Mining Architectural Patterns Using Association Rules

Cristiano Maffort, Marco Tulio Valente, Mariza Bigonha
Department of Computer Science
UFMG, Belo Horizonte, Brazil
{maffort,mtov,mariza}@dcc.ufmg.br

André Hora, Nicolas Anquetil
RMoD Team
Inria, Lille, France
{andre.hora,nicolas.anquetil}@inria.fr

Jonata Menezes
Department of Computer Engineering
CEFET-MG, Belo Horizonte, Brazil
jonata@dri.cefetmg.br

***Abstract*—Software systems usually follow many programming rules prescribed in an architectural model. However, developers frequently violate these rules, introducing architectural drifts in the source code. In this paper, we present a data mining approach for architecture conformance based on a combination of static and historical software analysis. For this purpose, the proposed approach relies on data mining techniques to extract structural and historical architectural patterns. In addition, we propose a methodology that uses the extracted patterns to detect both absences and divergences in source-code based architectures. We applied the proposed approach in an industrial-strength system. As a result we detected 137 architectural violations, with an overall precision of 41.02%.**

***Keywords*-Software architecture conformance; Frequent itemset mining; Static analysis; Mining software repositories**

I. Introduction

The architecture of a system prescribes the organization of its components, their relationships, constraints, and the principles that guide its design and evolution over time [1]–[3]. An architectural model is a high-level representation of the software that documents and transmits the major decisions and principles that should be followed during the software development project.

However, during development of a software product, programming anomalies regarding the proposed architectural model are normally introduced. These anomalies are classified in this paper as architectural violations [4], [5]. In practice, the introduction of architectural violations is very common [6]. These violations usually make more complex subsequent maintenance tasks since the concrete architecture is not adhering to the planned and documented architecture [7].

Therefore, in this paper we assume that the inception of architectural violations in software products is a common task [8]. Moreover, we assume that some violations are detected and corrected in future revisions through inspection and/or quality assure activities. Furthermore, programs usually follow architectural patterns of implementation. With this in mind, it is observed that, according to software design best practices, classes belonging to the same component follow similar programming conventions.

Based on these assumptions, this paper proposes a method to detect architectural violations in software products. The proposed solution analyzes structural and historical architectural patterns at the level of structural dependencies between classes and considering the versions stored in a version control repository. The ultimate goal is to identify architectural violations from similar dependency patterns. Particularly, a structural dependency denotes any syntactic relation between two classes, including method calls, field and variables declaration, etc.

The proposed approach extracts architectural patterns that can be used, for example, as documentation artifacts. Furthermore, the detection method is statically performed in a non-invasive way, so it does not impact normal system programming activities. In order to evaluate our approach, this paper describes its application in a real information system used by a major Brazilian university. As result, we identified 334 evidences of architectural violations in this system. From such evidences, 137 were confirmed by a senior software developer, which implies in a precision of 41.02%.

The remainder of this paper is organized as follows. Section II presents an overview of the proposed approach. Sections II-A and II-B describe the heuristics to detect absences and divergences, respectively. Section III presents an evaluation of the proposed approach in a real system. Section IV describes related work and Section V presents the conclusions.

II. Proposed Approach

This paper proposes a technique for detecting architectural violations in object-oriented software systems. The proposed approach relies on data mining techniques over historical dependencies between the classes of a target system. This historical information is retrieved from the version control system repository. Basically, the proposed approach mines structural and historical dependencies between the classes of the target system.

Figure 1 illustrates our approach for detecting architectural violations. Initially, a *Code Extractor* component retrieves all source code versions from the version control system repository. Each revision is parsed by the VerveineJ[1] parser that extracts the dependencies from the source code. Next, the extracted dependencies are stored in a relational database. The *Architectural Miner* component relies on two types of input on the target system: (a) dependencies database and (b) high-level component specification. In our approach, we assume that classes are statically organized in modules (packages in Java terminology)

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

and modules are logically arranged in coarse–grained structures called components. The high-level component specification is essentially a mapping from modules to the defined components. Next, the *Architecture Miner* generates a Prolog database describing the structural and historical relations available in the source code. After that, the *Architectural Miner* uses Prolog queries to convert the Prolog database into a consistent frequent itemset mining dataset. Next, an association rule mining algorithm is used to detect structural and historical architectural patterns. Finally, the *Violation Detector* uses such architectural patterns to detect architectural violation evidences according to the methodology described in Sections II-A and II-B.

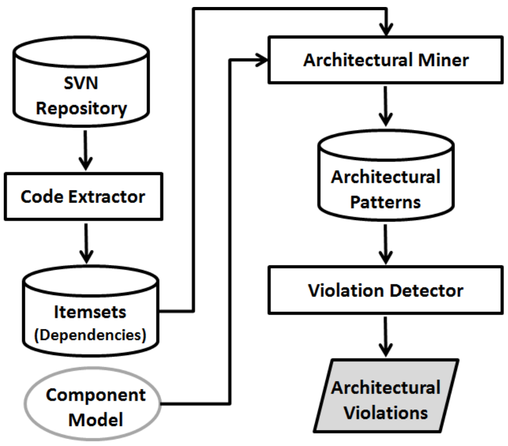

Figure 1. Proposed approach

In the proposed component model, high-level components are represented as simple regular expressions that represent the mapping from modules to components.

Our approach is based on a data mining technique called *frequent itemset mining* [9], which efficiently finds frequent itemsets in a dataset. Basically, this technique defines the *support* as the number of occurrences of a subset of items (sub-itemset). A sub-itemset is considered frequent if its support is greater than a specified threshold called *minimum support*. Thus, support counts the number of times a sub-itemset happens in the itemsets database.

After the frequent itemset has been mined, we can compute *association rules* [10], [11]. From the association rules, we make assumptions that two or more items occur simultaneously or conditionally. Furthermore, association rules can be used to discover causal relationships among elements. Each association rule has a *confidence*, which is a metric that represents the probability of a database transaction covered by a antecedent term (pre-condition of the rule) be covered by a consequent term (consequence of the rule).

To calculate frequent itemsets and to generate association rules, we use a FP-tree-based mining algorithm, called FPGrowth [10]. Instead of generating the complete set of frequent sub-itemsets, this algorithm generates only relevant itemset candidates. After the frequent itemsets are mined, FPGrowth also generates association rules. The proposed approach to architectural violation detection is based on the idea that an architectural pattern is frequently followed and violations represent a small percentage of the cases.

The remainder of this section is organized as follows: Section II-A presents the heuristics used to detect evidences of absences; Section II-B describes the heuristics for divergences.

A. Mining for Absences

An absence is a violation that occurs when a $BaseClass$ does not depend on a $TargetClass$, but a dependency like that *is prescribed* by the planned architecture. In other words, an absence is a violation that happens with a dependency defined by the planned architecture but that does *not exist* in the source code [4], [12]. Figure 2 illustrates an example of absence. In this case, the planned architecture prescribes that classes located in a DTO module must use services provided by a class located in JPA module. In this case, an absence is counted for each class in DTO that does not follow this rule.

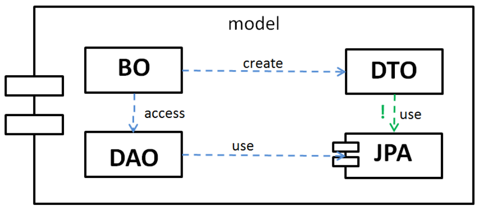

Figure 2. Example of absence (DTO must use JPA)

In order to detect absences we initially search for patterns of dependencies that frequently occur. Next, we search for dependencies that violate such patterns, and therefore denote minorities at the level of components. We assume that absences occur in a small percentage of cases, which are more likely to represent architectural violations. Additionally, we use the history of versions to mine for evolutionary architectural patterns. In this case, we search for patterns in which dependencies are introduced in classes originally created without such dependencies.

The proposed procedure for detecting absences relies on two steps. First, we identify architectural patterns that *frequently occur* in classes grouped as defined by the component model provided as input. Second, from classes in each component, we identify evolutionary architectural patterns. For instance, considering the example in Figure 2, we check how frequently classes in the component $Model$ that depend on $Entity$ (a class of the JPA module) in the current version of the system were initially created without this dependency.

The main idea of the evolutionary architectural patterns is to reinforce the violation evidences suggested by the first step. The assumption is that absences are frequently detected and fixed (i.e., classes created without a dependency prescribed by the planned architecture are frequently fixed in future revisions).

In order to find correlations among the dependencies, initially it is necessary to compute the frequent itemset mining dataset. For this purpose, we rely on a dataset based on Prolog facts, which describes the dependencies and historical information on the classes of the system under analysis, as follows:

```
[component(CompId,CompName).]+
[module(ModId,CompId,ModName).]+
[class(ClassId,ModId,ClassName).]+
[dependency(DepId,BaseClassId,TargetClassId,
    CreatedWith,ExistCurrently,AddAny).]+
```

The *component* predicate defines the components defined by the architect of the system under analysis. The *module* predicate defines the packages, in Java terminology, of classes of the system. The *class* predicate describes a class in the system. The *dependency* predicate defines a dependency relation between two classes ($BaseClassId$ depends on $TargetClassId$). In the *dependency* predicate, the attribute $CreatedWith$ informs whether the dependency was created together with the $BaseClassId$, the attribute $ExistCurrently$ informs whether the dependency exists on the last version of the system, and the attribute *AddAny* informs if the dependency existed in some version of the system.

In the first step, each class and its dependencies in the last version under analysis (attribute $ExistCurrently = true$) are written as a row into the itemset database, as follows:

```
BaseComponent(bcomp),BaseClass(bclass)
    [,TargetModule(tmod),TargetClass(tclass)]*
```

By mining this itemset database using FPGrowth algorithm, we can find the frequent sub-itemsets and generate the association rules of the corresponding *architectural pattern*, which represent dependencies that are frequently used together. Moreover, the FPGrowth requires the definition of a support (A_{dps}) and a confidence (A_{dpc}) threshold. For instance, suppose a pattern like that:

```
{BaseComponent('domain')}=>
    {TargetClass('Entity')}
```

This pattern states that all classes on the component $domain$ (antecedent term of the association rule) should depend on the class $Entity$ (consequent term of the association rule). Therefore, regarding this pattern, classes in the $domain$ component that do not depend on $Entity$ represent an absence violation.

The second step is used to reduce the amount of false violations. For each component in the system, we select the dependencies and the historical information from the Prolog facts database. In this particular case, we select the attributes $CreatedWith$ and $ExistCurrently$. Each dependency generates a row in the itemset database as follows:

```
BaseComponent(bcomp),TargetClass(tclass),
    CreatedWith([true|false]),
    ExistCurrently([true|false])
```

We compute the association rules of the corresponding *dependency evolution patterns* using the FPGrowth algorithm, using a given support (A_{deps}) and confidence (A_{depc}) threshold. The results are combined with the results obtained in the first step. For example, suppose that in the first step the classes in the $Model$ component that not depend on $Entity$ were classified as evidences of absences. Moreover, suppose that in the second step we found that classes in $Model$ created without a dependency with $Entity$ frequently (i.e., with a high confidence) added this dependency during their evolution, which therefore reinforces the evidence detected in the first step.

B. Mining for Divergences

A divergence is a violation that happens when a $BaseClass$ depends on a $TargetClass$, although such dependency *is not prescribed* by the planned architecture. In other words, a divergence is a violation due to a dependency that is not allowed by the planned architecture, but that *exists* in the source code [4], [12]. Figure 3 illustrates an example of divergence. In this case, the planned architecture prescribes that classes located in the *BO* module must not directly depend on the *JPA* module. In this particular example, a divergence is counted for each class in *BO* which relies on services provided by the *JPA*.

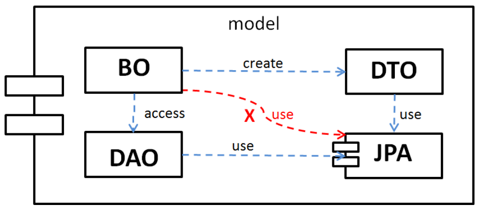

Figure 3. Example of divergence (BO cannot use JPA)

Likewise the heuristic for absences, we assume that divergences happen in a *small percentage* of cases. Therefore, a standard frequent itemset mining technique is not suitable for detecting minorities. However, divergences frequently do *not exist* in most classes of a component. More specifically, the divergences detection relies on two steps. First, we identify the dependencies that frequently do not occur in the classes of a given component. In the second step, we identify how frequently classes in this component have established and then removed a dependency like that in the past.

In the first step, we initially select all classes in the last version of the target system. For each $BaseClass$, we select the dependencies that do *not exist* between $BaseClass$ and a $TargetClass$, where $TargetClass$ is a class used by the component that contains $BaseClass$. Then, these items generate a row in the itemset database, as follows:

```
BaseComponent(bcomp),BaseClass(bclass)
    [,TargetModule(tmod),TargetClass(tclass)]*
```

Using FPGrowth algorithm, we compute the association rules, according to a given support (D_{dps}) and confidence (D_{dpc}). For instance, the following association rule states that classes in the $Model$ component frequently do not depend on $HttpServlet$.

```
{BaseComponent('model')}=>
    {TargetClass('HttpServlet')}
```

Therefore, the classes in $Model$ that depend on $HttpServlet$ represent an evidence of divergence.

In the second step, we perform a historical analysis to reduce the number of false positives. In this case, we select dependencies from the itemset database including the attributes $ExistCurrently$ and $AddAny$. This information generates an itemset in our database as follows:

```
BaseComponent(bcomp),TargetClass(tclass),
    AddAny([true|false]),
    ExistCurrently([true|false])
```

Applying the FPGrowth, using D_{deps} and D_{depc} as support and confidence respectively, we obtain the association rules for the *dependency evolution patterns*. Then, these results are combined with the results obtained in the first step. For instance, suppose that it was previously mined in the first step that the classes in the $Model$ component that depend on $HttpServlet$ represent evidences of divergences. Moreover, suppose that in this second step we discover that such classes removed the dependencies with $HttpServlet$ during their evolution. In this case, the evidence detected in the first step is reinforced by this second finding.

III. Evaluation

To evaluate our approach for detecting absences and divergences, we performed a study in a information system, called SGA, from a major Brazilian university. The SGA system automates many administrative activities, including human and material resource management, incomes/expenses, among others. The last revision considered in our study has 1,852 classes and interfaces, organized in 104 packages, comprising around 127 KLOC.

The SGA system follows a Model-View-Controller (MVC) architecture. The $Model$ layer has three main modules: $domain$, $persistence$, and $service$. The $domain$ module handles business objects, such as Students, Professors, etc. The $persistence$ module provides database transactional methods, such as insert, update, delete, etc, that are used to persist business objects in a relational database. The $service$ module handles the state of the domain objects according to the workflow and business rules required by the information system.

The $View$ layer is implemented in *Java Server Pages* and uses *JavaServer Faces* components. Basically, this layer provides a way to interact with the system, receiving and displaying results of the requests made by the users.

The $Controller$ layer provides a bridge between user interface and business-related components, transferring and adapting the user inputs.

A. Dataset

To detect absences and divergences, initially we retrieved 4,923 revisions of the SGA system, which is maintained in a $Subversion$ repository. Each revision was parsed by VerveineJ and the extracted dependencies were stored in a relational database with 4.5 GB.

Next, an architect defined its high-level component model. Finally, the high-level components and the dataset of historical dependencies were used as input to generate the Prolog facts. We executed our approach as described in Sections II-A and II-B. Then, the architect of the SGA system inspected the selected violations in order to classify them as true or false positives.

B. Results for Absences

As reported in Section II-A, the detection of absences relies on four thresholds: A_{dps} and A_{dpc}, the support and confidence of the structural dependency architectural patterns, and A_{deps} and A_{depc}, the support and confidence of historical dependency evolution patterns. Table I shows the values used for such thresholds:

Table I
Absences Thresholds.

Threshold	Value
A_{dps}	0.1
A_{dpc}	0.9
A_{deps}	0.1
A_{depc}	0.6

Basically, we consider as an architectural pattern only the rules that occurred in at least 10% of the classes and that present a confidence of at least 90%. For the architectural evolution patterns, we consider thresholds of 10% for support and 60% for confidence. Therefore, we consider as evidence of architectural violation classes that violated a rule followed by at least 90% of the other classes. Furthermore, only classes whose historical evolution rules were higher than 60% were considered as violations, i.e., at least 60% of the classes created with a violations regarding the rule have been later refactored to follow the rule.

C. Results for Divergences

The detection of divergences relies on four thresholds: D_{dps} and D_{dpc}, denoting respectively the support and confidence of the structural dependency architectural patterns, and D_{deps} and D_{depc}, denoting respectively the support and confidence of the historical dependency evolution patterns. Table II shows the thresholds values used for divergences.

Table II
Divergences Thresholds.

Threshold	Value
D_{dps}	0.1
D_{dpc}	0.9
D_{deps}	0.1
D_{depc}	0.25

Similarly to the absence detection, for divergences we consider architectural pattern rules with support of 10% and confidence of 90%. On the other hand, for the architectural evolution patterns, we consider the thresholds of 10% and 25% for support and confidence, respectively. In this case, we select as divergences the classes that violate both considered architectural patterns. More specifically, we select classes that depend on a class when at least 90% of the classes in the same component do not follow this rule. Furthermore, when in the past other classes added this dependency, in at least 25% of the cases the dependency was later removed.

D. Results

Our approach was applied in the SGA system using the thresholds defined in Sections III-B and III-C. The triggered violations were inspected by the SGA architect, who classified them as true or false violations.

As we can observe in Table III, we detected 261 evidences of absence, and 101 were classified as true-positives by the SGA architect. Furthermore, we triggered 73 divergence warnings, which 36 were classified as true-positives. Thus, the precision was 38.7% and 49.32% to absences and divergences, respectively. As total, the architect inspected 334 warnings, which 137 were considered true-positives, resulting in a global precision of 41.02%.

Table III
ARCHITECTURAL VIOLATIONS OF SGA SYSTEM.

	Absence	**Divergence**	**Total**
Warnings (E)	261	73	334
True-positives (TP)	101	36	137
False-positives (FP)	160	37	197
Precision (TP/E)	38.7%	49.32%	41.02%

IV. RELATED WORK

Lint [13] was one of the earliest and most successful tool to detect bugs and bad smells in software products. With the success achieved by Lint, many other static analysis tools to detect questionable programming strategies have been proposed. FindBugs [14] and PMD [15] are examples of tools inspired by Lint that highlight among the most popular tools to detect anomalies on Java programs. Null pointer dereferences, overflow in arrays, uncaught exceptions and security vulnerabilities are examples of suspicious programming constructs and events analyzed by FindBugs and PMD. However, such tools are not designed to detect architectural anomalies such as the ones associated to violations in the planned architecture of object-oriented systems. Moreover, in previous studies we concluded that FindBugs present precision rates less than 50%, which are only achieved when the tool is properly configured to raise particular categories of warnings [16]. In other study, we concluded that there is no static correspondence between field defects and warnings raised by FindBugs, although it seems to exist a moderate level of correlation between warnings and such kinds of software defects [17].

Several tools have been proposed to analyze version control software repositories and extract programming patterns. Zhou and Zhenmin present a tool, called PR-Miner (based on the frequent itemset data mining technique) for automatic extraction of programming rules [18]. The proposed approach uses a formalism to extract dependencies between functions that are heavily dependent on procedural languages. The proposed strategy for detecting violations only considers function call flows, independent of the modular and/or architectural context in which these calls occurred. On the other hand, the approach presented in this paper is focused on the detection of architectural violations. However, it is important to note that the precision values presented by PR-Miner were generally lower than those reported in Section III. For example, from the 60 warnings with higher priority triggered during Linux analysis, only 16 were true programming errors (bugs).

Mileva et al. conducted an analysis on evolution patterns between two versions of a system to detect pending changes in source code [19]. This study is supported by a tool called Lamarck. Such tool mines evolution patterns in software repositories by abstracting object usage into temporal properties in order to detect pending changes. From the pending changes, they recommend fixes based on usage patterns. Similarly to the study of Zhou and Zhenmin the approach used by Mileva et al. also analyze dependencies between functions. Therefore, it targets concepts of procedural languages, disregarding typical object-oriented language dependencies, such as inheritance. Moreover, in this paper, we take into account, in addition to the present formalism of object-oriented languages, the entire history of changes. Despite these differences in focus and languages, it is important to note that the approach described in this paper, in general, was able to discover a greater number of violations. For example, the approach of Mileva et al. was able to find only six violations in a case study involving Eclipse 1.0 and 2.0 platforms.

Among architecture conformance techniques, reflexion models currently highlight as the main technique based on models. Such approach compares a high-level model, manually created by the architect, with a concrete model, automatically extracted from the source code [12], [20]. However, the application of reflexion models for architecture conformance usually requires successive refinements in the high-level model to reveal the whole spectrum of absences and divergences. On the other hand, the approach proposed in this paper is more lightweight, demanding a simple high-level component specification.

Sarkar et al. [7] conducted a study aiming to discover layered organization models of software systems. The architectural model generated was used to detect architectural violations through dependencies among modules. They have only detected calls violating layer hierarchical structures. On the other hand, in this paper, we have also presented a methodology to detect absences between two layers and divergences between components, which do not necessarily need to follow a hierarchical structure.

Besides reflexion models, another common solution for architecture conformance is centered on domain-specific languages. In this case, Terra and Valente proposed a declarative language, called DCL (Dependency Constraint Language), for constraint dependency that statically checks the architecture of a software in relation to restrictions defined by an architect [8]. Therefore, DCL requires an architect to define constraints, and a tool included in the solution verifies only what the architect has prescribed. On the other hand, in our approach the architect does not need to manually specify architectural constraints, which invariably tends to be a tedious and error-prone task.

V. Conclusion

Software systems frequently follow several programming conventions. During software evolution, the development team commonly uses programming strategies that do not adhere to the planned architecture for the system.

This paper presented an approach that use frequent itemset mining techniques to architecture conformance. We consider both dependencies prescribed in the planned architectural model but absent in the source code, as well as dependencies presented in the source code but absent in architectural model.

We evaluated the proposed approach with a large information system. We detected 137 architectural violations, divided in 101 absences and 36 divergences, with precision of 38.7% and 49.32%, respectively, and a global precision of precision of 41.02%.

As future work, we intend to extend the study evaluating specific correlations between dependencies as well as classifying the dependency type, such as attributes, annotations, inheritance, etc. Additionally, we intend to conduct a sensibility analysis in order to discover the best combination of values for the thresholds used by our approach. Finally, we plan to apply our study in case studies involving systems using architectural patterns different from SGA system. We also plan to integrate our approach for architecture conformance with ArchFix [21], which is a recommendation tool that suggests refactorings for repairing architectural violations.

Acknowledgments

This research has been supported by CAPES, FAPEMIG, and CNPq. We thank the architect of the SGA system for validating the warnings raised by proposed approach.

References

[1] D. Garlan, "Software architecture: a roadmap," in *Conference on The Future of Software Engineering*, ser. ICSE '00, 2000, pp. 91–101.

[2] D. Garlan and M. Shaw, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[3] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

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

[5] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Engineering Notes*, vol. 17, pp. 40–52, 1992.

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

[7] S. Sarkar, G. Maskeri, and S. Ramachandran, "Discovery of architectural layers and measurement of layering violations in source code," *Journal of Systems and Software*, vol. 82, pp. 1891–1905, 2009.

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

[9] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *20th International Conference on Very Large Data Bases*, 1994, pp. 487–499.

[10] M. J. Zaki and W. Meira Jr., *Fundamentals of Data Mining Algorithms*. Cambridge University Press, 2011.

[11] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *International Conference on Management of Data*, 1993, pp. 207–216.

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

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

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

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

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

[17] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

[18] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *13th Symposium on Foundations of Software Engineering (FSE)*, 2005, pp. 306–315.

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

[20] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *10th European Conference on Software Maintenance and Reengineering (CSMR)*, 2006, pp. 279–294.

[21] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, 2012, pp. 335–340.