D_{dir} = 0.05$.

To make this selection, we first defined that a given threshold value must have achieved a minimal precision of 25% to be considered for selection. Second, considering the values attending this first condition, we selected the thresholds with the highest number of true positives. Finally, in the case of ties we selected the most flexible threshold value.

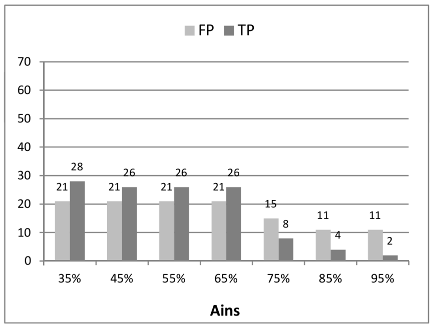
By assuming the proposed combination of thresholds, Table II presents the precision and recall achieved by our approach[3]. As can be observed, ArchLint achieved a precision of 39.8% for absences and 51.7% for divergences. These precision values are similar to the ones generated for example by FindBugs [11] and PMD [12], which are well-known bug finding tools based on static analysis. For example, in a previous study, using as target five stable releases of the Eclipse platform, we found that precision rates superior to 50% are only possible by restricting the analysis to a small subset of the warnings raised by FindBugs (basically, high priority warnings from the correctness category) [14]. For PMD, the precision was inferior to 10%. In another study, Kim and Ernst using a strict definition for false positives achieved a precision of less than 12% for the warnings raised by FindBugs [15]. Clearly, such tools have different purposes than ArchLint, but our intention here is to show that developers accept false warnings when using static analysis tools.

For divergences, ArchLint achieved a recall of 96.2%, missing only three divergences. We did not measure recall for absences, because it would require finding the whole set

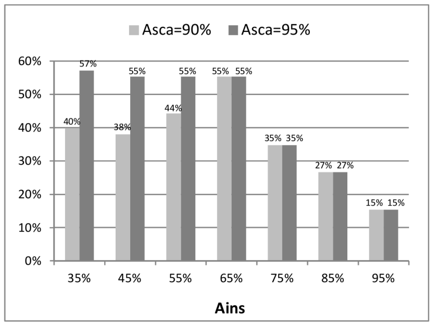
[3]The true and false positives achieved by each heuristic individually are presented in Figures 7 to 10.



(a) $A_{sca} = 0.90$

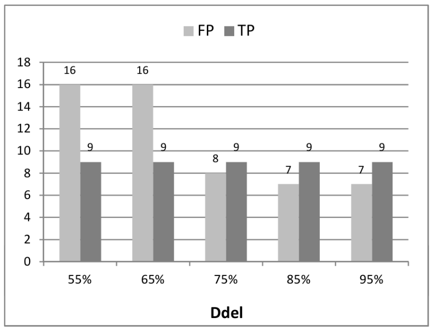


(b) $A_{sca} = 0.95$

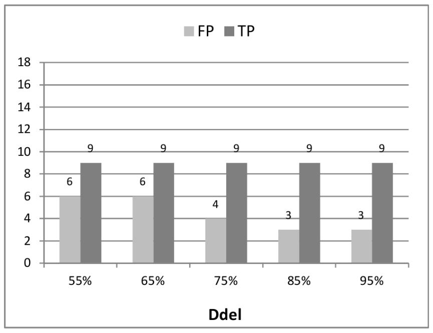


(c) Precision

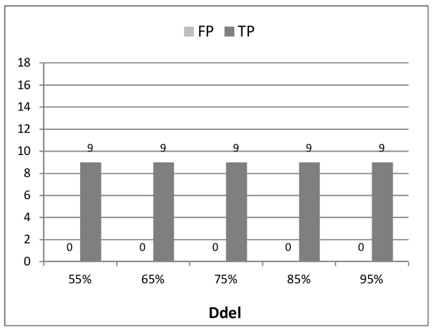
Fig. 7. Absence violations (FP = false positives; TP = true positives)



(a) $D_{sca} = 0.15$

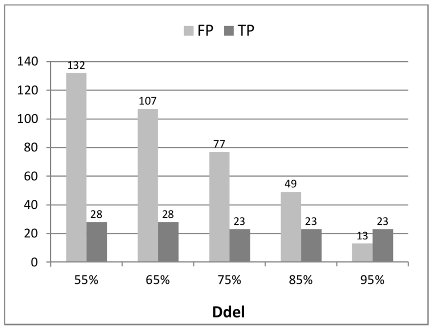


(b) $D_{sca} = 0.1$

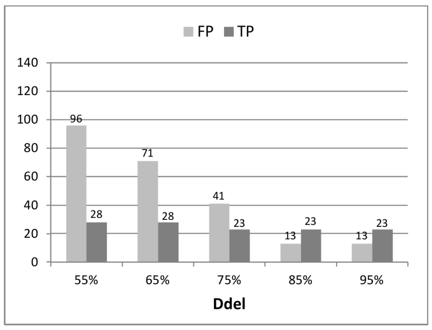


(c) $D_{sca} = 0.05$

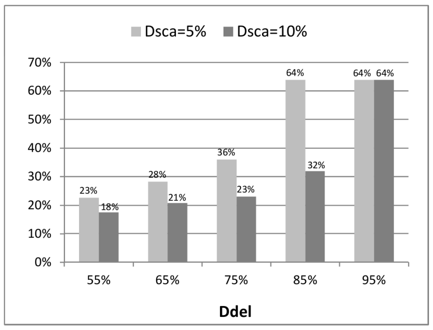
Fig. 8. Divergences detected by heuristic #1



(a) $D_{sca} = 0.1$



(b) $D_{sca} = 0.05$



(c) Precision

Fig. 10. Divergences detected by heuristic #2

TABLE II
OVERALL PRECISION AND RECALL

	Absences	**Divergences**	**Total**
Warnings	108	147	255
True Positives	43	76	119
False Positives	65	71	136
False Negatives	-	3	-
Precision	39.8%	51.7%	46.7%
Recall	-	96.2%	-

of missing dependencies, which in practice requires a detailed and complete inspection in the source code.

Figure 12 factors out the true results achieved by each heuristic proposed to detect divergences. The heuristic #3 was the one responsible for detecting more divergences (48 out of the 76 true results we achieved for divergences). On the other hand, the heuristic #1 did not contribute to the detection of a single violation that was also not found by the other heuristics. Therefore, the factorization suggests that the heuristic #1 is a particular case of the heuristic #2. However, this result in fact reflects our particular thresholds settings; for example, for other thresholds values, the heuristic #1 has found violations not detected by the remaining heuristics. For this reason, we decided to preserve this heuristic in ArchLint.

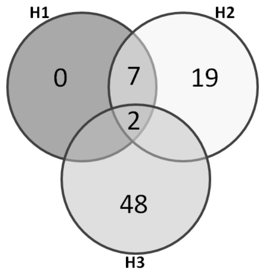


Fig. 12. Divergences detected by each heuristic

E. Threats to Validity

We relied on an architect to design our initial model and to classify our warnings. Therefore, as any human-made artifact, the model and the classification are subjected to errors and imprecision. However, we interviewed an experienced architect, with a complete domain of SGA's architecture and implementation. Furthermore, one can argue that this architect might be influenced to design a model favoring ArchLint. However, we never explained to the architect the heuristics followed by ArchLint to discover architectural violations.

VII. Second Study: Reflexion Models Dataset

In this section, we report the application of ArchLint in four open-source systems: Ant, ArgoUML, Lucene, and SweetHome3D. These systems—which we collectively call the RM Dataset—have RM-based high-level models independently proposed by Bittencourt et al. [16]. Therefore, we reused the component specifications from the high-level models defined in this dataset as the input for ArchLint. Basically, our goal was to evaluate ArchLint in other systems, but using the same threshold values suggested after our first study.

Table III shows the number of revisions and the time frame of the historical data we have extracted from the `svn` repository of the mentioned systems. The table also presents the number of components in the high-level models reused from the RM dataset.

TABLE III
Systems in the RM Dataset

System	# Revisions	# Months	# Components
Ant	6664	151	16
ArgoUML	10938	175	19
Lucene	1959	28	7
SweetHome3D	9146	82	9

Since the RM dataset was carefully designed for architecture conformance purposes, we consider the reflexion models defined in the dataset as a ground-truth architecture for evaluating the precision and recall of ArchLint. More specifically, we classify a warning reported by ArchLint as a true positive when it is also reported in the reflexion models computed using the information available in the RM dataset.

A. Results for Divergences

Table IV reports the precision achieved by ArchLint for divergences. For the first three systems (Ant, ArgoUML, and Lucene) we achieved a precision greater than 57%. For SweetHome3D, ArchLint has not reported a single true positive, but the number of warnings was also small (only seven warnings).

TABLE IV
Precision for divergences in the RM dataset

System	Warnings	TP	FP	Precision
Ant	35	27	8	77.1%
ArgoUML	42	24	18	57.1%
Lucene	160	143	17	89.4%
SweetHome3D	7	0	7	0.0%

Table V reports the results for divergences, but in terms of recall (considering as false negatives the violations reported in the reflexion models but that have not been detected by ArchLint). For the Ant system, we achieved the highest recall result (93%). On the other hand, for SweetHome3D it was not possible to calculate a recall because the system does not have divergences, as reported by the RM technique. Therefore, for a system without divergences, ArchLint reported only seven false positives.

TABLE V
Recall for divergences in the RM dataset

System	RM	ArchLint	Recall
Ant	29	27	93.1%
ArgoUML	148	24	16.2%
Lucene	312	143	45.8%
SweetHome3D	0	0	-

For the Lucene system, ArchLint missed many divergences with a high scattering and a low deletion rate, which explains the relative low recall rate (45.8%). For example, the high-level model defined in the RM Dataset *does not* define a dependency between the components *Search* and *Store*. However, 81 dependencies like that are scattered in 32% of the classes in *Store*, which exceeds by a large margin the threshold for divergences assumed by ArchLint (D_{sca}= 0.05). Moreover, only 6% of such dependencies have been removed along Lucene's evolution (whereas our D_{del} is 0.65). Stated otherwise, in the Lucene system, it is common to observe divergences that are not spatially and historically confined in their source components. A similar observation can be made for ArgoUML, where the divergences missed by ArchLint have also a low deletion rate. Therefore, we argue that both Lucene's and ArgoUML's architecture might have evolved during the time frame considered in our study. As as result, many dependencies not authorized by the initial high-level model have turned themselves into a frequent and enduring property of the systems.

B. Results for Absences

The reflexion models have not indicated absences in the evaluated systems. On the other hand, ArchLint has detected seven absences in the ArgoUML system. In fact, because the source and target components of these absences are internal to the same high-level component (*Explorer*) they were not detected by the reflexion model. In the remaining systems, as with reflexion models, ArchLint has not found absences.

C. Threats to Validity

As in the case of the models we used for the SGA system, it is possible that the high-level models defined in the RM dataset do not capture some violations. However, we argue that the chances are reduced, since the models were carefully designed and then refined to create a benchmark for architecture conformance. Furthermore, missing divergences would impact negatively our recall results. Regarding precision, we consider that the chances of having false positives are also reduced, since the proposed component relations were carefully defined by the dataset's authors.

VIII. Related Work

We divided related work into three groups: static analysis tools, software repository analysis tools, and architecture conformance tools. The tools in the first two groups detect program anomalies, but not at the architectural level. The tools in the third group target architectural anomalies, but are not based on static or historical analysis techniques.

A. Static Analysis Tools

Starting with the Lint tool [17] in the late seventies, several tools have been proposed to detect suspicious programming constructs by means of static analysis, including PREfix/PREfast [18] (for programs in C/C++), and FindBugs [11] and PMD [12] (for programs in Java). Such tools rely on static analysis to detect problematic programming constructs and events, such as uncaught exceptions, null pointer dereferences, overflow in arrays, synchronization pitfalls, security vulnerabilities, etc. Therefore, they are not designed to detect architectural anomalies, such the ones associated to violations in the planned architecture of object-oriented systems.

The dissemination of static analysis tools has motivated the empirical evaluation of the relevance of the warnings raised by such tools. For example, in a previous study, using as target five stable releases of the Eclipse platform, we measured the precision of the warnings raised by two Java-based bug finding tools [14]. We defined precision by the following ratio: (#warnings removed after a given time frame) / (#warnings issued by the tool). We found that precision rates superior to 50% are only possible by restricting the analysis to a small subset of the warnings raised by FindBugs (basically, high priority warnings from the correctness category). For PMD, the precision was less than 10%. In another study, Kim and Ernst define precision in a different way: (#warnings on bug-related lines) / (#warnings issued by the tool) [15]. Using this strict definition, the precision was less than 12%. Therefore, precision ranging from 46.7% (SGA System) to 89.4% (Lucene) as the ones we achieve with ArchLint seems to be greater than the values typically provided by traditional static analysis tools.

B. Software Repository Analysis Tools

Many tools have been proposed to extract programming patterns from software repositories. DynaMine is a tool that analyzes source code check-ins to discover application-specific coding patterns, like highly correlated method calls [19]. BugMem [20] and FixWizard [21] are tools that mine for repeated bug fix changes in a project's revision history (e.g., changes where an incorrect condition is replaced by a correct one). Lamarck is a tool that mines for evolution patterns (i.e., not only bug fixes) in software repositories by abstracting object usage into temporal properties [22]. In Lamark, to evaluate the tool effectiveness in detecting errors, precision is defined as: (#code smells and defects) / (#warnings issued by the tool). Using this definition, Lamarck's success rate ranges from 33% to 64%. Hora et al. [23] extract system specific rules from source code history by monitoring how API is evolving with the goal of providing better rules to developers. They focus on structural changes done to support API modification or evolution. Differently from previous approaches they do not focus on just mining bug-fixes or system releases. In common, such approaches adopt a vertical approach for discovering project-specific patterns in software repositories (in contrast with static analysis tools that assume a horizontal approach based on a pre-defined set of bug patterns). ArchLint also relies on a vertical approach, but with focus on architecture conformance.

C. Architecture Conformance Tools

Besides reflexion models, another common solution for architecture conformance is centered on domain-specific languages, such as SCL [24], LogEn [7], DCL [6], Grok [25], Intensional Views [8], and DesignWizard [26]. Certainly, by using such languages it is possible to detect the same absences and divergences than ArchLint. On the other hand, even using a customized syntax, the definition of architectural constraints may represent a burden for software architects and maintainers. For example, in a previous experience with the DCL language, we had to define 50 constraints to provide a partial specification for the architecture of a large information system [6].

In a recent work, we used association rules to mine architectural patterns in versions history [27]. First, our goal was to investigate the automatic generation of architectural constraints in the DCL language. Second, we aimed to propose a theory to explain and support the heuristics used in this paper. In fact, we found that the heuristic for absences and the first two heuristics for divergences can be modeled as a frequent itemset mining problem. On the other hand, the number of association rules produced by frequent itemset mining techniques is considerable.

IX. Concluding Remarks

In this section, we conclude by presenting ArchLint's contributions and limitations. We also comment on future research.

A. Contributions

To the best of our knowledge, ArchLint is the first architecture conformance tool that relies on a combination of static and historical source code analysis. As a result, we provide a lightweight and agile approach for architecture conformance that does not require successive refinements in high-level architectural models neither requires the specification of an extensive list of architectural constraints (as with domain-specific languages). On the other hand, ArchLint can generate false positive warnings, as common in most bug finding tools based on static analysis. We reported the results of applying ArchLint in five real-world systems, with precision results ranging from 46.7% to 89.4% and recall ranging from 16.2% to 93.1% (for divergences).

ArchLint is publicly available at:

$\boxed{\text{http://aserg.labsoft.dcc.ufmg.br/archlint}}$

B. Limitations

One of the limitations of our approach seems to be the high number of thresholds that need to be fixed. This fact was not a problem in practice, because the thresholds are independent which means that they can be fine tuned separately. In practice, we propose to adopt a conservative attitude starting with restrictive values for the thresholds that should give good results (most of the violations reported will be true positives). In a second step, one can loosen the constraints, discovering new violations but also more false positives. One can stop when the level of false positive is deemed too important.

C. Future Work

As future work, we plan to apply ArchLint in other systems and to extend our approach with new heuristics. We also plan to make a qualitative analysis including feedback from architects. We are also working on the integration of ArchLint with ArchFix [28], which is a recommendation tool that suggests refactorings for repairing architectural violations.

Acknowledgments

Our research is supported by CAPES, FAPEMIG, and CNPq. We thank the architect of the SGA system for validating the warnings raised by ArchLint.

References

[1] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. Mendonca, "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[2] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, p. 12.

[3] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[4] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[5] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *3rd Symposium on Foundations of Software Engineering (FSE)*, 1995, pp. 18–28.

[6] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 32, no. 12, pp. 1073–1094, 2009.

[7] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 391–400.

[8] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.

[9] R. Koschke, "Incremental reflexion analysis," in *14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 1–10.

[10] R. Koschke and D. Simon, "Hierarchical reflexion models," in *10th Working Conference on Reverse Engineering (WCRE)*, 2003, pp. 36–45.

[11] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[12] T. Copeland, *PMD Applied.* Centennial Books, 2005.

[13] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a large-scale business application: A case study," *IEEE Software*, vol. 26, pp. 28–35, 2009.

[14] J. E. Araujo, S. Souza, and M. T. Valente, "Study on the relevance of the warnings reported by Java bug-finding tools," *IET Software*, vol. 5, no. 4, pp. 366–374, 2011.

[15] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *15th International Symposium on Foundations of Software Engineering (FSE)*, 2007, pp. 45–54.

[16] R. A. Bittencourt, "Enabling static architecture conformance checking of evolving software," Ph.D. dissertation, Universidade Federal de Campina Grande, 2012.

[17] S. C. Johnson, "Lint: A C program checker," Bell Laboratories, Tech. Rep. 65, dec 1977.

[18] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Software*, vol. 21, no. 3, pp. 92–100, 2004.

[19] B. Livshits and T. Zimmermann, "DynaMine: finding common error patterns by mining software revision histories," in *13th International Symposium on Foundations of Software Engineering (FSE)*, 2005, pp. 296–305.

[20] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *14th International Symposium on Foundations of Software Engineering (FSE)*, 2006, pp. 35–45.

[21] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 315–324.

[22] Y. M. Mileva, A. Wasylkowski, and A. Zeller, "Mining evolution of object usage," in *25th European conference on Object-oriented programming*, 2011, pp. 105–129.

[23] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining system specific rules from change patterns," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 1–10.

[24] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[25] R. C. Holt, "Structural manipulations of software architecture using tarski relational algebra," in *5th Working Conference on Reverse Engineering (WCRE)*, 1998, pp. 210–219.

[26] J. Brunet, D. Guerreiro, and J. Figueiredo, "Structural conformance checking with design tests: An evaluation of usability and scalability," in *27th International Conference on Software Maintenance (ICSM)*, 2011, pp. 143–152.

[27] C. Maffort, M. T. Valente, M. Bigonha, A. Hora, and N. Anquetil, "Mining architectural patterns using association rules," in *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2013, pp. 375–380.

[28] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, pp. 1–36, 2013.