Mining architectural violations from version history

**Cristiano Maffort · Marco Tulio Valente ·
Ricardo Terra · Mariza Bigonha · Nicolas Anquetil ·
André Hora**

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. However, formally specifying an architecture can be difficult, as it must be done by an expert of the system having a high level understanding of it. In this paper, we present a lightweighted approach for architecture conformance based on a combination of static and historical source code analysis. The proposed approach relies on four heuristics for detecting absences (something expected was not found) and divergences (something prohibited was found) in source code based architectures. We also present an architecture conformance process based on the proposed approach. We followed this process to evaluate

C. Maffort (✉)
Department of Computer Engineering, CEFET-MG, Minas Gerais, Brazil
e-mail: cristiano@decom.cefetmg.br

C. Maffort · M. T. Valente · M. Bigonha
Department of Computer Science, UFMG, Minas Gerais, Brazil

M. T. Valente
e-mail: mtov@dcc.ufmg.br

M. Bigonha
e-mail: mariza@dcc.ufmg.br

R. Terra
Department of Computer Science, UFLA, Lavras, Brazil
e-mail: terra@dcc.ufla.br

N. Anquetil · A. Hora
RMoD Team, INRIA, Lille Nord Europe, France

N. Anquetil
e-mail: nicolas.anquetil@inria.fr

A. Hora
e-mail: andre.cavalcante.hora@inria.fr

the architecture of two industrial-strength information systems, achieving an overall precision of 62.7 % and 53.8 %. We also evaluated our approach in an open-source information retrieval library, achieving an overall precision of 59.2 %. We envision that an heuristic-based approach for architecture conformance can be used to rapidly raise architectural warnings, without deeply involving experts in the process.

1 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 (Passos et al. 2010; Knodel and Popescu 2007; Ducasse and Pollet D 2009; Brunet et al. 2014). More specifically, the 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 incorrect implementation decisions and therefore to avoid the phenomenon known as architectural drift or erosion (Perry and Wolf 1992).

There are at least two main techniques for architecture conformance: reflexion models and domain-specific languages. Reflexion models compare a high-level model manually created by the architect with a concrete model, extracted automatically from the source code (Murphy et al. 1995). 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 to express in a customized syntax the constraints defined by the planned architecture (Terra and Valente 2009; Eichberg et al. 2008; Mens et al. 2006). However, the application of current architecture conformance techniques requires a considerable effort. For example, reflexion models may require successive refinements in the high-level models to reveal the whole spectrum of architectural violations (Koschke 2010; Koschke and Simon 2003) and domain-specific languages may require the extensive definition of constraints.

In a previous paper, we presented an approach that combines static and historical source code analysis to provide an alternative technique for architecture conformance (Maffort et al. 2013). The proposed approach includes 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 that are not facing a massive erosion process—are *rare events in the space-time domain*, i.e., they appear in a small number of classes (according to particular thresholds) and they are frequently removed during the evolution of the system (according to other thresholds). In this paper, we extend your previous work by proposing an iterative architecture conformance process, based on the defined heuristics. By following this process, architects can experiment and adjust the thresholds required by the defined heuristics, starting with rigid thresholds. Basically, as the thresholds are made less rigid, more false warnings are generated. Therefore, the architect can finish the conformance activity when enough violations are detected or when the heuristics start to produce too many false positives. We also propose a strategy to rank the generated

warnings, which is used to show first the warnings that are more likely to denote real violations.

We evaluated our work in three systems. First, we applied the proposed conformance process in two industrial-strength information systems. We were able to detect 389 and 150 architectural violations, with an overall precision of 62.7 % and 53.8 %, respectively. We also present and discuss examples of architectural violations detected by our approach and the architectural constraints associated to such violations, according to the systems' architects. Finally, we relied on the proposed conformance process to evaluate the architecture of a well-known open-source system (Lucene). In this case, using as oracle a reflexion model independently proposed by another researcher, we found 264 architectural violations, with an overall precision of 59.2 %.

The remainder of this paper is organized in nine sections and three appendices. In Section 2, we introduce the proposed approach for architecture conformance and the heuristics for detecting absences and divergences, respectively. Section 3 describes the architecture of the prototype tool that supports our approach. Section 4 describes an iterative conformance process, based on the proposed heuristics. Particularly, Sections 5 and 6 describe the usage of this process to evaluate the architecture of two proprietary information systems and an open-source information retrieval library (Lucene). Section 7 discusses the lessons learned with our work. Section 8 presents related work and Section 9 concludes the paper. There are also three appendices, presenting a formal definition of the proposed heuristics (Appendix A), the detailed results of the evaluation of one of the information systems considered in the paper (Appendix B) and the results achieved for Lucene (Appendix C).

2 Heuristics for Detecting Architectural Violations

Figure 1 illustrates the input and output of the proposed heuristics for detecting architectural violations. Basically, the heuristics rely on two types of input information on the target system: (a) history of versions; and (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 a mapping from modules to components, using regular expressions (complete examples are provided in Sections 5.1 and 5.3). Given the component model, the proposed heuristics automatically identify suspicious dependencies (or lack of) in source code by relying on frequency hypotheses and past corrections made on these dependencies. In practice, the heuristics consider all static dependencies between classes, including dependencies due to method calls, variable declarations, inheritance, exceptions, etc.
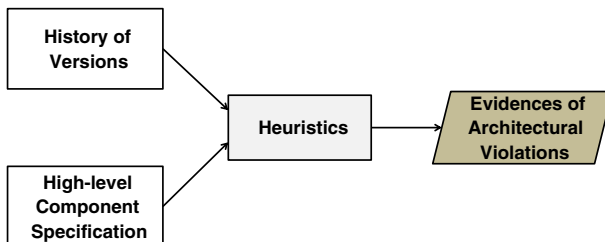

Fig. 1 Input and output of the proposed heuristics

We do not make efforts in automatically inferring the high-level components because it is usually straightforward for architects to provide this representation. When architects are not available (e.g., in the case of open-source systems), a high-level decomposition in major subsystems is often included in developers' documentation or can be retrieved by inspecting the package structure. In fact, as described in Section 6, we applied our approach to an open-source system (Lucene). In this case, we reused high-level models independently defined by other researchers using information available in the Lucene's documentation.

In the following sections, we motivate and describe the heuristics to detect absences (Section 2.1) and divergences (Section 2.2). We also propose a strategy to rank the warnings produced by the heuristics according to their relevance (Section 2.3). A complete formal specification of the heuristics is presented in Appendix A.

2.1 Heuristic for 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 (Murphy et al. 1995; Passos et al. 2010). For example, suppose an architectural rule that requires classes located in a `View` component to extend a class called `ViewFrame`. In this case, an absence is counted for each class in `View` that does not follow this rule.

To detect absences, we initially search for dependencies denoting minorities at the level of components. We assume that absences are an exceptional property in classes and therefore 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 is that absences are usually detected and fixed. The goal is to reinforce the evidences collected in the previous step by checking whether classes originally created with the architectural violation under analysis (i.e., absence of *dep*) were later refactored to include the missing dependency.

Figure 2 illustrates this heuristic for detecting absences. As can be observed, class C_2 has an absence regarding $TargetClass$ because: (a) C_2 is the unique class in component cp that does not depend on $TargetClass$; and (b) a typical evolution pattern among the classes in cp is to introduce a dependency with $TargetClass$, when it does not exist, as observed in the history of classes C_1, C_4, and C_5.
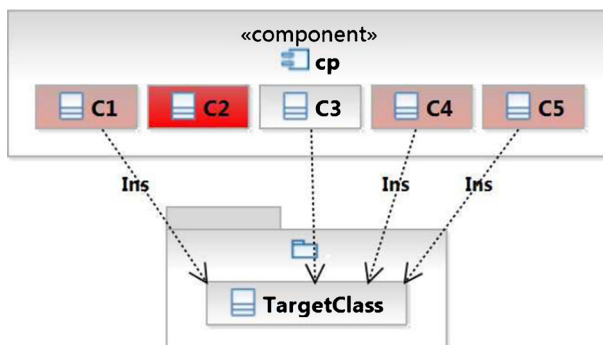

Fig. 2 Example of absence (C_2 does not depend on $TargetClass$). The label *Ins* denotes a dependency inserted later in the class

Additionally, we consider specific types of dependencies. For example, the planned architecture might prescribe that a given *BaseClass* must depend on a *TargetClass* by means of *inheritance*, i.e., *BaseClass* must be a subclass of *TargetClass*. Table 1 reports the types of dependency considered by the heuristic.

Definition The proposed heuristic for detecting absences relies on two definitions:

- Dependency Scattering Rate—denoted by $DepScaRate(c, t, cp)$—is the ratio between (i) the number of classes in component cp that have a dependency of type t with a target class c and (ii) the total number of classes in component cp.
- Dependency Insertion Rate—denoted by $DepInsRate(c, t, cp)$—is the ratio between (i) the number of classes in component cp originally created without a dependency of type t with a target class c, but that have this dependency in the last version of the system under analysis, and (ii) the total number of classes in component cp originally created without the a dependency of type t with class c.

Using these definitions, the candidates for absences in component cp are defined as follows:

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

According to this definition, an absence is a tuple (x, c, t) where x is a class located in component cp that, in the current version of the system in the control version repository (denoted by the symbol H), *does not* include a dependency of type t with the target class c, when most classes in component cp have this dependency. Moreover, several classes in component cp were initially created without this dependency, but have evolved to establish it. Parameters A_{sca} and A_{ins} define the thresholds for dependency scattering and insertion, respectively.

Table 1 Dependency types, assuming that C_1 depends on C_2

Dependency type	Description
AttributeAnnotation	C_2 is used as an annotation over an attribute in C_1
ClassAnnotation	C_2 is used as an annotation over C_1
LocalVariableAnnotation	C_2 is used as an annotation over a local variable in C_1
MethodAnnotation	C_2 is used as an annotation over a method of C_1
ClassAttribute	C_2 is used as an attribute in C_1
CaughtException	C_2 is an exception caught in a method of C_1
DeclaredException	C_2 is an exception declared in a method of C_1
Inheritance	C_2 is the superclass of C_1
LocalVariable	C_2 is used as a local variable in a method of C_1
ParameterizedType	C_2 is used as a generic type in C_1
ReturnMethod	C_2 is the type returned by a method of C_1
ThrownException	C_2 is an exception thrown in a method of C_1

2.2 Heuristics for 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 (Murphy et al. 1995; Passos et al. 2010). Our approach includes three heuristics for detecting divergences, as described next.

2.2.1 Heuristic #1

This heuristic targets a common pattern of divergences: the use of frameworks and APIs by unauthorized components (Terra and Valente 2009; Santonu S et al. 2009). For example, enterprise software architectures commonly define that object-relational mapping frameworks must only be accessed by components in the persistence layer (Fowler 2002). Therefore, this constraint authorizes the use of an external framework, but only by well-defined components.

The heuristic initially defines that the searching for divergences must be restricted to dependencies present in a small number of the classes of a given component (according to a given threshold, as described next). However, although this is a necessary condition for divergences, it is not enough to characterize this violation. For this reason, the heuristic includes two extra conditions: (i) the dependency must have been removed several times from the high-level component under analysis (i.e., along the component's evolution, the system was changed to fix the violation; but it was introduced again, possibly by another developer in another package or class that is part of the component); and, (ii) the heuristic also searches for components where the dependency under analysis is extensively found (i.e., components that act as "heavy-users" of the target module). The assumption is that it is common to have modules that—according to the intended architecture—are only 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
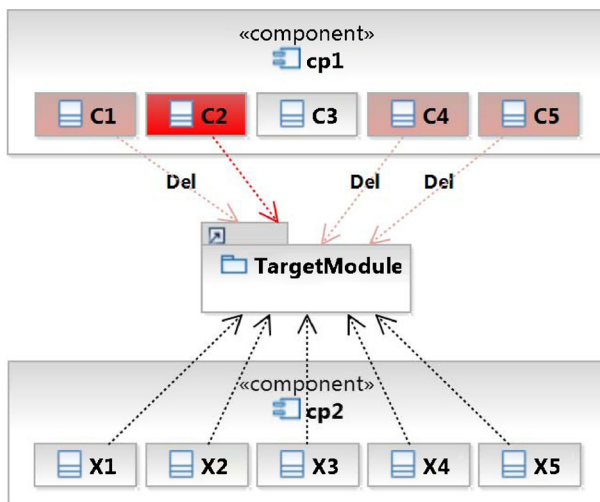

Fig. 3 Example of divergence *(C_2 depends on $TargetModule$)*. The label *Del* denotes a dependency removed in a previous version of the class

on $TargetModule$; (b) many classes in cp_1 (such as C_1, C_4, and C_5) had in the past established and then removed a dependency with $TargetModule$; and (c) most dependencies to $TargetModule$ come from another component cp_2 (i.e., cp_2 is a "heavy-user" of $TargetModule$).

Definition This heuristic relies on two definitions:

- *Dependency Deletion Rate* of a component cp regarding a target module m—denoted by $DepDelRate(m, cp)$—is the ratio between (i) the number of classes in component cp that established a dependency in the past with classes in module m, but no longer have this dependency, and (ii) the total number of classes in component cp that have a dependency with any class in module m. As described before, a module is a set of classes (e.g., a package, in the case of Java systems).

- $HeavyUser(m)$ is a function that return the component whose classes mostly depend on classes located in module m.

  The candidates for divergences in a component cp_1 are defined as follows:[1]

$$
\begin{aligned}
Div_1(cp) = \{\, (x, c) \mid \quad & comp(x) = cp \wedge mod(c) = m \wedge depends(x, c, \_, H) \wedge \\
& DepScaRate(m, cp) \leq D_{sca} \wedge \\
& DepScaRate(m, cp) \leq D_{del} \wedge \\
& HeavyUser(m) \neq cp \,\}
\end{aligned}
$$

According to this definition, a divergence is a pair (x, c), where x is a class located in component cp that depends on a target class c located in a module m, when most classes in component cp do *not* have this dependency (as defined by the scattering rate lower than 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 with a heavy-user behavior with respect to module m.

2.2.2 Heuristic #2

Similarly to the previous case, this second heuristic restricts the analysis to dependencies defined by few classes of a component and that were removed in the past (in other classes of the component). However, this heuristic has two important differences to the first one: (a) it is based on dependencies to a specific target class (instead to an entire module); and (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 component cp that depends on $TargetClass$; and (b) a common evolution pattern among the classes in cp is to remove dependencies to $TargetClass$, as observed in the history of classes C_1, C_4, and C_5.

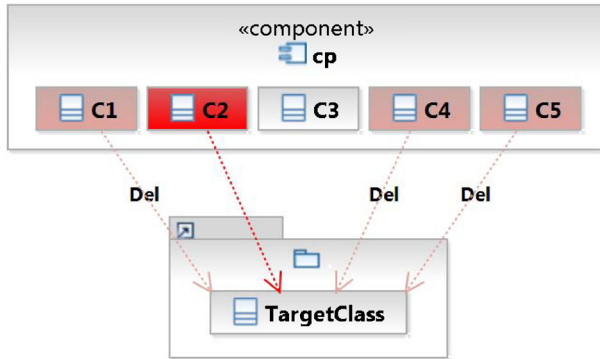[1]In a *depends* predicate, the pattern _ (underscore) matches any value.

Fig. 4 Example of divergence (C_2 depends on $TargetClass$). The label Del denotes a dependency removed in a previous version of the class

This heuristic aims to detect two possible sources of divergences: (a) the use of frameworks that are not authorized by the planned architecture (e.g., a system that occasionally relies on SQL statements instead of using the object-relational mapping framework prescribed by the architecture) (Terra and Valente 2009); and (b) the use of incorrect abstractions provided by an authorized framework (e.g., a system that occasionally relies on inheritance instead of annotations when accessing a framework that provides both forms of reuse, although the architecture authorizes only the latter).

Definition This heuristic relies on the *Dependency Deletion Rate*, as defined by the previous heuristic. However, it counts deletions regarding a target class c and a dependency type t—and not an entire module m. Thereupon, the heuristic is formalized as follows:

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

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

2.2.3 Heuristic #3

This heuristic is based on the assumption that a common type of divergences is the creation of asymmetrical cycles between components. More specifically, as illustrated in Fig. 5, this heuristic aims to identify pairs of components cp_1 and cp_2 where most references are from cp_2 to cp_1, but there are also few references in the reverse direction. The assumption is that the components were originally designed to communicate unidirectionally and the dependencies in the "wrong" direction are highly likely to represent architectural violations (and might not be exceptions authorized by the architecture, e.g., for performance issues).
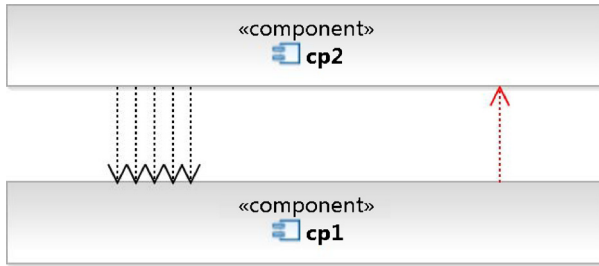
Fig. 5 Divergences due to asymmetrical cycles

This heuristic is particularly useful to detect *back-call* violations, a typical violation in layered architectures that occurs when a lower layer relies on services implemented by upper layers (Sarkar et al. 2009).

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

$$Div_3(cp_1) = \{ (x, c) \mid comp(x) = cp_1 \wedge comp(c) = cp_2 \wedge cp_1 \neq cp_2 \wedge$$
$$depends(x, c, \_, H) \wedge$$
$$D_{dir} \leqslant DepDirWeight(cp_1, cp_2) < 0.5 \}$$

Basically, divergences are pairs of classes (x, c) where x is a class in component cp_1 (i.e., the component under analysis) that have a dependency with a class c in component cp_2 and the dependencies from cp_1 to cp_2 satisfy the following conditions: (a) they are not exceptions, since they occur in a number that is greater than the minimal threshold D_{dir}; (b) but they are also not dominant, since there are more dependencies in the reverse direction, as specified by the *Dependency Direction Weight* lower than 0.5.

2.3 Ranking Strategy

The proposed heuristics generate warnings for architectural absences and divergences. However, by their nature, they are subjected to false positives. For this reason, it is important to report the warnings sorted by their potential to denote true violations. As usual in the case of heuristic-based results, the first presented warnings should ideally denote real violations to increase the confidence of the architects in the heuristics.
To rank the warnings generated by our approach, the natural strategy is to rely on the scattering and change (insertion or deletion) rates of the dependencies that characterize an absence or divergence. For example, in the cases of absences, we should first present the dependencies that are observed frequently in a component (i.e., have a very high *Dependency Scattering Rate*) and that are also introduced frequently (i.e., have a very high *Dependency Insertion Rate*). More specifically, the rank score of a given warning denoting

an absence (x, c, t)— where x is a class that is missing a dependency of type t with a target class c—is defined as:

$$ScoreAbsence(x, c, t) = \frac{DepScaRate(c, t, cp) + DepInsRate(c, t, cp)}{2}$$

where $cp = comp(x)$. This score represents the arithmetic mean of the scattering rate and the insertion rate of the dependency that characterizes the absence. The warnings denoting absences must be presented according to their respective scores, the ones with the highest score values first.

Additionally, the ranking scores of the warnings detected by heuristics #1 and #2 for divergences are defined as follows, respectively:

$$ScoreDiv_1(x, m) = \frac{(1 - DepScaRate(m, cp)) + DepDelRate(m, cp)}{2}$$

$$ScoreDiv_2(x, c, t) = \frac{(1 - DepScaRate(c, t, cp)) + DepDelRate(c, t, cp)}{2}$$

where $cp = comp(x)$. In the first score, the pair (x, m) is used to express that a class x is incorrectly establishing a dependency with a class in module m. Analogously, in the second score, the tuple (x, c, t) is used to express that a class x is incorrectly establishing a dependency of type t with a target class c. In both cases, we assume that high-ranked divergences should have a low scattering rate and a high deletion rate.

Finally, divergences detected by heuristic #3 are ranked according to the *Dependency Direction Weight* between the components in a cycle, as follows:

$$ScoreDiv_3(cp_1, cp_2) = DepDirWeight(cp_1, cp_2)$$

where the divergences in this case denote a dependency between classes in components cp_1 and cp_2 and they represent the "wrong" direction of the interaction between these components. For example, consider two cycles, where the first cycle has 18 % of the dependencies and the second one has 15 % of the dependencies in the *wrong* direction. In this case, the dependencies responsible for the "wrong" interaction of the second cycle should be ranked before the dependencies in the first one.

3 Tool Support

We implemented a prototype tool, called ArchLint, that supports the four heuristics for detecting architectural violations. As presented in Fig. 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 to svn repositories.

- The *Dependency Extractor* module is responsible for creating a model describing the dependencies available in each version considered in the evaluation. Essentially, this model is a directed graph, whose nodes are classes and the edges are dependencies. To extract the dependencies from source code, we rely on VerveineJ,[2] a Java parser that exports dependency relations in the format for modeling static information assumed by the Moose platform for software analysis (Nierstrasz et al. 2005; Ducasse et al. 2011).
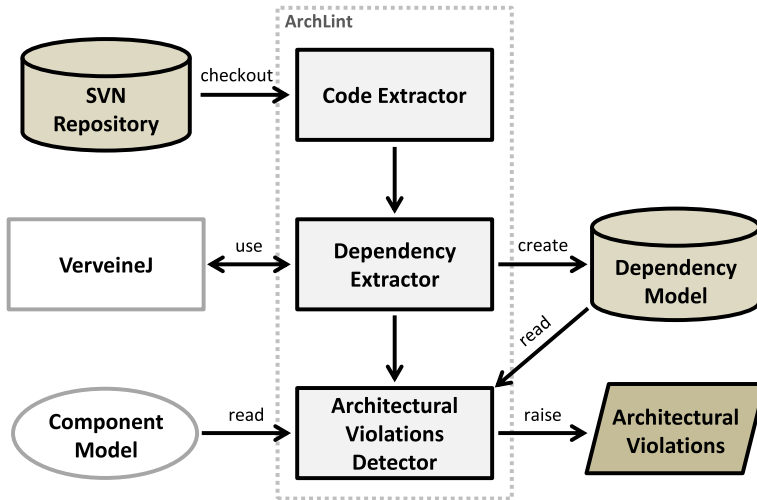
Fig. 6 ArchLint architecture

Nevertheless, we modified VerveineJ to store this information in a relational database to facilitate queries over the collected data.

- The *Architectural Violations Detector* module implements the heuristics described in Section 2. Basically, the heuristics are implemented as SQL queries. Additionally, this module ranks the architectural violations evidences—as described in Section 2.3—and reports them to the architect of the system under analysis.

4 A Heuristic-Based Architecture Conformance Process

In this section, we describe a process for architecture conformance, based on the proposed heuristics, as implemented by the ArchLint tool. Basically, this process addresses two central challenges regarding the practical use of our heuristics:

- The heuristics rely on thresholds to classify a dependency as a rare event in the space (scattering thresholds) and in time (insertion and deletion thresholds). Therefore, the thresholds must be defined before using a tool like ArchLint. Moreover, based on our initial experiments with the proposed heuristics (Maffort et al. 2013), we figured out that it is not possible to rely on universal thresholds, which could be reused for any system. This is the case especially of the insertion and deletion thresholds, since they depend on how often the architectural violations are detected and fixed, which vary from system to system.
- By their own nature, the proposed heuristics may lead to false positive warnings. For this reason, it is important to avoid the generation of a massive number of warnings, possibly with many false positives. Moreover, when presenting the architectural warnings to developers or architects, it is important to present the true warnings before the false ones, following the ranking strategies defined in Section 2.3.

---

[2]https://gforge.inria.fr/projects/verveinej.

To tackle the aforementioned challenges, we advocate that an architecture conformance process based on the proposed heuristics should follow an iterative approach. More specifically, we propose that a tool like ArchLint must be executed several times, starting with rigid thresholds. After each execution, the new warnings, i.e., the warnings not raised by the previous iterations, should be evaluated by the architect, in order to check whether they really denote true architectural violations. As a practical consequence of this evaluation step, the architect can for example request a change in the system to fix the detected violations. The architect may also decide to perform another iteration of the conformance process, with flexible thresholds. This process stops when a relevant number of violations is detected, e.g., a number of violations that is possible and worth to fix by the maintenance team in a given time frame. Moreover, it is also possible that he/she decides to finish the conformance process when most of the warnings raised after an iteration are false positives—and hence it is not worth to experiment with new thresholds.

Figure 7 defines the key steps of the proposed iterative conformance process. Basically, the process consists of a main loop where a given heuristic is applied (Step 2) and the old warnings, i.e., the warnings already detected in a previous iteration, are discarded (Step 3). After that, if very few warnings remain as the result of the iteration (Step 4), a new iteration is automatically started with more flexible thresholds (Step 5). The rationale is that it is better to trigger a new execution immediately than to evaluate few warnings that will be raised anyway by the next iteration. However, in case of enough warnings, they are first ranked—as described in Section 2.3—and then presented to the architect for analysis and classification as true or false warnings (Steps 6 and 7). After that, if the architect evaluates that it is worthwhile to continue searching for new warnings, considering the current workload of the maintenance team and the precision achieved by the current iteration, the thresholds are adjusted (Step 5) and a new iteration is started.

It is worth noting that the proposed conformance process is not a fully automatic procedure, as expected in the case of architecture conformance. Particularly, the final word on when the process should stop depends on the architect's judgment, based on his evaluation on whether it is relevant to fix the already detected violations and whether smaller precision rates can be tolerated. Moreover, the process depends on a constant that defines the minimal number of warnings that are worthwhile to evaluate in a given iteration (constant MIN_RESULTS).

Finally, the process depends on the initial threshold values used by each heuristic and on a procedure to adjust such thresholds before a new iteration, in order to make them less rigid. Figure 8 presents the proposed initial threshold values and the thresholds adjustment procedure, for each heuristic. Basically, the initial values represent very rigid thresholds. For example, for absences, we are recommending to start with a scattering rate of 95 % and an insertion rate of 95 %. Regarding the adjustment procedure, initially the insertion threshold is decremented in intervals of 5 %, starting at 95 % and finishing at 35 %. When this lower bound is reached, the scattering rate is decremented by 5 % and the insertion rate is reset to 95 %.

5 First Study: Proprietary Systems

To start evaluating our approach, we conducted a first study using two real-world information systems: SGA and M2M (we omit the real names for confidentiality reasons). Our central goal is to perform experiments with the iterative conformance process described in
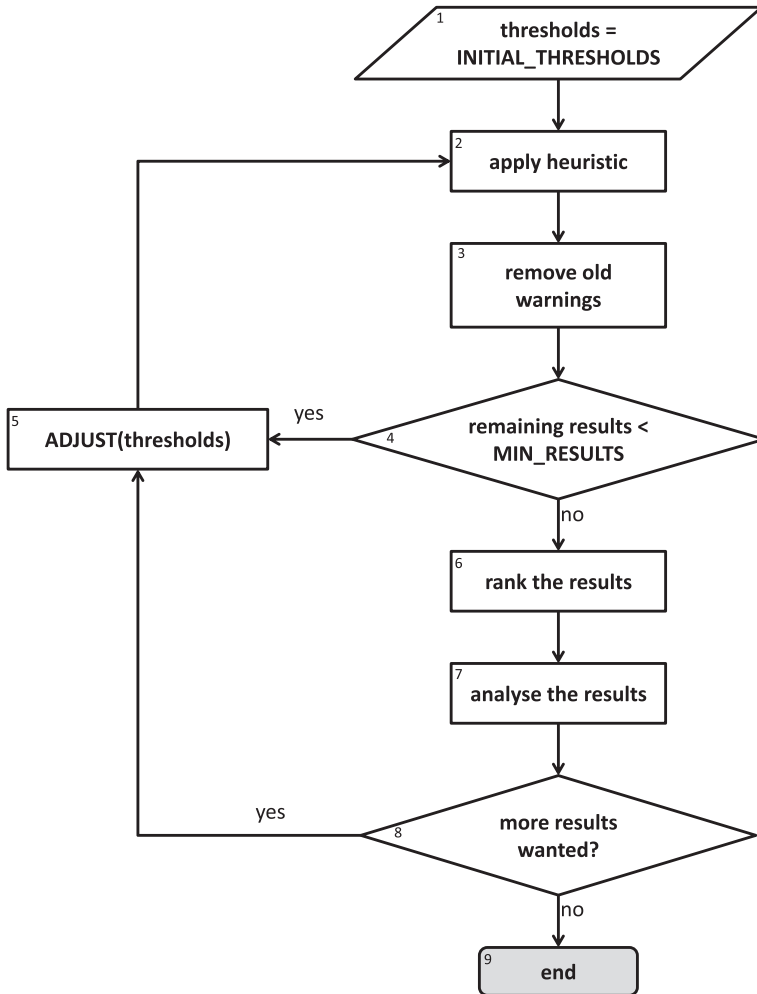
Fig. 7 Architecture conformance using the proposed heuristics

Section 4. Thereupon, we report the number of iterations required by the process, the precision achieved after each iteration, and the effectiveness of the strategy proposed to rank the warnings raised by a given heuristic.

This first study is organized as follows. Sections 5.1 and 5.2 present the methodology and the results of using the conformance process on the SGA system. Likewise, Sections 5.3 and 5.4 present the methodology and the results on the M2M system. Last, Section 5.5 enumerates threats to validity.

5.1 Methodology for the SGA System

We followed the architecture conformance process defined in Section 4 to detect violations in the architecture of an EJB-based information system used by a major Brazilian university,

```
    // INITIAL_THRESHOLDS:
1   Asca = 0.95;
2   Ains = 0.95;
    // ADJUST():
3   if Ains > 0.35 then
4       Ains = Ains - 0.05;
5   else
6       Asca = Asca - 0.05;
7       Ains = 0.95;
8   end
```

(a) Heuristic for Absences thresholds

```
    // INITIAL_THRESHOLDS:
1   Dsca = 0.05;
2   Ddel = 0.95;
    // ADJUST():
3   if Ddel > 0.35 then
4       Ddel = Ddel - 0.05;
5   else
6       Dsca = Dsca + 0.05;
7       Ddel = 0.95;
8   end
```

(b) Heuristics #1 and #2 thresholds (for divergences)

```
    // INITIAL_THRESHOLDS:
1   Ddir = 0.45;
    // ADJUST():
2   if Ddir > 0.0 then
3       Ddir = Ddir - 0.05;
4   end
```

(c) Heuristic #3 threshold (for divergences)

Fig. 8 Initial thresholds values and thresholds adjustment procedures for each heuristic
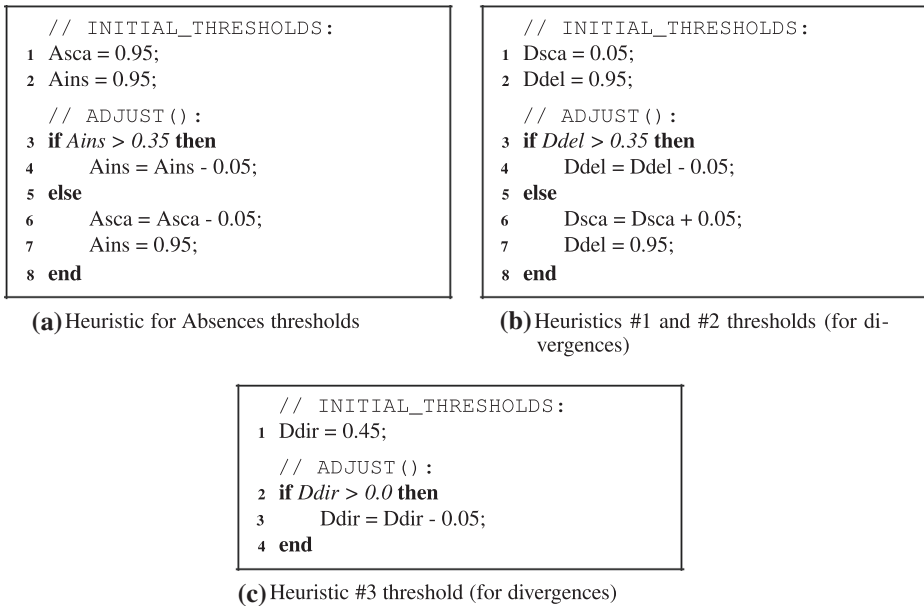
which for confidentiality reasons we will just call SGA. The system includes functionalities for human resource management, finance and accounting management, and material management, among others. In this system, we considered 7,692 revisions (all available revisions), stored in a `svn` repository, from March, 2009 to June, 2013. After parsing these revisions, ArchLint —our prototype tool— generated a dependency model with more than 147 million relations, requiring 68 GB of storage in a relational database. The generation of this database took 72 hours and 27 minutes (in a six-core Intel Xeon 2.20 GHz server, with 64 GB RAM, running Ubuntu 12.04 and Java version 1.7). Regarding this total time, 11 hours and 57 minutes were used by VerveineJ to parse the extracted versions.

All extracted versions were considered for computing the functions *DepInsRate* and *DepDelRate*, described in Sections 2.1 and 2.2. The last considered revision has 1,864 classes and interfaces, organized in 100 packages, comprising around 273 KLOC.

We initially asked SGA's senior architect to define the system's 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 process automated by the system.
- `IPersistence`: facade for the persistence layer.
- `PersistenceLayer`: implementation of persistence.
- `BusinessEntity`: domain types (e.g., Professor, Student, etc.).

Table 2 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 15 packages and 286 classes (`ManagedBean`) to components with 17 packages and 330 classes (`BusinessEntity`). The table also shows

Table 2 High-level components in the SGA system

Component	Packages	Classes	Regular Expression
ManagedBean	15	286	br.sga*.managedbeans*
IService	17	312	br.sga*.ejb.facade*
ServiceLayer	17	312	br.sga*.ejb.local*
IPersistence	17	313	br.sga*.dao* <excludes> br.sga*.dao.jpa*
PersistenceLayer	17	311	br.sga*.dao.jpa*
BusinessEntity	17	330	br.sga*.domain*

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 with common names or prefixes.

Using as input the regular expressions specifying the high-level SGA components, we executed ArchLint multiple times, as prescribed by the conformance process described in Section 4. Particularly, for each heuristic, we considered the initial thresholds and the thresholds adjustment procedure suggested in Fig. 8. Moreover, SGA's architect was only requested to evaluate the warnings generated by iterations that produces at least 10 new warnings (constant MIN_RESULTS). When this happened, we asked the architect to carefully examine the new warnings 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 evaluation. We did not measure recall because it would require finding the whole set of missing or undesirable dependencies, which in practice requires a detailed and complete inspection of the source code, which is certainly a hard task considering the size of the SGA system.

To evaluate the strategy used to rank the warnings generated by a given iteration, we relied on a *discounting cumulative function*, often used to evaluate web search engines and other information retrieval systems (Baeza-Yates and Ribeiro-Neto 2011). This function progressively reduces the value of a document—a warning, in our case—as its position in the rank increases. Basically, the value of a warning is divided by the log of its rank position, as follows:

$$DCG = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i)}$$

where p is the number of warnings generated by the heurist ic and *rel* is the relevance of a warning. In our particular case, this relevance is a binary value: true positive warnings have relevance value equal to 1; false positive warnings have a relevance value of zero.

More specifically, we report the effectiveness of the ranking strategy using a *normalized discounted cumulative gain* (nDCG) function, as follows:

$$nDCG = \frac{DCG}{IDCG}$$

where *IDCG* is the best possible value for the *DCG* function, i.e., the value generated by a perfect ranking strategy, considering a given list of warnings. Therefore, *nDCG* values range from 0.0 to 1.0, where 1.0 is the value produced by a perfect ranking algorithm.

Table 3 Detecting absences in the SGA system

Iteration	A_{sca}; A_{ins}	Warnings			Precision			nDCG
		Iter.	New	Eval.	Iter.	Overall		
1	0.95; 0.80	3	3	—	—	—		—
2	0.95; 0.55	26	23	26	100.0 %	100.0 %		1.00
3	0.95; 0.40	42	16	16	87.5 %	95.2 %		0.94
4	0.95; 0.35	46	4	—	—	—		—
5	0.90; 0.55	52	26	30	83.3 %	90.3 %		0.99
6	0.90; 0.50	73	21	21	95.2 %	91.4 %		0.98
7	0.85; 0.50	108	35	35	74.3 %	86.7 %		0.90

5.2 Results for the SGA System

This section presents the results achieved after following the proposed conformance process to detect absences (Section 5.2.1) and divergences (Sections 5.2.2, 5.2.3, and 5.2.4) in the architecture of the SGA system. Additionally, Section 5.2.5 summarizes the precision achieved by our approach for divergences. Next, Section 5.2.6 compares our results with reflexion models (RM). Finally, Section 5.2.7 evaluates how the proposed heuristics perform in different stages of the evolution of the SGA system.

5.2.1 Results for Absences

Table 3 presents the results achieved by each iteration of the conformance process, when it was used to provide warnings for absences. For each iteration, the table presents the following data: (a) the thresholds required by the heuristic for detecting absences; (b) the number of warnings produced in the iteration, including the number of new warnings and the number of warnings evaluated by the architect, if any; (c) the precision achieved by the current iteration and the overall precision until this execution, i.e., considering the warnings evaluated in the current iteration and also in previous iterations. Precision is defined as usual, by dividing the number of true warnings by the total number of warnings. For the sake of clarity, we do not show data on thresholds that did not produce warnings or that produced exactly the same warnings as previous iterations. For example, the first execution was performed with $A_{sca} = 0.95$ and $A_{ins} = 0.95$. These thresholds did not generate warnings and therefore are not presented in Table 3. The same happened with the next two tested thresholds, i.e., (0.95; 0.90) and (0.95; 0.85). The first selection to generate warnings was (0.95; 0.80), which generated three (new) warnings. However, since we configured the process to just require the architect's evaluation when a minimal of ten new warnings is generated by an iteration, these initial warnings were not presented to the architect. In the second iteration, 26 warnings were produced in total. From these warnings, 23 warnings are new and three warnings correspond exactly to the warnings generated by the first iteration. Therefore, the 26 warnings were showed and discussed with the architect, for classification as true or false positives. In this case, a precision of 100 % was achieved.

As can be observed in Table 3, we decided to stop the process after seven iterations, including iterations #1 and #4 that did not generate enough warnings for the evaluation. In

Table 4 Detecting divergences in the SGA system using Heuristic #1

Iteration	D_{sca}; D_{del}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.05; 0.85	1	1	—	—	—	—
2	0.05; 0.75	4	3	—	—	—	—
3	0.05; 0.50	5	1	—	—	—	—
4	0.10; 0.60	10	6	11	100 %	100 %	1.00
5	0.10; 0.30	92	81	81	100 %	100 %	1.00

the remaining five iterations, the architect evaluated 128 warnings, with an overall precision of 86.7 %. In Table 3, we can also observe a downward tendency in the precision after each iteration. For example, in iteration #2 we achieved a precision of 100 % and in the last iteration the precision was 74.3 %. Finally, by evaluating the *nDCG* results, we can conclude that the criteria used to rank the warnings generated by a given iteration was quite effective. As in the case of the precision, the *nDCG* values in Table 3 present a tendency to decrease after each iteration. For example, in the last iteration the ranking strategy achieved 90 % of the effectiveness of a perfect ranking algorithm.

We finished after seven iterations because the architect considered that the true warnings detected by such iterations should be first addressed by the maintenance team before continuing with the conformance process.

Example 1 As an example of a true warning (detected in iteration #1), we can mention the following:[3]

Component:	`IService`
Class:	`br.sga.doc.ejb.facade.DictionaryService`
Missing Dependency :	`javax.ejb.RemoteClassAnnotation`
DepScaRate; *DepInsRate* :	0.990; 0.800

In the SGA system, the architect explained that interfaces in the `IService` component must receive a `Remote` annotation, which is an EJB annotation used to mark a remote business interface for a session bean. In fact, 99 % of the interfaces in `IService` have this annotation (*DepScaRate*). Moreover, 80 % of the interfaces originally created without this annotation where later maintained to include the annotation (*DepInsRate*). The lack of this annotation does not have an impact on the behavior of the system in its current version because the classes implementing the interfaces missing the annotation are used only by local clients. However, according to their specification, they should also support remote accesses.

---

[3]To improve the paper's comprehension, we translated the class names from Portuguese to English.

Example 2 As an example of a false warning, we can mention the following (detected in iteration #7):

Component:	`BusinessEntity`
Class:	`br.sga.core.domain.FederatedUnit`
Missing Dependency:	`br.sga.core.domain.AuditInfoInheritance`
DepScaRate; *DepInsRate* :	0.885; 0.524

The SGA system has an internal audit service, used to log changes in classes storing highly sensitive data, such as personal info. The classes subjected to this service must inherit from a special class, called `AuditInfo`. Particularly, in the `BusinessEntity` component, 88.5 % of the classes use this service (*DepScaRate*). Moreover, more than half of the classes in `BusinessEntity` were changed after their initial creation to inherit from `AuditInfo` (*DepInsRate*) because the audit service was introduced later in the system. For this reason, the heuristic incorrectly inferred that all classes in `BusinessEntity` must inherit from `AuditInfo`. However, there are classes that by their own nature do not need this service, such as `FederatedUnit`, which is a class that stores information about the Brazilian States (i.e., data that rarely changes and therefore does not need an audit service, according to SGA's architect).

5.2.2 Results for Divergences - Heuristic #1

Table 4 shows the results achieved after each iteration of the conformance process, when configured to provide warnings using the first heuristic for divergences. As can be observed, we performed five iterations, but only in the last two the evaluation of the architect was required. We asked the architect to evaluate 92 warnings, with a precision of 100 %. We finish the process because the architect considered this number of true divergences worth to be handled, before continuing to search for new warnings.

Example 3 As an example of a true warning (detected in iteration #2), we can mention the following:

Component:	`PersistenceLayer`
Class:	`br.sga.core.dao.jpa.PRSystDAO`
Unauthorized dependency:	`br.sga.ejb.facade.PersonFacade`
DepScaRate; *DepDelRate*:	0.012; 0.750

In this case, a `DAO` class in the `PersistenceLayer` has a dependency with a class in the SGA's facade, which is not allowed by the architecture. In fact, less than 1.5 % of the `DAO`s establish a dependency with `IService` classes (*DepScaRate*). Moreover, in the past, 75 % of the classes that established a dependency like that in a given version were later changed to remove the dependency (*DepDelRate*). Finally, package `br.sga.ejb.facade` has a well-defined heavy-user in the system, which is the `ManagedBean` component. In fact, 73.4 % of the dependencies to this package are established by classes located in `ManagedBean`. Therefore, these evidences when combined are responsible for this true divergence. In fact, the architect commented that this divergence represents a back-call because a lower layer (`PersistenceLayer`) is using a service from an upper module (`br.sga.ejb.facade`).

5.2.3 Results for Divergences - Heuristic #2

Table 5 shows the results achieved by the second heuristic for divergences. In six out of nine iterations, the evaluation of the architect was required. In total, we asked the architect to evaluate 325 warnings, with an overall precision of 34.2 %, which corresponds to the lowest precision in the conformance process. We finish the process because the architect considered this precision too low, specially the precision of the last iteration, which was 20.3 %. In summary, after nine iterations, the architect considered the process no longer productive, demanding the evaluation of many false positives per true warning discovered.

Despite the lower precision, by analyzing the *nDCG* values in Table 5, it is possible to observe that the strategy to rank the warnings generated by the iterations was partially effective. In the last five iterations, for example, we achieved an average precision of 40.7 % with the *nDCG* values ranging from 0.44 to 0.92, with an average value of 0.68. In other words, the lower precision was compensated by a tendency to present the true warnings in the top ranked results.

Example 4 As an example of a false warning (detected in iteration #1), we can mention the following:

Component:	`ManagedBean`
Class:	`br.sga.web.managedbeans.MBEducLevel`
Unauthorized dependency:	`br.sga.ejb.facade.EducLevelFacade`
	`AttributeClass`
DepScaRate; *DepDelRate*:	0.003; 0.888

This particular false warning is due to two facts. First, among the 286 classes in `ManagedBean`, only a single class references a particular class in the SGA's facade, called `br.sga.ejb.facade.EducLevelFacade` (*DepScaRate* = 0.003). Second, in the past, a common refactoring in SGA was removing the dependencies to this class coming from `ManagedBean`. In fact, 88.8 % of the classes that once had this dependency were later changed to remove it (*DepDelRate*). Despite these two evidences, the warning in this case is false, according to the architect. He explained that `EducLevelFacade` is a specific class in the system, responsible for very specific scholar degrees. However, in the past

Table 5 Detecting divergences in the SGA system using Heuristic #2

Iteration	D_{sca}; D_{del}	Warnings			Precision		
		Iter.	New	Eval.	Iter.	Overall	nDCG
1	0.05; 0.85	5	5	—	—	—	—
2	0.05; 0.80	12	7	—	—	—	—
3	0.05; 0.70	25	13	25	60.0 %	60.0 %	0.75
4	0.05; 0.65	27	2	—	—	—	—
5	0.05; 0.60	58	31	33	27.3 %	41.4 %	0.71
6	0.05; 0.55	88	30	30	60.0 %	47.7 %	0.76
7	0.05; 0.50	136	48	48	29.2 %	41.2 %	0.44
8	0.05; 0.45	172	36	36	66.7 %	46.5 %	0.92
9	0.05; 0.40	325	153	153	20.3 %	34.2 %	0.51

this class was also responsible for regular scholar degrees and at a certain point in the system's evolution a design change was made towards creating a new class to represent such degrees. Despite that, `EducLevelFacade` remained in the system, but it is used only for very specific degrees. In summary, the changes in the system responsible for the high *Dependency Deletion Rate* were motivated by a design decision not related to removing architectural violations.

5.2.4 Results for Divergences - Heuristic #3

Table 6 shows the results achieved by the second heuristic for divergences. In this case, as defined in Fig. 8, we started searching for cycles where 45 % of the dependencies are in one direction and 55 % are in the reverse one, i.e., $D_{dir} = 0.45$. We found no pair of components attending this precondition. The same happened when we reduced D_{dir} until 0.20. However, when we defined $D_{dir} = 0.15$, 75 warnings were generated for the first time and they were all ranked as true positives. Finally, in the next three iterations, no new warning was produced.

Example 5 By analyzing the results with SGA's architect, we discovered that all 75 warnings are between the components `PersistenceLayer` and `ServiceLayer`. Specifically, there are 320 dependencies from `ServiceLayer` to `PersistenceLayer` and 75 (unauthorized) dependencies in the reverse direction, which represents a *DepDirWeight* equal to 0.189 (75/(320 + 75)). For this reason, the warnings were only produced when we tested a minimal threshold of 15 % to classify dependencies in the "wrong direction" as divergences. Moreover, exactly the same warnings were generated again when this threshold was reduced until zero.

5.2.5 Overall Results for Divergences

Table 7 presents the precision achieved by our approach for divergences, considering the warnings evaluated for the three heuristics. As can be observed, both heuristics #1 and #3 achieved 100 % of precision, and heuristic #2 achieved a precision of 34.2 %. Considering the results of all heuristics, we generated 278 true divergences and 214 false warnings in nine iterations, with an overall precision of 56.5 %.

5.2.6 Comparison with Reflexion Models

This section compares our results with reflexion models (RM) (Murphy et al. 2001; Murphy et al. 1995), which is a well-known and lightweight approach for architecture conformance.

Table 6 Detecting divergences in the SGA system using Heuristic #3

Iteration	D_{dir}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.15	75	75	75	100 %	100 %	1.00
2	0.10	75	0	—	—	—	—
3	0.05	75	0	—	—	—	—
4	0.00	75	0	—	—	—	—

Table 7 Precision considering the warnings evaluated for three heuristics for divergences

	Heuristic #1	Heuristic #2	Heuristic #3	Total
Iterations	2	6	1	9
Warnings	92	325	75	492
True Positives	92	111	75	278
False Positives	0	214	0	214
Precision	100 %	34.2 %	100 %	56.5 %

To make this comparison, we calculated a reflexion model for the SGA system, reusing the high-level model used as input by our approach. As illustrated in Fig. 9, we had to enrich our initial model in two directions. First, we defined six extra components, to denote external components used by the SGA implementation, including frameworks for presentation (Java Server Faces), for communication (Servlets), and for persistence (Java Persistence API and SQL). Second, we included 25 relations (edges) between the defined components. On the other hand, when using our approach, external frameworks and relations between components are automatically inferred by the considered heuristics. Using the enriched high-level model, we calculated a reflexion model, i.e., a model that highlights divergences.

Figure 10a compares the results for divergences achieved by RM and by our approach. As mentioned in Section 5.2, the proposed heuristics detected 254 true and unique warnings in the SGA system. On the other hand, RM was able to detect 75 divergences. For example, RM missed 57 divergences between ManagedBean and JavaIO, two divergences between IService and EJB, and 26 divergences between BusinessEntity and JPA. In fact, ManagedBean has a dependency with JavaIO, but with the wrong class in this component. Specifically, an architectural rule states that ManagedBean can only establish
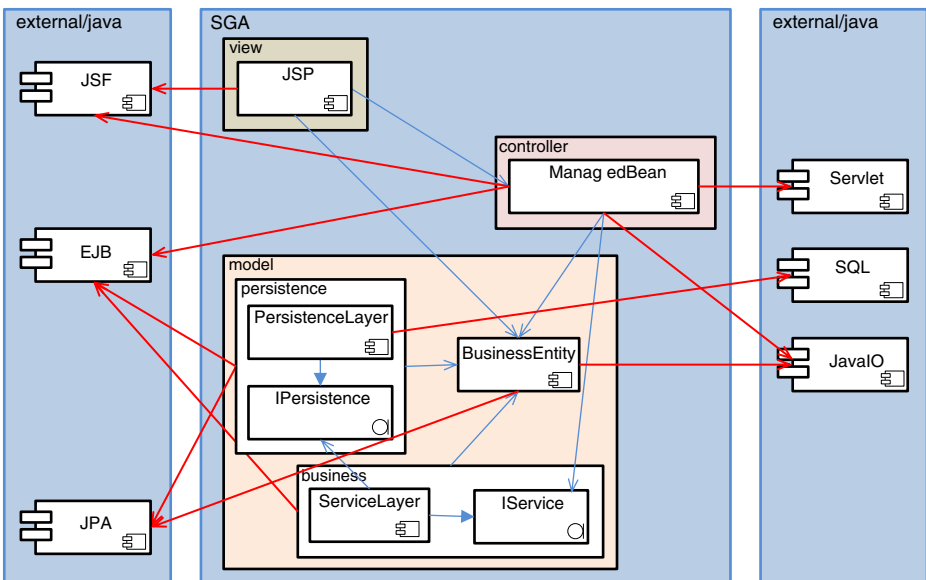

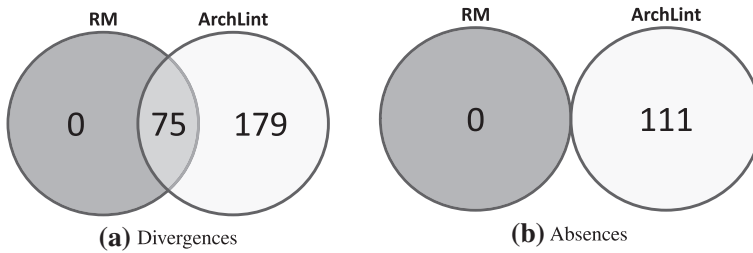**Fig. 9** Enriched high-level model for the SGA system

Fig. 10 Absences and divergences detected by RM and the proposed heuristics

dependencies with a single class in `JavaIO`, called `IOException`. Despite this, there are 57 dependencies with other `JavaIO` classes, such as `BufferedReader` and `File`. To detect these divergences, the high-level model used by the RM technique must be further refined, by creating two nested components in `JavaIO`, one component with only the `IOException` class and another one with `File`, `FileReader`, `BufferedReader`, `FileOutputStream`, and `OutputStream`. After this modification, we must update the dependency from `ManagedBean` to reach just the `IOException` subcomponent. In fact, this need to refine reflexion models motivated the extension of the original proposal with hierarchical modules (Koschke and Simon 2003).

Figure 10b compares the results for absences achieved by RM and by our approach. As reported in Section 5.2, the proposed heuristics detected 111 true absences in the SGA system. On the other hand, RM missed all of them. To explain the reason for this massive failure in detecting absences, we will consider the components `PersistenceLayer` and `JPA`. As illustrated in Fig. 9, the high-level model prescribes that there must exist a dependency from `PersistenceLayer` to `JPA`. However, `PersistenceLayer` is a coarse-grained component—with 311 classes. For this reason, a single class that relies on `JPA` is sufficient to hide all eventual absences in the remaining classes of the component. Of course, it is possible to refine the high-level model by creating a nested component in `PersistenceLayer` with exactly the classes that must depend on `JPA` and to establish an edge between each of such classes and `JPA`. However, the proliferation of nested components increases complexity and contrasts with the lightweight profile normally associated with RM-based techniques.

Finally, it is important to state that RM is a precise technique, assuming the relations defined by the architect reflect the idealized architecture. Therefore, the technique does not generate false warnings. On the other hand, for the 278 true divergence warnings raised by the proposed heuristics, there were also 214 false warnings (precision equals 56.5 %).

5.2.7 Historical Analysis

In this section, we evaluate how the proposed heuristics perform in different stages of the evolution of the SGA system. More specifically, we performed again the heuristics that depend on historical information, i.e., heuristic for absence and heuristics #1 and #2 for divergences, but considering a limited number of versions. In each execution, we discarded the versions of the first, second, third, and fourth years, respectively. Moreover, we reused the same thresholds from the first iteration of the process followed by the SGA architect when validating the results using the complete dataset. For example, when computing the heuristic for absence, we considered $A_{sca} = 0.95$ and $A_{ins} = 0.55$, which are exactly

Table 8 Historical analysis results

	Full dataset	Dataset discarding			
		1st yr	2nd yr	3rd yr	4th yr
Absences	26	3	3	3	0
Divergence - Heuristic #1	11	11	7	7	0
Divergence - Heuristic #2	15	2	2	2	0

the first thresholds evaluated by the architect in the original study (see Table 3). We then checked whether each violation detected using the complete dataset is also detected when the first n initial years are discarded ($1 \leq n \leq 4$).

Table 8 reports the true warnings detected in each time frame. Considering the complete dataset, the heuristic for absences detected 26 violations, and the heuristics #1 and #2 for divergences detected 11 and 15 violations, respectively. When we discard the first-year versions, there is a major reduction in the number of absences (from 26 violations to three violations) and in the number of divergences detected by heuristic #2 (from 15 violations to two violations). On the other hand, the number of violations detected by heuristic #1 remains exactly the same when considering the full dataset (11 violations).

To explain these results, we first characterize the changes that have an impact in the proposed heuristics. The heuristic for absences monitors a change that inserts a missing dependency in the target class, which we will refer to *Insert Missing Dependency* change. In the case of divergences, the heuristics monitor a change that removes an undesirable dependency from a target class, which we will refer to *Remove Undesirable Dependency* change. Figure 11 reports the distribution of these changes in our dataset, in four years. We can observe that both changes happened most of the times in the first year of SGA's evolution. For example, 53 % of the *Insert Missing Dependency* changes were performed in
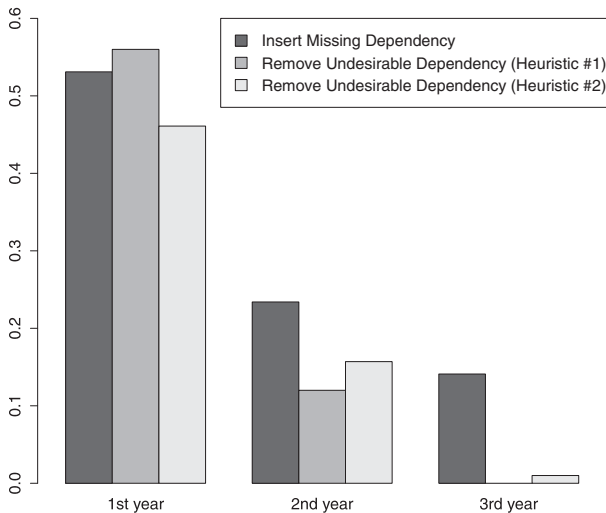

Fig. 11 Distribution of the change operations by year (for each operation, the bars show the percentage of changes performed in each year considered in the SGA conformance process)

the first year. Regarding the *Remove Undesirable Dependency*, we have that 56 % (for the ones associated to heuristic #1) and 46 % (for the ones associated to heuristic #2) happened in the first year. Therefore, when we removed the commits collected in the first year, we also removed most of the changes responsible for triggering the warnings of architectural violations, as considered by the three heuristics that depend on historical data. In the case of the heuristic for absence and the heuristic #2 for divergences, the changes performed in the remaining years were not sufficient to attend the respective thresholds ($D_{sca} = 0.05$ and $D_{del} = 0.70$), which are very rigid. On the other hand, in the case of the heuristic #1 for divergences, they were still sufficient to trigger the same 11 violations when using the full dataset. The central reason in this case is the fact that the computation of this heuristic uses more flexible thresholds ($D_{sca} = 0.10$ and $D_{del} = 0.60$). Finally, in all cases, after removing four years of revisions, we were not able to detect violations anymore.

Clearly, it is not possible to generalize the results of this subsection to other systems. However, in the specific case of the SGA system, they show that most changes the proposed heuristics depend on happened in the first year of the system's evolution. Therefore, we can extrapolate that at this year the development team was not completely aware of SGA's planned architecture. For that reason, many violations were introduced but also fixed, as the architecture quickly became clearer to the initial team of developers. Finally, the results reported in this historical analysis reinforce the importance of the thresholds when computing the heuristics. For example, the heuristic #1 for divergences was not deeply impacted by removing the commits of the first year due to its evaluation with more flexible thresholds.

5.3 Methodology for the M2M System

M2M is an ERP management system designed for use by Brazilian government institutions. The system manages the administrative process of acquisition and distribution of products and services. The system also documents the entire process workflow and includes other features such as integration with governmental systems, reports, etc.

Table 9 High-level components in the M2M system

Component	# Classes	Regular Expression
PersistenceLayer	173	br.m2m.*Impl
IPersistenceLayer	398	br.m2m.*.dao.*DAO <excludes> br.m2m.*Impl
BusinessEntity	1,143	br.m2m.*DTO <or> br.m2m.*.domain.*
ExceptionHandler	12	br.m2m.*Exception
Timer	58	br.m2m.*.timers.* <or> br.m2m.*.Timer*
Security	9	br.m2m.*.security.*
Action	1,056	br.m2m.*Action
Form	243	br.m2m.*Form
WEBController	1,048	br.m2m.*MBean <or> br.m2m.*.jsf.* <or> br.m2m.*Servlet <or> br.m2m.*.struts.*
Report	17	br.m2m.*.Rep* <or> br.m2m.*.Graphic*
IService	16	br.m2m*.interfaces.*
ServiceLayer	656	br.m2m.*.Processor* <or> br.m2m.*.business.*
Util	170	br.m2m.*Utils <or> br.m2m.*.util.*

Table 10 Precision considering the warnings raised in M2M system.

	Iterations	Warnings	Mean precision	Overall precision
Absences	2	112	81.5 %	82.1 %
Divergence - Heuristic #1	0	0	—	—
Divergence - Heuristic #2	3	119	41.7 %	18.5 %
Divergence - Heuristic #3	2	48	63.9 %	75.0 %
All Heuristics	7	279	62.4 %	53.8 %

We considered 61,785 revisions available in the system's control version repository (all available revisions), from November, 2010 to October, 2013. The last considered revision has 4,999 classes and interfaces, organized in 485 packages, comprising 610 KLOC. After parsing all revisions, the dependency model generated by our approach has 271.5 million relations and requires 107 GB of storage in a relational database.

Similarly to SGA, we asked M2M's architect to define the system's high-level component model. Table 9 presents the components suggested by the architect and the regular expressions that define the classes in each component, besides the respective number of classes. We can observe that the regular expressions in M2M map classes to components, and not packages to components as occur in the SGA system. The main reason is that classes associated to different components may be located in the same package. As an example, classes from components `PersistenceLayer` and `IPersistenceLayer` are located in the same package, called `br.m2m.arq.dao.contract`. Furthermore, the size of the proposed components ranges from nine classes (component `Security`) to 1,143 classes (component `BusinessEntity`).

The regular expressions in Table 9 were used as input to the heuristics. Each heuristic was executed several times and the architect was only requested to evaluate the warnings raised by the iterations that produced at least 10 new warnings. In this case, the architect carefully examined the warnings and classified them as true or false positives.

5.4 Results for the M2M System

Table 10 summarizes the precision achieved by the proposed heuristics in M2M. In short, we achieved an overall precision ranging from 18.5 % (heuristic #2 to detect divergences) to 82.1 % (the heuristic to detect absences). Nevertheless, heuristic #1 did not indicate any divergence in M2M. Considering the mean precision of the iterations, we achieved results ranging from 41.7 % to 81.5 %.[4]

Moreover, to discover the violations we executed seven iterations, raising 279 warnings with an overall precision of 53.8 %. Appendix B presents a detailed description of the warnings detected by each heuristic.

During the evaluation, the architect commented that the detected violations are, in fact, due to some relevant architectural constraints in M2M, as follows:

- *All* classes in `PersistenceLayer` *must depend* on class `org.hibernate.Query` (35 absences detected).

---

[4]*Mean precision* is the average precision of the iterations evaluated by the architect, whereas *Overall precision* is the total number of true warnings by the total number of warnings.

- *Only* classes in `IPersistenceLayer` *must depend* on class `org.hibernate.Session` (three divergences detected by heuristic #2).
- Classes in `ServiceLayer` *cannot depend* on class `java.net.UnknownHostException` as a `CaughtException` (four divergences detected by heuristic #2).
- Classes in `BusinessEntity` *cannot depend* on classes located in `PersistenceLayer` (four divergences detected by heuristic #3).
- *Only* classes in `WEBController` *can depend* on classes located in `WEBController` (18 divergences detected by heuristic #3).
- Classes in `PersistenceLayer` *cannot depend* on classes located in `ServiceLayer` (three divergences detected by heuristic #3).

Therefore, we argue that the proposed heuristics were able to detect violations of well-known architectural patterns and rules in the M2M system, without requiring their explicit formalization, as required by other architecture conformance approaches.

5.5 Threats to Validity

The threats of this study are the same in both systems. We relied on a single architect per system 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 senior architects, with a complete domain of SGA's and M2M's architecture and implementation. Furthermore, one can argue that these architects might be influenced to design a model favoring ArchLint. However, we never explained to the architects the heuristics followed by ArchLint to discover architectural violations.

6 Second Study: an Open-Source System

In this study, we report the application of the proposed heuristics in an open-source system named Lucene.

6.1 Study Setup

In this system, our evaluation is fully based on a Reflexion Model (RM) independently proposed by Bittencourt et al. (Bittencourt 2012). We reused the component specifications from the high-level model (HLM) defined as the input for the proposed heuristics. Table 11 lists the components defined by the Lucene's HLM.

Because the HLM was carefully designed for architecture conformance purposes, we considered the computed reflexion models as a reliable oracle for evaluating the precision of the heuristics. More specifically, we classify a warning as a true positive when it is also reported in the reflexion model. In other words, in this second study, we replaced the architect with a reflexion model. Moreover, we decided by ourselves when to stop the iterative process followed for each heuristic. Basically, we targeted around 100 warnings per heuristic, stopping when this value was reached.

In the case of absences, the reflexion model did not indicated absences in Lucene because in RM a single class in a component satisfying the prescribed architectural rule is sufficient to hide all absences in this component. For instance, the HLM prescribes that a dependency from *Search* to *Index* must exist. However, *Search* is a component with 351 classes

Table 11 High-level components in Lucene

Component	Regular Expression
QueryParser	org.apache.lucene.queryparser.*
Search	org.apache.lucene.search.*
Index	org.apache.lucene.index.*
Store	org.apache.lucene.store.*
Analysis	org.apache.lucene.analysis.* <or> org.apache.lucene.collation.*
Util	org.apache.lucene.util.* <or> org.apache.lucene.message.*
Document	org.apache.lucene.document.*

and therefore a single class from *Search* that relies on *Index* is sufficient to hide eventual absences in the remaining classes of the component.

To evaluate the heuristics, we checked out 1,959 revisions, from March, 2010 to July, 2012. The last revision considered in the study has 336 KLOC.

6.2 Results for the Lucene System

Table 12 reports the precision achieved by the heuristics for divergences. The overall precision was 59.2 %. In 16 iterations, our approach raised 446 warnings with a mean precision in the iterations used for each heuristic ranging from 7.0 % to 98.5 %. Appendix C presents a detailed description of the warnings detected by each heuristic.

An analysis of the divergences missed by our approach—i.e., divergences we missed but that were detected by the reflexion model—revealed that we missed many divergences with a high scattering and a low deletion rate. For example, the high-level model does not define a dependency between components `Search` and `Store`. However, 81 dependencies like that are presented in 32 % of the classes in `Store`, which exceed by a large margin the thresholds we tested. Moreover, only 6 % of such dependencies were removed along Lucene's evolution. Stated otherwise, in Lucene, it is common to observe divergences that are not spatially and historically confined in their source components. Therefore, we argue that Lucene's architecture might have evolved during the time frame considered in our study. As a result, many dependencies that were not authorized by the initial high-level model might have turned themselves into a frequent and enduring property of the system.

Table 12 Precision considering the warnings raised in Lucene system.

	Iterations	Warnings	Mean precision	Overall precision
Divergence - Heuristic #1	6	168	49.3 %	55.4 %
Divergence - Heuristic #2	4	114	7.0 %	7.9 %
Divergence - Heuristic #3	6	164	98.5 %	98.8 %
All Heuristics	16	446	51.6 %	59.2 %

6.3 Threats to Validity

It is possible that the Lucene's high-level model does not capture some (true) violations. However, we argue that the chances are reduced since the models were carefully designed and refined to create a benchmark for architecture conformance.

7 Discussion

In this section, we discuss the main lessons learned in the studies reported in Sections 5 and 6.

7.1 Are our Results Good Enough?

We detected a relevant number of architectural violations with the proposed heuristics: 389 violations in the SGA system; 150 violations in the M2M system; and 264 violations in Lucene.

Furthermore, we achieved the following overall precision rates: 53.8 % (M2M), 59.2 % (Lucene), and 62.7 % (SGA). These precision values are compatible to the ones normally achieved by static analysis tools, such as FindBugs (Hovemeyer and Pugh 2004). For example, in a previous study, we found that precision rates greater than 50 % are only possible by restricting the analysis to a small subset of the warnings raised by FindBugs (Araujo et al. 2011). Clearly, such tools have different purposes than ArchLint, but our intention here is to show that developers accept false warnings when using software analysis tools.

According to the architects of the SGA and M2M systems, most warnings generated by our approach are in fact due to violations in meaningful architectural constraints. For example, the SGA's architect commented that a relevant architecture rule in his system prescribes that "all `IService` classes must have a `Remote` annotation". The heuristic for absences was able to detect three violations in this rule.

Regarding the false positives generated by the heuristics, we observed that they can be due to a design or requirement change that implied in a bulk insertion or deletion of dependencies from a component. For example, this happened in the SGA system when the audit service (a new requirement) was introduced, adding new dependencies in many classes. Finally, we also observed that we may miss many true warnings when the system under evaluation is facing a major erosion process or when its architecture has evolved. For example, in Lucene we missed many divergences which are not "minorities" in their components, i.e., the dependencies responsible for such divergences are not spatially and historically confined in their source components.

7.2 How Difficult is to Set Up the Required Thresholds?

After applying the heuristic-based conformance process three systems, we concluded that it is not possible to rely on universal thresholds, which could be reused from system to system, especially in the case of thresholds denoting insertion and deletion rates. For example, Figures 12a and b present respectively the distribution of the scattering (*DepScaRate*) and the deletion rates (*DepDelRate*), regarding the true warnings detected by heuristic #2 for divergences. We can observe that usually the warnings present very low scattering rates. For example, the 3rd quartile values for *DepScaRate* are 2.7 % (SGA), 0.7 % (M2M),
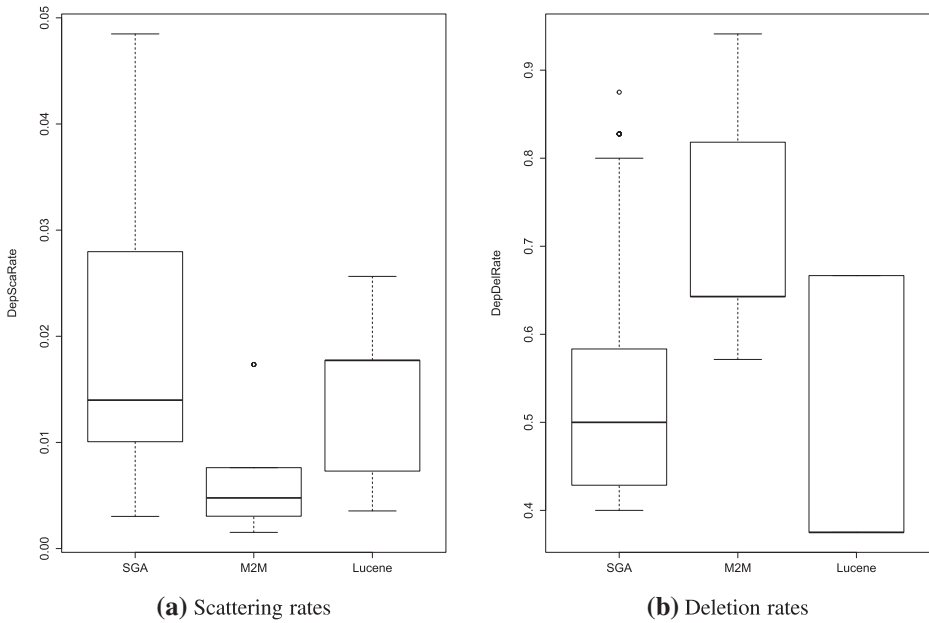
(a) Scattering rates   **(b)** Deletion rates

Fig. 12 Thresholds distribution in heuristic #2 for divergences

and 1.7 % (Lucene). On the other hand, there are more differences in terms of the dele-tion rates (*DepDelRate*). For example, the median values of *DepDelRate* are 50 % (SGA), 64 % (M2M), and 37 % (Lucene). Such differences reveal that the frequency that true architectural violations are removed varies significantly among the considered systems.

Therefore, the proposed conformance process, by allowing developers to gradually test and evaluate the required thresholds, demonstrated to be the right strategy to use the pro-posed heuristics. First, the process did not require many iterations. Considering all systems and both absences and divergences, we counted 14, 7, and 16 iterations requiring feedback from the developers in the SGA, M2M, and Lucene systems, respectively. Second, we nor-mally observed lower precision rates as soon as new iterations were executed, as expected. For this reason, we claim that the detected true warnings are not mere coincidences, but the result of spatial and temporal patterns that characterize architectural violations.

7.3 How Much Overlapping is there in the Heuristics for Divergences?

In the specific case of divergences, since we have three heuristics, it is possible for a warning to be raised by more than one heuristic. However, we observed that such warnings followed different patterns in the three systems, especially in the case of true warnings. In the SGA system, as presented in Fig. 13a, there is some intersection between the true warnings raised by the heuristics for divergences, although it is not relevant. In the M2M system, we have not found true warnings raised by more than one heuristic, as showed in Fig. 13b. Finally, in Lucene, we found an expressive intersection between heuristics #1 and #3, as showed in Fig. 13c. Also, only in Lucene we found warnings detected simultaneously by the three heuristics. In summary, our results show that each single heuristic could detect real and unique violations in at least one of the evaluated systems.
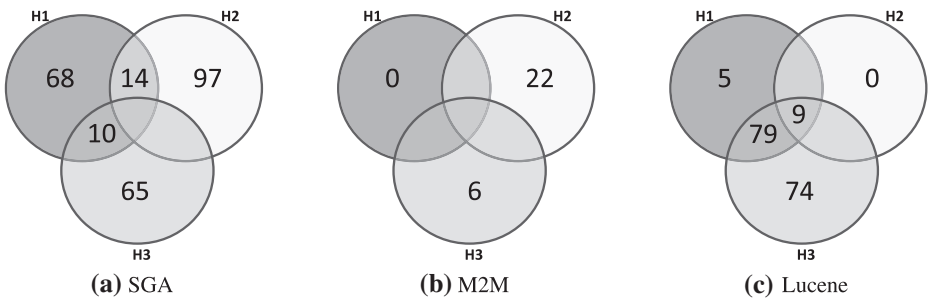
Fig. 13 Warnings raised by more than one heuristic for detecting divergences

7.4 What are the Most Common Dependency Types Responsible for Violations?

As defined in Section 2, the heuristics for absence and the second heuristic for divergence consider a violation regarding a specific dependency type. Table 13 shows the dependency types more common considering the true violations detected by these two heuristics in the SGA system. As we can observe, the most common dependency types were due to missing local variable declarations (absences) or due to unauthorized variable declarations (divergences). In the case of absences, most missing local variables are related to the implementation of the audit service. In some cases, the classes subjected to this service must inherit from `AuditInfo` (as discussed in Example #2, Section 5.2.1). In other cases, the methods requiring auditing must declare a local variable of type `AuditDAO` and call a `save` method from this class. However, the proposed heuristic for absences detected many classes whose methods do not use the audit service by declaring this local variable when they were supposed to. Regarding the divergences detected by heuristic #2, many methods were using a local variable of an incorrect type to persist data. Specifically, in many cases classes from JPA—a Java API for persistence—should have been used, but instead the code used local variables of types supporting direct access to `SQL`. In the case of absences, we also detected classes that were not inheriting for example from `br.sga.core.domain.AuditInfo` and also classes missing a `javax.ejb.Local` annotation. Finally, in the case of divergences, we also detected classes incorrectly using the `javax.persistence.OneToMany` annotation.

Table 13 Most common dependency types in the SGA system

	Absences	Heuristic #2
LocalVariable	32.8 %	42.3 %
Inheritance	21.8 %	0.0 %
DeclaredException	17.6 %	0.0 %
AnnotationClass	15.1 %	13.5 %
CaughtException	0.0 %	12.6 %
AnnotationAttribute	10.0 %	19.8 %

8 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.

8.1 Static Analysis Tools

Starting with the Lint tool (Johnson 1977) in the late seventies, several tools have been proposed to detect suspicious programming constructs by means of static analysis, including PREfix/PREfast (Larus et al. 2004) (for programs in C/C++), FindBugs (Hovemeyer and Pugh 2004), and PMD (Copeland 2005) (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, based on five stable releases of the Eclipse platform, we measured the precision of the warnings raised by two Java-based bug finding tools (Araujo et al. 2011). 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) (Kim and Ernst 2007). Using this strict definition, the precision was less than 12 %. Therefore, precision values ranging from 53.8 % (M2M System) to 62.7 % (SGA System) as the ones we achieve with ArchLint are greater than the values typically provided by traditional static analysis tools.

8.2 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, such as highly correlated method calls (Livshits and Zimmermann 2005). BugMem (Kim et al. 2006) and FixWizard (Nguyen et al. 2010) are tools that mine for repeated bug fix changes in a project's revision history (e.g., changes where an incorrect condition is replaced with 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 (Mileva et al. 2011). 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. (2013) 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 that have been done to support API modification or evolution. In contrast to previous approaches, they do not only focus on just mining bug-fixes or system releases. Palomba et al. (2013) propose an approach called HIST to detect five different code smells (Divergent Change, Shotgun Surgery, Parallel Inheritance, Feature Envy, and Blob) that are

distinguished by inspecting how the source code changed over time. Basically, they use change history information extracted from software repositories to detect bad smells by analyzing co-changes among source code artifacts over time. Using only historical analysis, their precision ranges from 61 % to 80 %, which is compatible with those found by our approach. The authors suggest that better performances can be achieved by combining static and historical analysis, as performed by our approach. Silva et al. (2014) rely on a sparse graph clustering algorithm to extract groups of classes that frequently change together, called co-change clusters. They also propose some patterns of co-change clusters, like well-encapsulated, octopus, and crosscutting, which are used to assess the traditional decomposition of systems in packages, but using historical information.

In common, the aforementioned works adopt a vertical approach for discovering project-specific patterns in software repositories (in contrast to static analysis tools that assume a horizontal approach based on a pre-defined set of bug patterns). Our approach also relies on a vertical approach, but with focus on architecture conformance.

8.3 Architecture Conformance Tools

Besides reflexion models, another common solution for architecture conformance is centered on domain-specific languages, such as SCL (Hou and Hoover 2006), LogEn (Eichberg et al. 2008), DCL (Terra and Valente 2009), Grok (Holt 1998), Intensional Views (Mens et al. 2006), and DesignWizard (Brunet et al. 2011). 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 (Terra and Valente 2009).

In a recent work, we used association rules to mine architectural patterns in version history(Maffort et al. 2013). 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 proposed in this paper. On one hand, 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 large. Hammad et al. (2009) proposed a technique based on source code changes for extracting UML class diagrams, which could be used as a first approximation for the component model required by our approach.

9 Conclusion

We conclude with the main contributions of our research both for practitioners and for software engineering researchers. First, for practitioners, especially ones who are not experts on the system under evaluation, we envision that an heuristic-based approach for architecture conformance can be used to rapidly raise architectural warnings, without deeply involving experts in the process. Moreover, after evaluating many of the warnings raised by the heuristics, practitioners can get confidence on the most relevant architectural constraints, which can be therefore formalized using languages such as DCL (Terra and Valente 2009). Moreover, especially among developers who frequently use popular static analysis tools, ArchLint can be promoted as a complementary tool that elevates to an

architectural level the warnings raised by such tools. Finally, for researchers the approach described in this paper may open a novel direction for the investigation on architectural conformance techniques, based not only on static information, but also on information extracted from version repositories, which are ubiquitously used nowadays on software projects.

As future work, we plan to evaluate new heuristics, especially heuristics that take into account the age of the changes, which can mitigate the impact that changes in architectural decisions have in our current approach. We are also working on the integration of ArchLint with ArchFix (Terra et al. 2013; 2012), which is a recommendation tool that suggests refactorings for repairing architectural violations. ArchLint—our supporting tool—is publicly available at: http://aserg.labsoft.dcc.ufmg.br/archlint

Appendix A: Formal Definition

In this appendix section, we describe the heuristics proposed by ArchLint.

Notation

The definition of the heuristics relies on the following notation:

- $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ is the set of all classes in the system under analysis.
- $CP = \{cp_1, cp_2, ..., cp_n\}$ is the set of components in the high-level component model.
- $depends(c_1, c_2, t, v)$ indicates that class c_1 has a dependency of type t with class c_2 in a given version v.
- $comp(c)$ is the component cp of a class c.
- $mod(c)$ is the module m of a class c.
- $first(c)$ is the version in which class c was originally inserted in the repository.
- H is the identifier of the last version of the system in the repository.

In a *depends* predicate, the pattern _ (underscore) matches any value. For example, $depends(c_1, c_2, \_, \_)$ indicates that class c_1 depends on class c_2, despite the dependency type and the version.

Detecting Absences

$DepCompClass(c, t, cp)$ is the set of classes in a component cp that—in the current version of the system—have a dependency of type t with a class c, as follows:

$$DepCompClass(c, t, cp) = \{x \in \mathcal{C} \mid depends(x, x, t, H) \wedge comp(x) = cp\}$$

$ClassComp(cp)$ is the set of classes in the component cp, as follows:

$$ClassComp(cp) = \{x \in \mathcal{C} \mid comp(x) = cp\}$$

$DepScaRate(c, t, cp)$ is the ratio between (i) the number of classes in component cp that have a dependency of type t with a target class c and (ii) the total number of classes in component cp, as follows:

$$DepScaRate(c, t, cp) = \frac{|DepCompClass(c, t, cp)|}{|ClassComp(cp)|}$$

$CreatedWithoutDep(c, t, cp)$ is the set of classes of a component cp that were committed in the repository for the first time without a dependency of type t with a target class c, as defined next:

$$CreatedWithoutDep(c, t, cp) = \{x \in \mathcal{C} \mid comp(x) = cp \land \neg depends(x, c, t, first(x))\}$$

$DepAdd(c, t, cp)$ is the set of classes in component cp initially created without a dependency of type t with a target class c but that later were maintained to include this dependency, as follows:

$$DepAdd(c, t, cp) = \{x \in CreatedWithoutDep(c, t, cp) \mid depends(x, c, t, H)\}$$

$DepInsRate(c, t, cp)$ is the ratio between (i) the number of classes in the component cp originally created without a dependency of type t with a target class c, but that have this dependency in the last version of the system under analysis, and (ii) the total number of classes in component cp originally created without a dependency of type t with class c, as follows:

$$DepInsRate(c, t, cp) = \frac{|DepAdd(c, t, cp)|}{|CreatedwithoutDep(c, t, p)|}$$

Finally, the candidates for absences in a component cp are defined as follows:

$$Absences(cp) = \{ (x, c, t) \mid comp(x) = cp \land \neg depends(x, c, t, H) \land$$
$$DepScaRate(c, t, p) \geq A_{sca} \land$$
$$DepScaRate(c, t, p) \geq A_{ins} \}$$

Detecting Divergences

Heuristic #1

$DepSysMod(m)$ is the set of classes in the current version of the system that have a dependency with classes of a module m, as follows:

$$DepSysMod(m) = \{x \in \mathcal{C} \mid depends(x, c, \_, H) \land mod(c) = m\}$$

$DepCompMod(m, cp)$ is the set of classes in component cp that have a dependency with a module m, as defined next:

$$DepCompMod(m, cp) = \{x \in DepSysMod(m) \mid comp(x) = cp\}$$

$DepScaRate(m, cp)$ is the ratio between (i) the number of classes in component cp that have a dependency with a module m and (ii) the total number of classes in the current version of the system that have a dependency with classes of m, as follows:

$$DepScaRate(m, cp) = \frac{|DepCompMod(m, cp|)}{|DepSysMod(m)|}$$

$DepAddAny(m, cp)$ is the set of classes in component cp that have established—in any version of the system—a dependency with a class in module m, as defined next:

$$DepAddAny(m, cp) = \{x \in \mathcal{C} \mid comp(x) = cp \wedge depends(x, c, \_, \_) \wedge mod(c) = m\}$$

$DepDel(m, cp)$ is the set of classes returned by $DepAddAny(m, cp)$ that in the current version of the system no longer have a dependency with classes in module m, as defined next:

$$DepDel(m, cp) = \{x \in DepAddAny(m, cp) \mid \neg depends(x, c, \_H) \wedge \quad mod(c) = m\}$$

$DepDelRate(m, cp)$ is the ratio between (i) the number of classes in component cp that no longer have a dependency with classes in module m and (ii) the total number of classes in component cp that have established a dependency with any class in module m, as defined next:

$$DepDelRate(m, cp) = \frac{|DepDel(m, cp)|}{|DepAddAny(m, cp)|}$$

$HeavyUser(m)$ is a function that returns the component whose classes mostly depend on classes located in module m, i.e., the component cp that provides the following maximal value:

$$\max_{\forall cp \in CP} \left(\frac{|DepCompMod(m, cp)|}{|DepSysMod(m)|} \right)$$

However, this maximal value must be greater than 0.5. Otherwise, the function $HeavyUser$ returns null.

Finally, the candidates for divergences in a given component cp are defined as follows:

$$\begin{aligned} Div_1(cp) = \{ \ (x, c) \ \mid \ & comp(x) = cp \ \wedge \ mod(c) = m \ \wedge \ depends(x, c, \_, H) \ \wedge \\ & DepScaRate(m, cp) \ \leq \ D_{sca} \ \wedge \\ & DepDelRate(m, cp) \ \geq \ D_{del} \ \wedge \\ & HeavyUser(m) \ \neq \ cp \ \} \end{aligned}$$

Heuristic #2

$DepAddAny(c, t, cp)$ is the set of classes in component cp that have established—in any version of the system—a dependency of type t with a class c, as defined next:

$$DepAddAny(c, t, cp) = \{x \in C \mid comp(x) = cp \ \wedge \ depends(x, c, t, \_) \}$$

$DepDel(c, t, cp)$ is the set of classes returned by $DepAddAny(c, t, cp)$ that no longer have a dependency of type t with a class c (i.e., the dependencies were removed), as defined next:

$$DepDel(c, t, cp) = \{ x \in DepAddAny(c, t, cp) \mid comp(x) = cp \wedge \neg depends(x, c, t, H) \}$$

Additionally, $DepDelRate(c, t, cp)$ is the ratio between (i) the number of classes in component cp that no longer have a dependency of type t with a class c, and (ii) the total number of classes in component cp that have established a dependency of type t with a class c, as defined next:

$$DepDelRate(c, t, cp) = \frac{|DepDel(c, t, cp)|}{|DepAddAny(c, t, cp)|}$$

Finally, the candidates for divergences in a given component cp are defined as follows:

$$\begin{aligned} Div_2(cp) = \{ \ (x, c, t) \mid comp(x) = cp \ \wedge \ depends(x, c, t, H) \ \wedge \\ DepScaRate(c, t, cp) \leq D_{sca} \ \wedge \\ DepDelRate(c, t, cp) \geq D_{del} \ \} \end{aligned}$$

Heuristic #3

This heuristic assumes that $rf(cp_1, cp_2)$ denotes the number of references from classes in component cp_1 to classes in component cp_2, as defined next:

$$rf(cp1, cp2) = | \ \{ \ (x, c) \mid comp(x) = cp1 \wedge comp(c) = cp2 \ \wedge \ depends(x, c, \_, H) \ \}|$$

$DepDirWeight(cp_1, cp_2)$ is defined as follows:

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

Finally, the candidates for divergences in a given component cp are defined as follows:

$$\begin{aligned} Div_3(cp_1) = \{(x, c) \mid comp(x) = cp_1 \ \wedge comp(c) = cp_2 \ \wedge cp_1 \neq cp_2 \ \wedge \\ depends(x, c, \_, H) \ \wedge \\ D_{dir} \leqslant DepDirWeight(cp_1, cp_2) < 0.5\} \end{aligned}$$

Appendix B: M2M Conformance Process

In this section, we show the results achieved after each iteration when detecting architectural violations in the M2M system. Table 14 shows the iterations performed for detecting absences. Tables 15 and 16 shows the results achieved by the second and third heuristics for detecting divergences, respectively. Heuristic #1 for divergences did not report warnings in the M2M system.

Table 14 Detecting absences in the M2M system

Iteration	A_{sca}; A_{ins}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.70; 0.55	45	45	45	77.8 %	77.8 %	0.97
2	0.60; 0.55	112	67	67	85.1 %	82.1 %	0.98

Table 15 Detecting divergences in the M2M system using Heuristic #2

Iteration	A_{sca}; A_{ins}	Warnings			Precision		
		Iter.	New	Eval.	Iter.	Overall	nDCG
1	0.05;0.90	1	1	0	—	—	—
2	0.05;0.85	3	2	0	—	—	—
3	0.05;0.80	8	5	0	—	—	—
4	0.05;0.75	10	2	10	90.0 %	90.0 %	0.96
5	0.05;0.70	14	4	0	—	—	—
6	0.05;0.65	18	4	0	—	—	—
7	0.05;0.60	42	24	32	31.3 %	45.2 %	0.52
8	0.05;0.55	51	9	0	—	—	—
9	0.05;0.50	119	68	77	3.9 %	18.5 %	0.50

Table 16 Detecting divergences in the M2M system using Heuristic #3

Iteration	D_{dir}	Warnings			Precision		
		Iter.	New	Eval.	Iter.	Overall	nDCG
1	0.25	3	3	—	—	—	—
2	0.20	5	2	—	—	—	—
3	0.10	12	7	12	41.7 %	41.7 %	1.0
4	0.05	17	5	—	—	—	—
5	0.00	48	31	36	86.1 %	75.0 %	0.94

Appendix C: Lucene Conformance Process

In this section, we show the results achieved after each iteration when detecting architectural violations in the Lucene system. Tables 17, 18, and 19 shows the results achieved by the first, second and third heuristics for detecting divergences, respectively. The heuristic for absences did not report warnings in the Lucene system.

Table 17 Detecting divergences in Lucene using Heuristic #1

Iteration	D_{sca}; D_{del}	Warnings			Precision		
		Iter.	New	Eval.	Iter.	Overall	nDCG
1	0.05;0.70	2	2	—	—	—	—
2	0.05;0.65	6	4	—	—	—	—
3	0.05;0.60	10	4	10	60.0 %	60.0 %	0.67
4	0.05;0.55	17	7	—	—	—	—
5	0.05;0.50	19	2	—	—	—	—
6	0.05;0.40	25	6	15	60.0 %	60.0 %	1.00
7	0.05;0.30	37	12	12	66.7 %	62.2 %	0.93
8	0.05;0.25	40	3	—	—	—	—
9	0.05;0.20	70	30	33	9.1 %	37.1 %	1.00
10	0.10;0.50	50	31	31	0.0 %	25.7 %	0.00
11	0.10;0.25	74	3	—	—	—	—
12	0.10;0.20	168	64	67	100.0 %	55.4 %	1.00

Table 18 Detecting divergences in Lucene using Heuristic #2

Iteration	D_{sca}; D_{del}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.05;0.90	1	1	0	—	—	—
2	0.05;0.80	3	2	0	—	—	—
3	0.05;0.75	4	1	0	—	—	—
4	0.05;0.70	7	3	0	—	—	—
5	0.05;0.65	24	17	24	12.5 %	12.5 %	0.27
6	0.05;0.50	56	32	32	3.1 %	7.1 %	0.26
7	0.05;0.40	59	3	0	—	—	—
8	0.05;0.35	97	38	41	12.2 %	9.3 %	0.41
9	0.10;0.75	21	17	17	0.0 %	7.9 %	0.00

Table 19 Detecting divergences in Lucene using Heuristic #3

Iteration	D_{dir}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.30	12	12	12	100.0 %	100.0 %	1.00
2	0.25	16	4	—	—	—	
3	0.20	34	18	22	90.9 %	94.1 %	1.00
4	0.15	98	64	64	100.0 %	98.0 %	1.00
5	0.10	128	30	30	100.0 %	98.4 %	1.00
6	0.05	142	14	14	100.0 %	98.6 %	1.00
7	0.00	164	22	22	100.0 %	98.8 %	1.00

References

Araujo JE, Souza S, Valente MT (2011) Study on the relevance of the warnings reported by Java bug-finding tools. IET Software 5(4):366–374

Baeza-Yates R, Ribeiro-Neto B (2011) Modern Information Retrieval: The Concepts and Technology Behind Search. Addison Wesley Professional

Bittencourt RA (2012) Enabling Static Architecture Conformance Checking of Evolving Software. PhD thesis, Universidade Federal de Campina Grande

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

Brunet J, Murphy GC, Serey D, Figueiredo J (2014) Five years of software architecture checking: A case study of Eclipse. IEEE Software 1–6

Copeland T (2005) PMD Applied. Centennial Books

Ducasse S, Anquetil N, Bhatti MU, Hora A, Laval J, Girba T (2011) MSE and FAMIX 3.0 an Interexchange Format and Source Code Model Family. Technical report. Software Composition Group - SCG, RMOD - INRIA Lille - Nord Europe

Ducasse S, Pollet D (2009) Software architecture reconstruction: A process-oriented taxonomy. IEEE Trans Software Eng 35(4):573–591

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

Fowler M (2002) Patterns of Enterprise Application Architecture. Addison-Wesley

Hammad M, Collard ML, Maletic JI (2009) Automatically identifying changes that impact code-to-design traceability. In: 17th IEEE/ACM International Conference on Program Comprehension (ICPC), pp 20–29

Holt RC (1998) Structural manipulations of software architecture using tarski relational algebra. In: 5th Working Conference on Reverse Engineering (WCRE), pp 210–219

Hora A, Anquetil N, Ducasse S, Valente MT (2013) Mining system specific rules from change patterns. In: 20th Working Conference on Reverse Engineering (WCRE), pp 1–10

Hou D, Hoover JH (2006) Using SCL to specify and check design intent in source code. IEEE Trans Software Eng 32(6):404–423

Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Notices 39(12):92–106

Johnson SC (1977) Lint: A C program checker. Technical Report 65, Bell Laboratories

Kim S, Ernst MD (2007) Which warnings should I fix first? In: 15th International Symposium on Foundations of Software Engineering (FSE), pp 45–54

Kim S, Pan K, Whitehead EEJ Jr. (2006) Memories of bug fixes. In: 14th International Symposium on Foundations of Software Engineering (FSE), pp 35–45

Knodel J, Popescu D (2007) A comparison of static architecture compliance checking approaches. In: 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), p 12

Koschke R (2010) Incremental reflexion analysis. In: 14th European Conference on Software Maintenance and Reengineering (CSMR), pp 1–10

Koschke R, Simon D (2003) Hierarchical reflexion models. In: 10th Working Conference on Reverse Engineering (WCRE), pp 36–45

Larus JR, Ball T, Das M, DeLine R, Fahndrich M, Pincus J, Rajamani SK., Venkatapathy R (2004) Righting software. IEEE Software 21(3):92–100

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

Maffort C, Valente MT, Anquetil N, Hora A, Bigonha M (2013) Heuristics for discovering architectural violations. In: 20th Working Conference on Reverse Engineering (WCRE), pp 222–231

Maffort C, Valente MT, Bigonha M, Hora A, Anquetil N (2013) Mining architectural patterns using association rules. In: 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp 375–380

Mens K, Kellens A, Pluquet F, Wuyts Roel (2006) Co-evolving code and design with intensional views A case study. Comput Languages Syst & Struct 32(2-3):140–156

Mileva YM, Wasylkowski A, Zeller A (2011) Mining evolution of object usage. In: 25th European conference on Object-oriented programming, pp 105–129

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

Murphy G, Notkin D, Sullivan KJ (2001) Software reflexion models bridging the gap between design and implementation. IEEE Trans Software Eng 27:364–380

Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi J, Nguyen TN (2010) Recurring bug fixes in object-oriented programs. In: 32nd International Conference on Software Engineering (ICSE), pp 315–324

Nierstrasz O, Ducasse S, Gïrba T (2005) The story of Moose: an agile reengineering environment. In: Foundations of Software Engineering (FSE), pp 1–10

Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D (2013) Detecting bad smells in source code using change history information. In: 28th International Conference on Automated Software Engineering (ASE), pp 268–278

Passos L, Terra R, Diniz R, Valente MT, Mendonca N (2010) Static architecture-conformance checking An illustrative overview. IEEE Software 27(5):82–89

Perry DE, Wolf AL (1992) Foundations for the study of software architecture. Software Eng Notes 17(4):40–52

Sarkar S, Maskeri G, Ramachandran S (2009) Discovery of architectural layers and measurement of layering violations in source code. J Syst Software 82:1891–1905

Santonu S, Ramachandran S, Kumar GS, Madhu K, Iyengar K, Rangarajan, Sivagnanam S (2009) Modularization of a large-scale business application: A case study. IEEE Software 26:28–35

Silva L, Valente MT, Maia M (2014) Assessing modularity using co-change clusters. In: 13th International Conference on Modularity, pp 49–60

Terra R, Valente MT (2009) A dependency constraint language to manage object-oriented software architectures. Software: Practice and Experience 32(12):1073–1094

Terra R, Valente MT, Czarnecki K, Bigonha R (2012) Recommending refactorings to reverse software architecture erosion. In: 16th European Conference on Software Maintenance and Reengineering (CSMR), pp 335–340

Terra R, Valente MT, Czarnecki K, Bigonha RS (2013) A recommendation system for repairing violations detected by static architecture conformance checking. Software: Practice and Experience 1–36

Cristiano Maffort received his PhD in Computer Science from the Federal University of Minas Gerais, Brazil (2014). He is now an associate professor in the Department of Computing at CEFET-MG, Brazil. His research interests include software architecture and software maintenance and evolution.

Marco Tulio Valente received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an associate professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a "Researcher I-D" of the Brazilian National Research Council (CNPq). He also holds a "Researcher from Minas Gerais State" scholarship, from FAPEMIG. Valente has co-authored more than 60 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.

Ricardo Terra received his PhD degree in Computer Science from the Federal University of Minas Gerais (DCC/UFMG) with a 1-year internship at the University of Waterloo. Terra is an associate professor in the Department of Computer Science at the Federal University of Lavras (DCC/UFLA). His research interests include software architecture and software maintenance and evolution. Currently, he co-heads the Applied Software Engineering Research Group (ASERG), at DCC/UFLA.

Mariza A. S. Bigonha is an Associate Professor at the Computer Science Department, Universidade Federal de Minas Gerais, (UFMG), Brazil. She received her MSc degree in Computer Science from UFMG, in 1985, and her PhD degree in Informatics: Computer Science from Pontifícia Universidade Católica do Rio de Janeiro, Brazil in 1994. Her main research interests include Programming Languages and Software Engineering. She has specific interest on software modularity and measurement methods that should be part of a software quality assessment process. She has published more than 30 papers in international journals and conference proceedings.

Nicolas Anquetil is been assistant professor at the University of Lille-1, France, since September 2009. He obtained his PhD in 1996 from the University of Montreal, Canada. He also worked at University of Ottawa, Canada; Federal University of Rio de Janeiro, Brazil; Catholic University of Brasilia, Brazil; and, Ecole des Mines de Nantes, France. His research focuses on software evolution and maintenance at large which already included work on software re-architecturing, knowledge management for software maintenance, or software maintenance management. He is best known for his work on software re-architecturing.
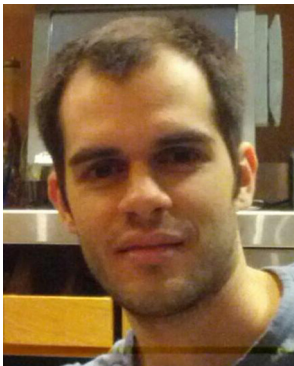
Andŕe Hora is a Post-doctoral fellow in Software Engineering in the ASERG Group at the Department of Computer Science, Federal University of Minas Gerais, Brazil. He received an MSc degree from Federal University of Minas Gerais in 2010 and a PhD degree from Université Lille 1/Inria, France, in 2014. His main research interests include software evolution, reverse engineering, software analysis, and empirical software engineering.