

**MINERAÇÃO DE VIOLAÇÕES ARQUITETURAIS
USANDO HISTÓRICO DE VERSÕES**

CRISTIANO AMARAL MAFFORT

MINERAÇÃO DE VIOLAÇÕES ARQUITETURAIS USANDO HISTÓRICO DE VERSÕES

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: MARCO TÚLIO DE OLIVEIRA VALENTE
COORIENTADORA: MARIZA ANDRADE DA SILVA BIGONHA

Belo Horizonte
Outubro de 2014

CRISTIANO AMARAL MAFFORT

MINING ARCHITECTURAL VIOLATIONS FROM VERSION HISTORY

Thesis presented to the Graduate Program
in Computer Science of the Universidade
Federal de Minas Gerais in partial
fulfillment of the requirements for the
degree of Doctor in Computer Science.

ADVISOR: MARCO TÚLIO DE OLIVEIRA VALENTE
CO-ADVISOR: MARIZA ANDRADE DA SILVA BIGONHA

Belo Horizonte

October 2014

© 2014, Cristiano Amaral Maffort.
Todos os direitos reservados.

Maffort, Cristiano Amaral

M187m Mining Architectural Violations from Version
History / Cristiano Amaral Maffort. — Belo
Horizonte, 2014
xxi, 109 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas
Gerais

Orientador: Marco Túlio de Oliveira Valente

Coorientadora: Mariza Andrade da Silva Bigonha

1. Computação - Teses. 2. Engenharia de software -
Teses. 3. Software - Verificação - Teses. I. Orientador.
II. Coorientador. III. Título.

CDU 519.6*32.(043)



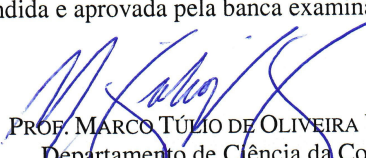
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

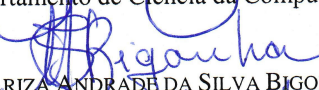
FOLHA DE APROVAÇÃO

Mining architectural violations from version history

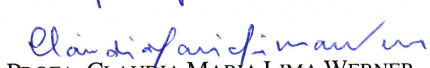
CRISTIANO AMARAL MAFFORT


Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

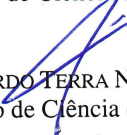

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG


PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Coorientadora
Departamento de Ciência da Computação - UFMG


PROF. ALESSANDRO FABRICIO GARCIA
Departamento de Informática - PUC/RJ


PROFA. CLAUDIA MARIA LIMA WERNER
Centro de Tecnologia - UFRJ


PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG


PROF. RICARDO TERRA NUNES BUENO VILLELA
Departamento de Ciência da Computação - UFLA

Belo Horizonte, 09 de outubro de 2014.

Agradecimentos

Agradecer não é, de modo algum, uma tarefa trivial. São tantas pessoas a agradecer que sempre se corre o risco de, equivocadamente, não se fazer devida menção a alguém. Então, objetivamente, começarei por agradecer a todos que de alguma forma me ajudaram, direta ou indiretamente. Assim, ninguém estará completamente esquecido.

De forma especial, inicialmente, agradeço ao CEFET-MG e ao Departamento de Computação por terem concedido liberação das minhas atividades docentes para que eu pudesse me dedicar integralmente ao doutoramento.

Ao Programa de Pós-Graduação em Ciência da Computação e à CAPES/CNPq pelo apoio financeiro.

Aos funcionários da Secretaria do PPGCC, sempre prestativos e competentes.

Aos amigos do LLP pelo companheirismo e amizade. Sentirei falta das agradáveis divagações diárias sobre os mais variados assuntos durante o almoço no "bandeco".

Aos meus familiares pelo apoio incondicional e por estarem sempre ao meu lado, especialmente aos meus irmãos Leandro e Patrícia, à minha filha Amanda, à minha namorada Letícia, à minha avó Terezinha e aos meus tios Ronaldo e Neide.

À Professora Mariza, minha coorientadora, pelas incontáveis contribuições para realização desse trabalho.

Finalmente, agradeço especialmente ao Professor Marco Túlio, pela paciência, comprometimento, disponibilidade e dedicação. Um profissional exemplar. Um amigo por quem tenho eterna gratidão.

Resumo

Verificação de conformidade arquitetural é uma atividade chave para controle da qualidade de sistemas de software, tendo como objetivo central revelar diferenças entre a arquitetura concreta e a arquitetura planejada de um sistema. Entretanto, especificar a arquitetura de um software é uma tarefa difícil, já que ela deve ser realizada por um especialista. Nesta tese de doutorado, propõe-se uma nova abordagem para verificação de conformidade arquitetural baseada na combinação de técnicas de análise estática e histórica de código fonte. Propõem-se quatro heurísticas para detectar ausências (dependências esperadas, mas inexistentes) e divergências (dependências proibidas, mas presentes) no código fonte de sistemas orientados por objetos. A abordagem proposta também inclui um processo iterativo para verificação de conformidade arquitetural, o qual foi utilizado para avaliar a arquitetura de dois sistemas de informação de grande porte, tendo sido capaz de identificar 539 violações, com precisão de 62,7% e 53,8%. Além disso, foram avaliados dois sistemas de código aberto, nos quais foram identificadas 345 violações, com precisão de 53,3% e 59,2%. De forma complementar, apresenta-se um estudo exploratório sobre a aplicação de uma técnica de mineração de dados, chamada mineração de itens frequentes, para detectar padrões arquiteturais a partir de informações estáticas e históricas extraídas do código fonte. Em seguida, esses padrões foram usados para detectar ausências e divergências no código de um sistema. Neste segundo estudo, foram detectadas 137 violações arquiteturais, com precisão global de 41,2%.

Palavras-chave: Arquitetura de Software, Erosão Arquitetural, Conformidade Arquitetural, Análise Estática, Mineração de Repositórios de Software.

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 software architectures is not a trivial task, as it must be done by an expert on the system under analysis. In this thesis, we present an approach for architecture conformance based on a combination of static and historical source code analysis. The proposed approach relies on four heuristics for detecting both 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 the architecture of two industrial-strength information systems, when 539 architectural violations were detected, with an overall precision of 62.7% and 53.8%. We also evaluated our approach in two open-source systems, when 345 architectural violations were detected, achieving an overall precision of 53.3% and 59.2%. Additionally, this thesis presents an exploratory study on the application of a data mining technique called frequent itemset mining, which was used to detect architectural patterns using static and historical information extracted from source code. Furthermore, the detected architectural patterns are used to identify absences and divergences in the code. We evaluated the proposed approach in an industrial-strength information system, founding 137 architectural violations, with an overall precision of 41.2%.

Keywords: Software Architecture, Architectural Erosion, Architectural Conformance, Static Analysis, Mining Software Repositories.

List of Figures

1.1	Proposed approach to architectural conformance checking	4
2.1	DSM example (class B depends on class A)	11
2.2	Database representations	21
2.3	Formal context of “famous animals” [Priss, 2006]	23
2.4	Concept lattice of Figure 2.3 [Priss, 2006]	24
3.1	Input and output of the proposed heuristics	30
3.2	Example of absence (C_2 does not depend on <i>TargetClass</i>). The label <i>Ins</i> denotes a dependency inserted later in the class	31
3.3	Example of divergence (C_2 depends on <i>TargetModule</i>). The label <i>Del</i> denotes a dependency removed in a previous version of the class	34
3.4	Example of divergence (C_2 depends on <i>TargetClass</i>). The label <i>Del</i> denotes a dependency removed in a previous version of the class	35
3.5	Divergences due to asymmetrical cycles	37
3.6	ArchLint architecture	44
3.7	Architecture conformance using the proposed heuristics	46
3.8	Initial thresholds values and thresholds adjustment procedures for each heuristic	48
4.1	Enriched high-level model for the SGA system	59
4.2	Absences and divergences detected by RM and the proposed heuristics . .	59
4.3	Distribution of the refactoring operations by year	62
4.4	Thresholds distribution in heuristic #2 for divergences	76
4.5	Warnings raised by more than one heuristic for detecting divergences . . .	77
5.1	Data Mining proposed approach	84
5.2	Example of absence (<i>DTO</i> must use <i>JPA</i>)	86
5.3	Example of divergence (<i>BO</i> cannot use <i>JPA</i>)	89

List of Tables

2.1	Techniques for detecting programming anomalies	26
2.1	Techniques for detecting programming anomalies	27
2.1	Techniques for detecting programming anomalies	28
3.1	Dependency types, assuming that C_1 depends on C_2	32
4.1	High-level components in the SGA system	51
4.2	Detecting absences in the SGA system	53
4.3	Detecting divergences in the SGA system using Heuristic #1	55
4.4	Detecting divergences in the SGA system using Heuristic #2	56
4.5	Detecting divergences in the SGA system using Heuristic #3	57
4.6	Precision considering the warnings evaluated for three heuristics for divergences	58
4.7	Historical analysis results	61
4.8	High-level components in the M2M system	63
4.9	Precision considering the warnings raised in M2M system	64
4.10	Detecting absences in the M2M system	65
4.11	Detecting divergences in the M2M system using Heuristic #2	66
4.12	Detecting divergences in the M2M system using Heuristic #3	67
4.13	High-level components in Lucene	67
4.14	Precision considering the warnings raised in Lucene system	68
4.15	Detecting divergences in Lucene using Heuristic #1	69
4.16	Detecting divergences in Lucene using Heuristic #2	70
4.17	Detecting divergences in Lucene using Heuristic #3	70
4.18	High-level components in ArgoUML	71
4.19	Precision considering the warnings raised in ArgoUML system	72
4.20	Detecting divergences in ArgoUML using Heuristic #1	73
4.21	Detecting divergences in ArgoUML using Heuristic #2	74

4.22	Detecting divergences in ArgoUML using Heuristic #3	74
4.23	Most common dependency types in the SGA system	78
5.1	Absences thresholds	91
5.2	Divergences thresholds	92
5.3	Architectural violations in the SGA system	92

Contents

Agradecimientos	ix
Resumo	xi
Abstract	xiii
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Problem	1
1.2 Thesis Statement	3
1.3 An Overview of the Proposed Approach	4
1.4 Outline of the Thesis	5
1.5 Publications	6
2 Background	9
2.1 Software Architecture	9
2.2 Architectural Erosion	10
2.3 Architectural Conformance Checking	10
2.3.1 Static Architecture Conformance	11
2.3.2 Critical Assessment	15
2.4 Detecting Source Code Anomalies	15
2.4.1 Structural Analysis Techniques	16
2.4.2 Historical Analysis Techniques	18
2.4.3 Static Analysis Techniques	19
2.4.4 Critical Assessment	20
2.5 Data Mining Techniques	20

2.5.1	Frequent Itemset Mining	21
2.5.2	Formal Concept Analysis	23
2.6	Final Remarks	24
3	Heuristics for Detecting Architectural Violations	29
3.1	Overview	29
3.2	Heuristic for Detecting Absences	30
3.3	Heuristics for Detecting Divergences	32
3.3.1	Heuristic #1	33
3.3.2	Heuristic #2	35
3.3.3	Heuristic #3	36
3.4	Formal Definition	37
3.4.1	Notation	37
3.4.2	Detecting Absences	38
3.4.3	Detecting Divergences	39
3.5	Ranking Strategy	42
3.6	Tool Support	43
3.7	A Heuristic-Based Architecture Conformance Process	44
3.8	Final Remarks	47
4	Evaluation	49
4.1	First Study: SGA System	49
4.1.1	Study Setup	49
4.1.2	Results	52
4.1.3	Comparison with Reflexion Models	58
4.1.4	Historical Analysis	60
4.2	Second Study: M2M System	62
4.2.1	Study Setup	62
4.2.2	Results for the M2M system	64
4.2.3	M2M Conformance Process	65
4.3	Third Study: Lucene System	67
4.3.1	Study Setup	67
4.3.2	Results for the Lucene system	68
4.3.3	Lucene Conformance Process	69
4.4	Fourth Study: ArgoUML System	71
4.4.1	Study Setup	71
4.4.2	Results for the ArgoUML system	72

4.4.3	ArgoUML Conformance Process	72
4.5	Discussion	74
4.6	Threats to Validity	78
4.7	Final Remarks	79
5	Extracting Architectural Patterns	81
5.1	Motivation	81
5.2	Data Mining Based Approach	83
5.2.1	Mining for Absences	85
5.2.2	Mining for Divergences	88
5.3	Evaluation	90
5.3.1	Dataset	90
5.3.2	Thresholds for Absences	91
5.3.3	Thresholds for Divergences	91
5.3.4	Results	92
5.4	Discussion	92
5.5	Final Remarks	94
6	Conclusion	97
6.1	Summary	97
6.2	Contributions	98
6.3	Limitations	99
6.4	Further Work	99
	Bibliography	101

Chapter 1

Introduction

In this initial chapter, we state the problem and present this thesis motivation (Section 1.1). Next, we present an overview of our approach to tackle the proposed problem (Section 1.3). Finally, we present the outline of the thesis (Section 1.4) and the publications derived from our research (Section 1.5).

1.1 Problem

The definition of a well-designed software architectural model plays an important role in modern software quality control tasks because important internal quality properties, such as maintainability and evolvability, directly depend on it. There are many definitions of software architecture. Typically, software architecture is defined as including the central components of a software system and their interconnections [Clements, 2003]. Bass et al. define software architecture as *“the structure or structures of the system, which comprise software elements, the externally visible properties of these elements, and the relationship among them”* [Bass et al., 2003]. Therefore, the architecture of a system prescribes the structure of its components, their relationships, constraints, principles, and guidelines that control its design and evolution over time [Garlan, 2000, Garlan and Shaw, 1996, Fowler, 2002]. An architectural model is a high-level software representation that supports documenting and communicating key design decisions and principles adopted by a software development team.

However, even well-designed software architectures can degenerate during the system evolution due to the introduction of implementation anomalies that correspond to mismatches between the concrete architecture, as implemented in source code, and the planned/intended architecture prescribed by the software architects. In

the more extreme cases, the introduction of architectural anomalies can trigger a major software reengineering effort or even the discontinuation of a software system [Hochstein and Lindvall, 2005].

Architectural conformance checking is a fundamental activity for controlling the quality of software systems, which aims to reveal deviations between the actual and planned software architectures [Passos et al., 2010]. More precisely, the goal is to detect the implementation decisions followed by the source code that are not in conformance with the restrictions prescribed by the planned architecture. The periodical application of architectural conformance checking aims to prevent the accumulation of incorrect or inadequate implementation decisions and thus to avoid the phenomena known as architectural drift or erosion [Perry and Wolf, 1992].

Architectural deviations pose a serious threat to the long term survival of software systems [Hochstein and Lindvall, 2005, Garcia et al., 2009]. The reason is that the accumulation of architectural violations in the source code may demand extra effort even when dealing with simple code changes. For instance, Knodel et al. applied an architectural conformance checking technique in 15 products from a software product line, called Testo, targeting climate and flue gas measurement devices [Knodel et al., 2008]. As a result, they identified more than 6,000 architectural anomalies in these products. Additionally, Terra and Valente detected, by using architectural conformance checking techniques, 2,241 architectural anomalies in a human resource management system, called SGP [Terra and Valente, 2009]. They report that to fix these anomalies more than 100 hours were necessary. Furthermore, Sarkar et al. report an experience on modularizing an application whose size increased from 2.5 to 25 MLOC. According to the authors, the system’s architecture eroded to a single monolithic block. They report that the refactoring and reconstruction of the system, to restore its original architecture, required nearly two years—approximately 520 person-days for design and 2,100 person-days for programming and testing activities.

Currently, there are two major techniques for architectural conformance checking: reflexion models and domain-specific languages [Ducasse and Pollet, 2009, Passos et al., 2010]. Reflexion models compare a high-level model, manually created by an architect, with the implemented (or concrete) model, automatically extracted from the source code [Murphy et al., 1995, Knodel et al., 2006]. Basically, reflexion models can detect two types of architectural anomalies: absences and divergences. An absence occurs when a dependency defined by the high-level model is not present in the implemented one, i.e., it does not exist in the source code. A divergence occurs when there is a dependency on the source code that is not prescribed by the high-level

model. Finally, domain-specific languages allow architects to express in a simple syntax the constraints defined by the planned architectural model [Terra and Valente, 2009, Eichberg et al., 2008, Mens et al., 2006].

However, the application of current techniques for architectural conformance checking requires a considerable effort [Knodel et al., 2008, Terra and Valente, 2009]. On one hand, reflexion models usually require successive refinements of the high-level model in order to adequately express the full spectrum of absences and divergences that may be present in complex software systems. On the other hand, domain-specific languages require a detailed definition of constraints between the classes of a system. In all cases, the existing techniques for identifying architectural problems heavily depend on the availability of coherent architectural documents [Eichberg et al., 2008]. However, in most cases, the planned architecture is not formally documented or up-to-date.

1.2 Thesis Statement

Our thesis statement is as follows:

Architectural conformance checking is a fundamental activity for controlling the quality of software systems. However, the state-of-the-art architectural conformance checking techniques usually require a considerable effort to prescribe the architectural documents, as a list of constraints or as a high-level model. Therefore, the practice of software architecture conformance may benefit from a technique that combines static and historical analysis and that does not require the definition of constraints or model refinements.

Therefore, the main goal of this thesis is to propose and evaluate an approach that combines static and historical source code analysis techniques to provide an alternative technique for architecture conformance checking.

To attend this goal we plan to:

- Propose a technique for architecture conformance based on information gathered from mining software repository.
- Evaluate the precision of this technique in a real setting, using both open-source and closed systems.
- Conduct an exploratory study for evaluating the use of data mining techniques for detecting architectural violations.

1.3 An Overview of the Proposed Approach

As stated in the previous section, the application of the current techniques for architecture conformance checking is a nontrivial task and may require a considerable effort [Knodel et al., 2008, Passos et al., 2010]. To tackle these issues, this thesis proposes an approach that combines static and historical source code analysis techniques to provide an alternative technique for architecture conformance. Figure 1.1 illustrates the proposed approach for detecting architectural erosion symptoms. As can be observed, the approach relies on two types of input on the target system: (a) history of versions; and (b) high-level component specification. Basically, this component model includes information on the names of the components and a mapping from modules to component names, using regular expressions. Using these inputs, we propose heuristics to identify suspicious dependencies, or lack of, in source code by relying on frequency hypotheses and past corrections made on the code.

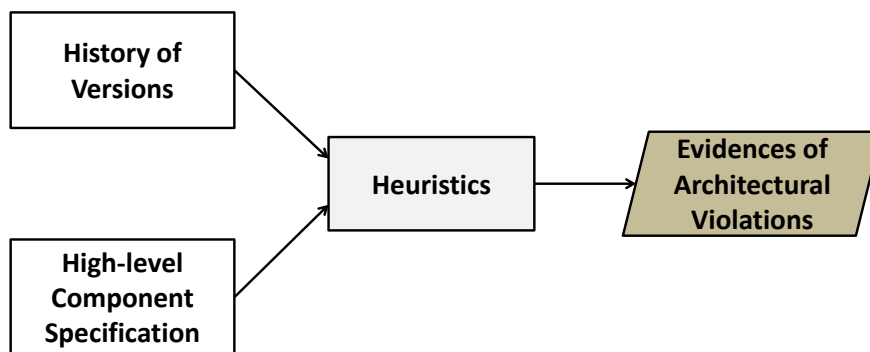


Figure 1.1. Proposed approach to architectural conformance checking

The proposed approach includes four heuristics to discover suspicious dependencies in the source code, including dependencies that may denote divergences (existing unwanted dependencies) or absences (missing expected dependencies). 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 systems (according to other thresholds). We also propose an iterative architecture conformance process, based on the defined heuristics. As proposed by this process, architects should experiment and adjust the thresholds required by the defined heuristics, starting with rigid thresholds. Usually, 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 firstly the warnings that are more likely to denote real violations. Finally, we implemented a prototype tool, called ArchLint, that supports the four heuristics for detecting architectural violations.

The proposed approach is (to the best of our knowledge) the first architecture conformance technique that relies on a combination of static and historical source code analysis. It does not require successive refinements on high-level architectural models neither the specification of an extensive list of architectural constraints, as required by domain-specific languages. However, the proposed heuristics can generate false positive warnings, as common in most bugs finding tools based on static analysis, such as FindBugs [Hovemeyer and Pugh, 2004] and PMD [Copeland, 2005].

We report the results of applying the proposed conformance checking process in four real-world systems. First, we applied this process in two industrial-strength information systems. The warnings generated by the proposed heuristics were evaluated by experts in these systems' architecture, who classified them as true or false positives. 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 with such violations, according to the systems' architects. Finally, we relied on the proposed conformance process to evaluate the architecture of two well-known open-source system, Lucene¹ and ArgoUML². In these cases, using as oracle a reflexion model independently proposed in another research [Bittencourt, 2012], we found 264 architectural violations in Lucene and 81 violations in ArgoUML, with an overall precision of 59.2% and 53.3% respectively.

1.4 Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 covers background work related to our research, including work on software architecture, architectural erosion, architectural conformance checking techniques, automatic anomaly detection in source code, and data mining techniques.
- Chapter 3 presents the proposed architectural conformance checking technique, including the description of the proposed heuristics to detect absences and

¹<http://lucene.apache.org>

²<http://argouml.tigris.org>

divergences, their formal definition, a strategy to rank violations, the design of the ArchLint tool, and a heuristic-based architecture conformance process.

- Chapter 4 evaluates our approach by presenting and discussing results on its usage in four real-world systems. In this chapter, we also summarize our main findings and the lessons learned after designing and evaluating the heuristics-based approach.
- Chapter 5 reports an exploratory study on mining architectural patterns using data mining techniques. Essentially, our goal with this final study is to investigate whether architectural patterns can be inferred by mining software repositories.
- Chapter 6 presents the final considerations of this thesis, including contributions, limitations, and future work.

1.5 Publications

This thesis generated the following publications and contains material from them:

- [Maffort et al., 2014]: Maffort, Cristiano; Valente, Marco Tulio; Terra, Ricardo; Bigonha, Mariza; Anquetil, Nicolas; Hora, Andre. *Mining Architectural Violations from Version History*. In *Empirical Software Engineering Journal (EMSE)*, p. 1–41, 2014. Invited for a special issue with best papers from WCRE 2013.
- [Maffort et al., 2013a]: Maffort, Cristiano; Valente, Marco Tulio; Anquetil, Nicolas; Hora, Andre; Bigonha, Mariza. *Heuristics for Discovering Architectural Violations*. In *20th Working Conference on Reverse Engineering (WCRE)*, p. 222–231, 2013.
- [Maffort et al., 2013b]: Maffort, Cristiano; Valente, Marco Tulio; Bigonha, Mariza; Anquetil, Nicolas; Hora, Andre. *Mining Architectural Patterns Using Association Rules*. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 375–380, 2013.
- [Maffort et al., 2013c]: Maffort, Cristiano; Valente, Marco Tulio; Bigonha, Mariza; Silva, Leonardo Humberto; Aparecido, Gladston. *ArchLint: Uma Ferramenta para Detecção de Violações Arquiteturais usando Histórico de Versões*. In *IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas)*, pages 1–6, 2013.

- [Maffort et al., 2012]: Maffort, Cristiano; Valente, Marco Tulio; Bigonha, Mariza. *Detecção de Violações Arquiteturais usando Histórico de Versões*. In *XI Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15, 2012.

Additionally, the software infrastructure proposed to extract structural dependencies from software repositories is used in the following papers:

- [Rocha et al., 2013]: Rocha, Henrique; Couto, Cesar; Maffort, Cristiano; Garcia, Rogel; Simões, Clarisse; Passos, Leonardo; Valente, Marco Tulio. *Mining the Impact of Evolution Categories on Object-Oriented Metrics*. In *Software Quality Journal*, vol. 21, issue 4, pages 529–549, 2013.
- [Couto et al., 2013]: Couto, Cesar; Maffort, Cristiano; Garcia, Rogel; Valente, Marco Tulio. *COMETS: A Dataset for Empirical Research on Software Evolution using Source Code Metrics and Time Series Analysis*. In *ACM SIGSOFT Software Engineering Notes*, pages 1–3, 2013.

Chapter 2

Background

This chapter is organized as follows. In Sections 2.1 and 2.2 we introduce fundamental concepts on software architecture and software architectural erosion. Section 2.3 presents methodologies and techniques for architectural conformance. Section 2.4 describes some techniques for automatic detection of source code anomalies. Finally, in Section 2.5, we present data mining techniques that can be used to detect architectural violations.

2.1 Software Architecture

The most common definition for software architecture follows a structural perspective, which considers that an architecture is composed by elements and their interconnections [Clements, 2003]. For example, Bass et al. define software architecture as the structure of a system, which comprises software components, externally visible properties of those components, and the relationships between them [Bass et al., 2003]. Moreover, the architecture of a software often includes the relationships, constraints, principles, and guidelines that should guide its design and evolution over time [Garlan, 2000, Garlan and Shaw, 1996, Fowler, 2002]. *Components* are usually defined as architectural entities which encapsulate a subset of the system's functionalities [Gurgel et al., 2014].

Software architecture decisions typically have a long-term impact on individual aspects of the construction and evolution of software systems [Pressman, 2010]. Therefore, software architectures must be documented, not only for purposes of analyzing the system, but also as an artifact for communication with stakeholders, facilitating the understanding of a system [Bittencourt, 2010]

2.2 Architectural Erosion

It is unquestionable that the architecture specification of a system is a crucial development activity, as described in Section 2.1. It is an activity that comes before implementation activities, when software architects define the modular decomposition of the system and the dependency relationships and constraints between modules and components, as well as relations with external systems, such as libraries frameworks.

However, during the evolution of a software product implementation anomalies are frequently introduced in the source code, i.e., decisions that are not compatible with the specified architectural model, making the current codebase inconsistent with the existing documentation [Kazman and Carrière, 1999, Knodel et al., 2006, Knodel and Popescu, 2007, Murphy et al., 1995, Murphy et al., 2001a, Schmerl et al., 2006]. In this thesis, these anomalies are called architectural violations [Passos et al., 2010, Perry and Wolf, 1992].

In practice, the introduction of architectural violations is common, mainly due to the developers' lack of knowledge, deadline pressures, technical difficulties, conflicting requirements, etc. [Knodel and Popescu, 2007]. As a result, these violations make maintenance a more difficult and time-consuming task, since the implemented product is not adherent to the planned and documented architecture [Sarkar et al., 2009a]. Additionally, the progressive introduction of architectural violations in the code can make even simple software evolution tasks more difficult [Macia et al., 2012]. Nonetheless, architectural violations usually remain in the source code, leading to the phenomena known as architectural deviations or architectural erosion [Perry and Wolf, 1992]. Basically, erosion is an architectural deviation and occurs when the rules governing the dependencies between architecture components are violated [Perry and Wolf, 1992]. In the more critical cases, architectural problems can lead to a full reengineering or even the discontinuation of a software products [Hochstein and Lindvall, 2005].

2.3 Architectural Conformance Checking

To avoid architectural erosion, many architectural conformance approaches were proposed [Bass et al., 2003, Gorton and Zhu, 2005]. Basically, this activity consists in checking whether a particular version of the system adheres to the planned architecture [van Gurp and Bosch, 2002]. In other words, architectural conformance checking can be viewed as a measure of the degree of adherence between

the concrete architecture, as implemented in source code, and the planned architecture [Knodel and Popescu, 2007].

Architectural conformance approaches rely on static or dynamic analysis techniques [Bell, 1999, Hamou-Lhadj and Lethbridge, 2004, Jerding and Rugaber, 1997, Schmerl et al., 2006]. Static analysis techniques are non-invasive, and depend only on the source code. For this reason, they do not impact the normal programming activities or cause any impact during a system execution. On the other hand, techniques based on dynamic analysis are performed during the execution of the system. Therefore, they can deal with systems whose behavior may change at runtime, such as systems that are implemented using techniques such as dependency injection, reflexion, and meta-programming [Brito et al., 2013]. In this thesis, we focus on techniques based on static analysis, because they are the most established ones.

2.3.1 Static Architecture Conformance

In this section we discuss some well-know architectural conformance approaches based on static analysis.

Dependency Structure Matrix (DSM)

Dependency structure matrices were proposed by Baldwin and Clark to demonstrate and assess the importance of the modular organization of software projects [Baldwin and Clark, 1999, Sullivan et al., 2001]. Essentially, DSMs are adjacency matrices that represent dependencies between the modules of a system. The elements in these matrices indicate the existence of static dependencies between the element of the column (source dependency) and the element of the row (target dependency). For example, in Figure 2.1, the X in the cell (1,2) denotes that class *B* depends on class *A*. In other words, class *B* has explicit references (method calls, parameters, exceptions, etc) to syntactic elements defined by class *A*.

		1	2	3
Class A	1	.	X	
Class B	2		.	
Class C	3			.

Figure 2.1. DSM example (class *B* depends on class *A*)

There are many tools that generate DSMs, such as the *Lattix Dependency Manager* (LDM) tool [Sangal et al., 2005].¹ This tool is also able to perform architectural conformance checking. It provides a graphical interface that can be used to reveal architectural patterns and detect dependencies that may indicate architectural violations. Initially, the architectural conformance activity requires the DSM extraction using static analysis techniques. In a second step, a declarative language is used to specify the conformance rules that must be followed by the source code of the system under evaluation. Basically, the domain specific language supported by the LDM tool is very simple. The rules for conformance checking have two forms: **A can-use B** and **A cannot-use B**, which are used to indicate that a particular class A may or may not depend on a given class B.

Source Code Query Language

Oege Moor et al. proposed a source code query language called .QL, which has a syntax similar to the SQL language [de Moor, 2007]. By using this language, it is possible to perform many activities to support software development, such as architectural conformance checking, searching for errors, software metrics calculation, identification of refactoring opportunities, etc. To illustrate, we will use a .QL query to check architectural conformance of a system implemented following the MVC architectural pattern². In systems implemented accordingly to this pattern, the **Model** layer cannot have any dependency with the **Controller** layer. To verify this constraint, the following query can be defined in .QL:

```

1:  FROM RefType r1, RefType r2
2:  WHERE r1.fromSource()
3:    AND depends(r1, r2)
4:    AND isModelLayer(r1)
5:    AND isControllerLayer(r2)
6:  SELECT "Warning: " + r1.getQualifiedName() +
        " depends on " + r2.getQualifiedName()

```

Basically, this query selects all classes of the layer **Model** that have dependencies with classes of the **Controller** layer. As can be observed, the class *r1* must be implemented in the source code (line 2). Specifically, *RefType* is a .QL built-in type

¹<http://www.lattix.com>

²A similar example is shown in [Passos et al., 2010]

that provides information about a particular type of a Java program. The predicates *isModelLayer* (line 4) and *isControllerLayer* (line 5) check whether a reference belongs to the `Model` and `Controller` layers, respectively.

Reflexion Models

The Reflexion Models (RM) technique was proposed by Murphy et al. [Murphy et al., 1995, Murphy et al., 2001a]. According to this technique, software architects should first define a high-level model representing the planned or desired architecture. They should also define the dependencies between the components prescribed in this high-level model. Moreover, architects must define a mapping between the concrete architecture (source code model) and the proposed high-level model (desired architecture).

Knodel et al. describe a tool called *Software Architecture Visualization and Evaluation* (SAVE) [Knodel et al., 2006] for architectural conformance checking based on reflexion models. Using as input the high-level model and a mapping of this model to the source code of the system under analysis, the SAVE tool produces the reflexion model for revealing architectural violations in the source code.

A Reflexion Model classifies the dependencies between the classes of a system as follows:

- Convergence: when a dependency prescribed in the architectural model exists in the source code.
- Divergence: when a dependency exists in the source code, but it is not prescribed by the architectural model.
- Absence: when a dependency does not exist in the source code, but it is prescribed by the architectural model.

A RM-based tool, such as SAVE, highlights the divergence and absence dependencies in the high-level model initially provided by the architects.

Dependency Constraint Languages

These solutions include domain-specific languages to detect dependencies that are allowed and not allowed in the code, which are inferred from declarative structural constraints between modules. As an example, we can mention DCL (*Dependency Constraint Language*) [Terra and Valente, 2009, Terra and Valente, 2008].

For using such languages, a software architect must first define the dependency constraints between the classes of the system under analysis. For example, using DCL, architects may define acceptable or unacceptable dependencies, according to the desired system architecture. The DCL specification presented next contains four dependency constraints (lines 5-8) for a hypothetical system that adopts the MVC architectural pattern.

```
1: module Model:      com.myapp.*.model.**
2: module View:       com.myapp.*.view.**
3: module Controller: com.myapp.*.control.**
4: module JPA:        javax.persistence.**

5: Model can-depend-only Model, JPA, $java$
6: only View can-depend Controller
7: View cannot-depend JPA
8: Controller cannot-depend JPA
```

Initially, this specification defines the modules of each layer in the MVC pattern (lines 1-3). Next, it defines that the JPA module is composed by classes of the package `javax.persistence`, including its subpackages (line 4). Finally, a sequence of dependency constraints are defined (lines 5-8). More specifically, the constraint on line 5 defines that classes in the `Model` layer can only depend on classes in this layer, on JPA classes, and on the Java API. Line 6 establishes that only the `View` layer can depend on classes in the `Controller` layer. Finally, the `View` and `Controller` layers cannot depend on JPA classes (lines 7-8).

The proposed approach also provides a plug-in for the Eclipse IDE called `dclcheck`. This plug-in checks whether the source code is in accordance with the constraints defined in DCL.

Gurgel et al. [Gurgel et al., 2014] proposed a conformance technique that, similarly to the DCL language, provides mechanisms to explicitly define the planned architecture of a system, describing their components and dependency constraints. Their approach assumes that similar degradation evidences occur in software projects. For this reason, the approach includes support for the hierarchical and compositional reuse of rules, providing specification mechanisms to specialize previously defined rules.

Constraint Programming Languages

Constraint programming languages, usually based on first-order logic, allow software architects to express architectural constraints on the static structure of object-oriented systems. The restrictions are specified by a sequence of statements and logical declarations. However, this definition might be a complex and error-prone activity, especially for architects and maintainers with experience only in imperative languages. As examples of logic-based constraint languages, we highlight SCL (*Structural Constraint Language*) [Hou and Hoover, 2006], FCL (*Framework Constraint Language*) [Hou et al., 2004], and LogEn [Eichberg et al., 2008]. To support a less complex and more comprehensive notation for expressing architectural constraints, LogEn authors have proposed a graphical notation, called VisEn, from which LogEn constraints can be automatically generated.

Architectural Description Languages (ADLs)

ADLs enable architectural conformance checking by constructing and expressing the architectural behavior and the structure of a software system in a declarative and abstract language [Allen and Garlan, 1997, Garlan et al., 1997, Magee et al., 1995]. From an ADL specification, code generation tools are proposed to transform architectural descriptions to code in a general-purpose language.

2.3.2 Critical Assessment

As reported in this section, architectural conformance checking approaches based on static analysis—such as dependency structural matrices (DSM), source code query languages, and reflexion models—adopt non-invasive strategies, but also require a detailed architectural specification (to prescribe modules, components, dependency constraints, etc.). However, these specifications have shortcomings, making difficult to prescribe some constraints or architectural representation. Furthermore, they often require the definition of many rules or similar programming artifacts that should be maintained and evolved, which are also subjected to errors and omissions.

2.4 Detecting Source Code Anomalies

Programs frequently follow informal or implicit programming conventions [Li and Zhou, 2005, Gruska et al., 2010, Chang et al., 2007,

Wasylkowski et al., 2007]. For example, some method calls usually occur in a given order, such as an `unlock` call, which normally follows a `lock` call. Other programming rules may also prescribe more detailed dependencies involving more functions, as well as other elements such as variables and data types.

On the other hand, developers often unconsciously violate these programming rules during their daily programming activities. As a result, they can, for example, add bugs when they do not follow such rules. Finally, manually providing a specification for each of such rules is not exactly a simple task [Mileva et al., 2011, Wasylkowski and Zeller, 2009].

The techniques for checking programming patterns, discussed in this section, do not assume any prior knowledge on the systems under evaluation, such as naming conventions or pre-defined programming standards, as required by the static architecture conformance checking approaches described in Section 2.3. Essentially, the approaches discussed in the present section are based on the observation that large systems adopt patterns in their implementation and that deviations from these patterns can therefore be considered as anomalies [Engler et al., 2001a].

In the remainder of this section, we present techniques for detecting programming patterns in an automate way. Initially, we present techniques for inferring such patterns from structural information extracted from source code (Section 2.4.1). Next, we present techniques based on usage and non-usage patterns extracted from different versions of a system (Section 2.4.2). Finally, we present techniques based on static information extracted from source code (Section 2.4.3).

2.4.1 Structural Analysis Techniques

PR-Miner is a tool that extracts programming rules from systems implemented in C, without requiring any previous knowledge on the software under evaluation or any form of instrumentation (e.g., insertion of annotations) [Li and Zhou, 2005]. This tool uses a data mining technique called *frequent itemset mining* to detect patterns from the extracted rules. More details on *frequent itemset mining* are presented in Section 2.5. Finally, PR-Miner searches for programming decisions in the source code that are not adherent to the programming patterns previously identified. Such divergent patterns are presented as evidences of bugs. The PR-Miner approach is based on the assumption that correct programming rules are frequently followed and violations rarely occur. To reinforce this assumption, the tool selects only programming rules that are similar to the detected patterns in at least 90% of the cases.

Wasylkowski and colleagues [Wasylkowski et al., 2007] proposed an approach

for detecting programming anomalies based on sequences of interdependent method calls, such as calls to method `Stack.push()`, which usually occur before calls to method `Stack.elements()`. This sequence is represented as an *object usage model*—which models typical object usage as possible sequences of method calls. The extracted patterns are compared with their instances in the source code to identify implementation decisions that violate the proposed patterns, which are classified as evidence of defects.

The proposed approach includes a tool called JADET that detects patterns in the form of method calls sequences. The tool also searches for sequences that are not in conformance with these patterns. The analysis of similarity among patterns performed on temporal properties extracted from the source code, using formal concept analysis algorithms.

Wasylkowski et al. proposed an approach for mining object usage models, extracted by the JADET tool, which checks whether certain preconditions are satisfied before method invocations [Wasylkowski and Zeller, 2009]. Specifically, the authors introduced the concept of *operational preconditions*, which establish how to satisfy the preconditions of a function or method. In the proposed approach, these preconditions are extracted from the source code using a tool called Tikanga. The ultimate goal is to discover the operational preconditions from the context that precedes particular function or method calls. The higher the number of calls of a particular method, the higher the precision of inferring its operational preconditions and also the precision in selecting method calls whose preconditions are not adequately satisfied.

The existing techniques for detecting violations using structural analysis extracts information only from the source code. In all cases, the precision of the results depends on the occurrence of patterns with structural similarity in the application under analysis. Therefore, if the expected implementation pattern rarely occurs in the system under analysis, it will be wrongly taken as a violation and not as a programming anomaly.

To overcome the weaknesses of the aforementioned techniques, Gruska et al. proposed an approach, also based on the JADET tool, which retrieves temporal properties on the methods of the application under analysis [Gruska et al., 2010]. The detection is also based on formal concept analysis, as implemented by the Colibri-Java [Götzmann, 2007] tool. However, the patterns are extracted from various systems whose implementation is admittedly correct and that are supposed to present high-levels of internal software quality. Moreover, the extracted temporal properties are not confronted with the patterns of the application itself. Instead, they are compared with patterns extracted from other systems, whose structural quality and correctness

is also recognized.

2.4.2 Historical Analysis Techniques

During the evolution of a software, changes frequently occur. For example, programming anomalies are introduced, defects are corrected, source code is refactored, etc. In other words, in high-quality and well-organized systems, some of these anomalies are detected and corrected as a result of software maintenance or inspection activities.

Architectural conformance techniques based on historical analysis searches for source code change patterns. Next, these techniques detect in a specific version of the system under evaluation (usually the current version), dependencies that are not in conformance with these evolution patterns.

Zimmermann et al. proposed one of the first approaches for detecting code anomalies using version history [Zimmermann et al., 2004, Zimmermann et al., 2005]. The proposed approach includes a tool, called ROSE, which relies on association rules extracted from version control history systems to suggest and predict missing changes in the artifacts of the system under analysis. Moreover, ROSE also identifies anomalies that occur outside the system's source code, like the need to update the documentation after changes in the source code. The overall precision achieved by ROSE was 40% in an evaluation with eight open-source systems. Considering only the first three warnings indicated in the list, ROSE achieved a precision of 90%.

Another event that often occurs during the evolution of software systems is the need to perform modifications in the source code as a result of updates in APIs. In this case, it is necessary to verify whether the code was updated consistently. To tackle this problem, Mileva et al. proposed a methodology, supported by a tool called LAMARCK, that extracts temporal properties regarding two different versions of a system and analyzes the changes between these versions to infer evolution patterns [Mileva et al., 2011]. Such patterns can be shared with developers to prevent the use of incorrect programming strategies. Furthermore, they can help to identify implementation decisions that are out of date. To derive the evolutionary patterns and to reveal the programming violations, LAMARCK relies on formal concept analysis techniques, as supported by the Java-Colibri tool.

Silva et al. proposed a technique to assess package modularity using co-change clusters [Silva et al., 2014], based on the assumption that programming decisions that are likely to change together should be implemented in the same module. In their work, the authors argue that the traditional package hierarchy suffers from the dominant decomposition problem and their approach can introduce improvements in

understanding whether a system is really well-modularized. Santos et al. describe a remodularization technique that relies on information retrieval and semantic clustering analysis over a vocabulary extracted from identifiers and comments presented in the classes of a system under evaluation [Santos et al., 2014]. This approach can also be used to suggest a more suitable modular organization for the evaluated system.

The precision results achieved by the discussed historical analysis approaches depend essentially on identifying usage and non-usage patterns that are then contrasted with the remainder of the application. In other words, precision depends on the number and the correctness of frequent items in the source code. Moreover, it also depends on the number of revisions in the repository.

2.4.3 Static Analysis Techniques

Techniques for detecting defects using static analysis are based on idioms representing programming defects, such as: division by zero, array indexing beyond its limits, method calls using *null* references, etc. [Araujo et al., 2011]. Several tools have been proposed to support static code analysis, which generally aim to extend and enhance the warning messages generated by compilers [Couto et al., 2012].

Lint [Johnson, 1977, Darwin, 1988] was one of the first tools to support static analysis for identifying bugs and programming bad smells [Hovemeyer and Pugh, 2004, Fowler, 1999]. The tool searches for common errors present in source code, as well as it aims to reinforce some common rules of the C language—such as type checking, operations *and/or*, and portability restrictions. As a result, some of the checks performed by Lint were later integrated into compilers, such as checking the use of uninitialized variables. LCLint [Evans et al., 1994, Evans, 1996] and JLint [Artho and Biere, 2001] are examples of static source code checking tools that inherited the philosophy originally proposed by Lint.

Among the existing static verification tools for Java, FindBugs [Hovemeyer and Pugh, 2004] and PMD [Copeland, 2005] are among the most popular. FindBugs is an open-source tool that implements a set of bug detectors able to point out more than 360 patterns of bugs. These patterns are classified into categories such as correctness, performance, threads synchronization, malicious code, bad practice, etc. Moreover, FindBugs classifies the bug patterns as having high, medium, or low priority.

PMD is an open source tool that supports an extensive set of rules for detecting potential bugs and to check coding styles. PMD also offers a set of metrics to detect violations in recommended programming practices, such as excessive number

of attributes or long methods. For the detection of bugs, PMD computes its rules over the AST (Abstract Syntax Tree) generated from the source code of the system under analysis, unlike FindBugs, which works at the bytecode level.

In a study performed by Araujo et al., the highest precision achieved by FindBugs was 52.5% [Araujo et al., 2011]. To achieve such result, the authors configured FindBugs to report only bug evidences with high priority from the correctness category. In the same study, the authors highlight that PMD produces a massive number of bug evidences, which must be manually inspected. More specifically, the best result obtained by the PMD tool was 10% of precision.

2.4.4 Critical Assessment

Among the approaches for automatic detection of source code anomalies described in this section, the ones based on structural analysis rely on formalisms to extract dependencies that are strongly linked to procedural languages. These approaches consider only function calls, independently from the modular and/or architectural context where they occur. Similarly, approaches based on historical analysis do not consider violations that occur at the architectural level, since they also consider only concepts of procedural languages. Furthermore, they often evaluate a limited historical context, since the analysis is typically performed with only two versions. Approaches based on static analysis consider only the language idioms, such as the use of types and typical objects of the language. Typically, such techniques do not consider modular or architectural aspects of the system under analysis.

2.5 Data Mining Techniques

Data mining includes a set of techniques for the analysis and extraction of information potentially useful, implicit, and previously unknown [Tan et al., 2002, Frawley et al., 1992, Fayyad et al., 1996]. Basically, data mining techniques perform a search for patterns in a dataset defined according to the chosen data mining algorithm. In the context of this thesis, two data mining techniques deserve special attention due to their potential application to architectural violation detection. Section 2.5.1 describes the *frequent itemset mining* technique and Section 2.5.2 describes the technique based on *formal concepts analysis*.

2.5.1 Frequent Itemset Mining

Frequent patterns are those that appear repeatedly in a dataset. For example, a pair of items, like “bread” and “butter”, which usually appear together in database transactions, are considered a frequent pattern. As an example in the software engineering context, calling an *open()* method and then calling a *close()* method, may also represent a frequent pattern if it occurs at various points in a program. As another example, considering the software architecture context, classes that depend on *Entity* annotation, with an expressive confidence, also depend on annotation *Id*.

The following concepts are fundamental on frequent itemset mining:

- **Items:** are the objects under study to which we want to discover the sets of co-occurring values or frequent patterns. For instance, the objects in a market basket can be considered items.
- **Itemset:** is a set of items. For instance, a given collection of objects in a market basket can be considered an itemset.
- **Transaction:** is a well-defined itemset, represented as a tuple $\langle t, X \rangle$, where t is a unique transaction identifier.
- **Database:** is a collection of transactions. Figure 2.2 shows these forms of database representations.

	A	B	C	D	E
1	1	1	0	0	1
2	1	1	0	1	1
3	1	0	1	0	0
4	1	1	0	0	1
5	0	1	0	1	0
6	1	1	1	1	1

(a) Binary

	itemset
1	ABE
2	ABDE
3	AC
4	ABE
5	BD
6	ABCDE

(b) Transaction

A	B	C	D	E
1	1	3	2	1
2	2	6	5	2
3	4		6	4
4	5			6
6	6			

(c) Vertical

Figure 2.2. Database representations

Support is a relevant measure in frequent itemset mining techniques, defined as the number of transactions that contain a given itemset, i.e., it is a measure of popularity of the pattern in the database. Objectively, a pattern is frequent if it has a support greater than a given threshold. Therefore, given a database of transactions and a minimum support, a frequent itemset mining algorithm must enumerate all itemsets that are frequent [Zaki and Meira Jr., 2011].

As an example, assuming *minimum-support* = 3. The frequent sub-itemsets of the transactions presented in Table 2.2 are A : 5, B : 5, D : 3, E : 4, AB : 4, AE : 4, and ABE : 3. The other itemsets are not frequent because their support is lower than 3.

Once frequent itemsets are mined, it is possible to extract *association rules* from these sets and to make assumptions on how often two sets of items occur simultaneously or conditionally [Agrawal et al., 1993]. Furthermore, association rules can be used to discover relationships between objects. For example, given a sales database in a supermarket, it is possible to generate rules of the following type: if consumers purchase a product A , there is a good chance that they will also purchase product B . In this case, product A is called the antecedent term and product B is called the consequent term of the association rule.

An association rule is formally represented as $\mathcal{X} \Rightarrow \mathcal{Y}$, where \mathcal{X} and \mathcal{Y} are itemsets and $\mathcal{X} \cap \mathcal{Y} = \emptyset$. Each association rule has a *support* and a *confidence* measure. The support is the number of times in which both \mathcal{X} and \mathcal{Y} simultaneously occur as subsets in the same transaction. The confidence represents the probability of a transaction covered by an antecedent term \mathcal{X} be also covered by a consequent term \mathcal{Y} . In practice, the confidence of a rule is calculated as its support divided by the support of the antecedent term.

Additionally, other measures are used when extracting association rules, such as:

- **Lift**: is a measure of the surprise or the strength of a rule. It is computed by dividing the confidence of a rule by the relative support of its consequent term.
- **Leverage**: is a measure of the difference between the observed relative support of the consequent term and the expected joint probability of the multiplication between the relative support of the antecedent term and the consequent term.
- **Jaccard**: is a coefficient that measures the similarity between the antecedent term and the consequent term.

Finally, many algorithms can be used to compute itemsets and association rules, such as:

- **Apriori**: an algorithm that improves a brute-force approach for frequent itemset mining [Agrawal and Srikant, 1994].
- **Eclat**: an algorithm that indexes the database to improve the computation of frequency [Zaki et al., 1997].

- **FP-Growth:** an algorithm that uses pattern fragment growth to mine the complete set of frequent patterns [Han et al., 2000].

2.5.2 Formal Concept Analysis

Concept analysis is a branch of applied mathematics, particularly of the lattice theory [Birkhoff, 1940, Ganter and Wille, 1999, Wille, 2009, Davey and Priestley, 2002]. It is a technique for analyzing binary relations between arbitrary objects and their attributes. It produces as output a concepts lattice, which provides an understanding of the underlying structure of the dependencies between objects.

The *formal context*, whose definition is fundamental to the study of formal concept analysis, is defined by a triple (O, A, R) , where O is a set of objects, A is a set of attributes, and R is a binary relation, called incidence, so that $R \subseteq O \times A$. As an example, in information retrieval system, documents can be considered objects and their attributes are considered terms.

A suitable way for representing formal contexts is by means of a cross table, where the rows are objects and the columns are attributes. The incidence is shown in this table using a symbol to indicate whether there is any relationship between the object and the attribute. Therefore, formal contexts are used for representing sets whose objects may or may not have certain attributes. Figure 2.3 illustrates an example of a representation of a formal context using a cross table. As we can observe, the objects are animals that are famous in certain regions of the world and the attributes denote whether these animals are cartoons or real animals, as well as whether they are dogs, cats, mammals, or turtles.

	cartoon	real	tortoise	dog	cat	mammal
Garfield	X				X	X
Snoopy	X			X		X
Socks		X			X	X
Greyfriar's Bobby		X		X		X
Harriet		X	X			

Figure 2.3. Formal context of “famous animals” [Priss, 2006]

A formal context determines *formal concepts*. The set of objects (O) of a formal concept is called *extension* (E) and the set of attributes (A) is called *intention* (I) such that $E \subseteq O$, $I \subseteq A$. A hierarchically ordered set of all formal concepts of a formal context is called *lattice concepts*.

A suitable way to represent the lattices is by means of a graph whose vertices denote the formal concepts and whose edges denote their relationships. An edge creates a relationship of super-concept and sub-concept between connected formal concepts. The highest vertex of the diagram represents the formal concept whose extension contains all objects, while the lower vertex contains all attributes in its intention.

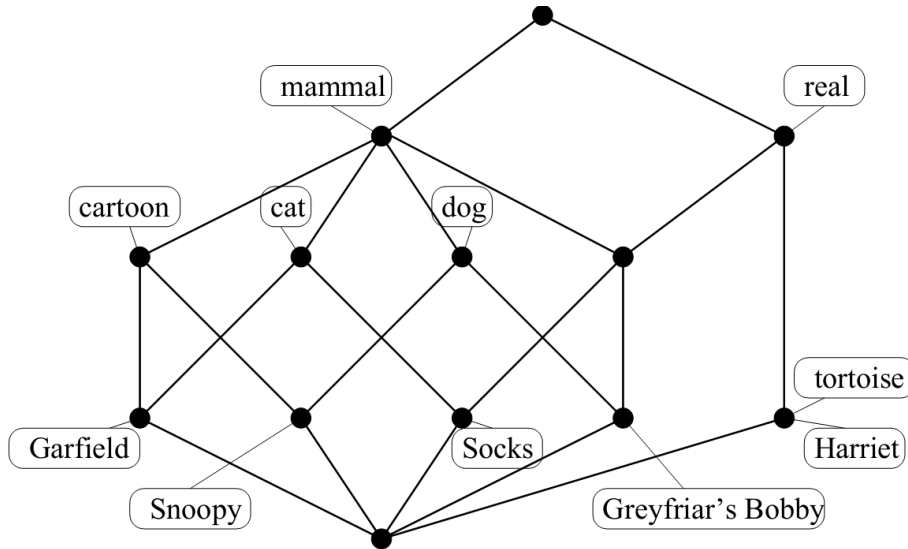


Figure 2.4. Concept lattice of Figure 2.3 [Priss, 2006]

Figure 2.4 shows a diagram of a concept lattice whose concepts correspond to the formal context of Figure 2.3. In this diagram, each vertex represents the formal concepts. The objects are arranged on the bottom of the diagram and the attributes are placed on top. Both objects and attributes are identified by their respective labels.

In summary, formal concept analysis provides a framework to understand and mine patterns between objects and attributes. From the identified patterns, this technique detect structural anomalies of the objects in relation to the attributes and vice versa. For this purpose, techniques based on formal concept analysis can group blocks of similar relationships, i.e., those that share common attributes.

2.6 Final Remarks

In software development, it is common that development teams adopt strategies that are not consonant with best practices, as defined by the architectural model of the system. As a result, several approaches have been proposed to detect source code anomalies, including architectural violations.

Table 2.1 summarize the techniques discussed before in this chapter. As can be observed, the current techniques for architectural conformance checking require as input an architectural representation, which is not trivial to be produced and that is also subject to errors and omissions. On the other side, the existing solutions for automatically detecting programming anomalies usually do not consider architectural violations. Finally, the approaches based on historical analysis restrict considerably the evaluated historical context, limiting the analysis on two versions of the system in most cases.

We concluded this chapter by discussing some data mining techniques that can contribute to an automatic analysis of architectural patterns. Both frequent itemset mining and formal concept analysis can be used to detect complex patterns of dependencies, composed by multiple classes, independently on any prior knowledge of software architecture or on the architectural scenarios where violations frequently occur. In other words, these techniques can detect any pattern of co-occurrence among items in a dataset. Particularly, frequent itemset mining techniques produce association rules that can be used: (i) as architectural patterns, allowing to detect dependencies that violates these patterns; and (ii) as documentation artifacts, supporting and guiding the development team on expliciting the dependencies in the system. Formal concept analysis can be a promising technique for mining architectural violations by highlighting dependency relationships between classes of a system, including use and disuse of relations.

Table 2.1. Techniques for detecting programming anomalies

Technique	Input	Tools	Intention	Critical Assessment
Dependency Structure Matrices	Bytecode of one version of a system	LDM [Sangal et al., 2005]	Architectural conformance checking	Require a detailed architectural specification and do not consider the history of version
Source Code Query Languages	Source code of one version of a system	.QL [de Moor, 2007]	Architectural conformance checking and identification of refactoring opportunities	
Reflexion Models	Source code of one version of a system and a high-level model of the desired architecture	SAVE [Knodel et al., 2006]	Architectural conformance checking	
Dependency Constraint Languages	Source code of one version of a system and a list of structural constraints between modules	DCL [Terra and Valente, 2009] and Tamdera [Gurgel et al., 2014]	Architectural conformance checking	

Table 2.1. Techniques for detecting programming anomalies

Technique	Input	Tools	Intention	Critical Assessment
Constraint Programming Languages	Source code of one version of a system and a list of architectural constraints on the static structure, defined in Prolog	SCL [Hou and Hoover, 2006], FCL [Hou et al., 2004], and LogEn [Eichberg et al., 2008]	Architectural conformance checking	Require a detailed architectural specification and do not consider the history of versions
Architectural Description Languages	Architectural description constraints specification	Acme [Garlan et al., 1994], ADL[Feiler, 2014], and Wright [Allen, 1997]	Architectural conformance checking	Require a detailed architectural specification
Structural Analysis Techniques	Source code of one version of a system	PR-Miner [Li and Zhou, 2005] and JADET [Wasykowski et al., 2007]	Detecting anomalies in programming patterns	Require a detailed architectural specification and do not consider the history of versions
Historical Analysis Techniques	Source code of two versions of a system	ROSE [Zimmermann et al., 2004] and LAMARCK [Mileva et al., 2011]	Detecting anomalies in programming patterns	Do not consider architectural or modular aspects and use only two versions of the system under analysis

Table 2.1. Techniques for detecting programming anomalies

Technique	Input	Tools	Intention	Critical Assessment
Static Analysis Techniques	Source code of one version of a system	Lint [Johnson, 1977, Darwin, 1988], FindBugs [Hovemeyer and Pugh, 2004], and PMD [Copeland, 2005]	Detect anomalies in source code by means of idioms representing programming defects	Do not consider architectural or modular aspects and do not consider the history of versions

Chapter 3

Heuristics for Detecting Architectural Violations

This chapter is organized as follows. We start by providing an overview of the proposed approach for detecting architectural violations (Section 3.1). Next, we motivate and describe the heuristics to detect absences (Section 3.2) and divergences (Section 3.3). A complete formal specification of the heuristics is presented in Section 3.4. We also propose a strategy to rank the warnings produced by the heuristics according to their relevance (Section 3.5). Next, we present a prototype tool, called ArchLint, that supports the proposed heuristics (Section 3.6). We also present an architecture conformance process based on the proposed approach (Section 3.7). Finally, we conclude the chapter with a general discussion (Section 3.8).

3.1 Overview

Figure 3.1 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 information on the names of the components and a mapping from modules to components, using regular expressions (complete examples are provided in Sections 4.1.1 and 4.2.1). 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 established between classes, including dependencies due to method calls, variable declarations, inheritance, exceptions, etc.

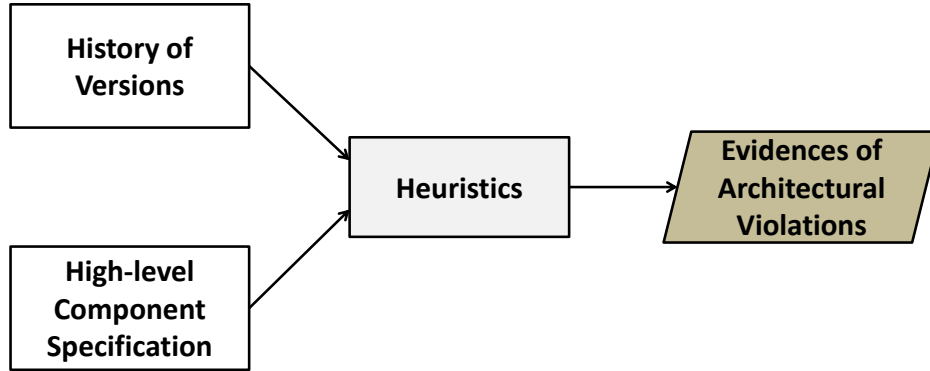


Figure 3.1. Input and output of the proposed heuristics

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 4.3.1, we applied our approach to an open-source system named Lucene, in which we reused high-level models independently defined by other researchers using information available in the systems' documentation.

3.2 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, regarding a given dependency. 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 in this case 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 fixed to introduce *dep*.

Figure 3.2 illustrates the proposed heuristic. 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 classes C_1 , C_4 , and C_5 .

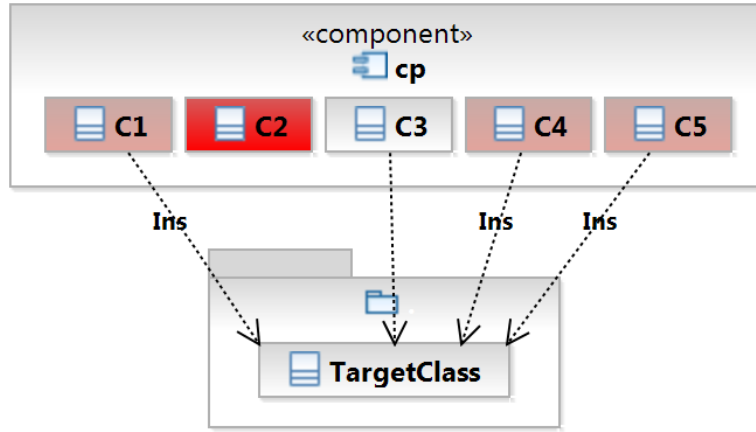


Figure 3.2. Example of absence (C_2 does not depend on *TargetClass*). The label *Ins* denotes a dependency inserted later in the class

Additionally, our approach considers 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 3.1 reports the types of dependency supported by this 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 establish 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 having 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 establishment of a dependency of type t with class c .

Table 3.1. 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_1 is used as a subclass of C_2
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

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

$$\begin{aligned}
Absences(cp) = \{ (x, c, t) \mid & \text{comp}(x) = cp \wedge \neg depends(x, c, t, H) \wedge \\
& DepScaRate(c, t, cp) \geq A_{sca} \wedge \\
& DepInsRate(c, t, cp) \geq A_{ins} \}
\end{aligned}$$

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* establish 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.

3.3 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 in the following.

3.3.1 Heuristic #1

This heuristic targets a common pattern of divergences: the use of frameworks and APIs by unauthorized components [Terra and Valente, 2009, Sarkar et al., 2009b]. 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 prescribes that the searching for divergences must be restricted to dependencies present in a small number of 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 these violations. 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 refactored 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 behave 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 3.3 illustrates the proposed heuristic. In this figure, class C_2 presents a divergence regarding *TargetModule* because: (a) C_2 is the only class in component cp_1 that depends on *TargetModule*; (b) many classes in cp_1 (such as C_1 , C_4 , and C_5) have in the past established and then removed a dependency with *TargetModule*; and (c) most dependencies with *TargetModule* come from by 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 established a dependency with any class in module m .

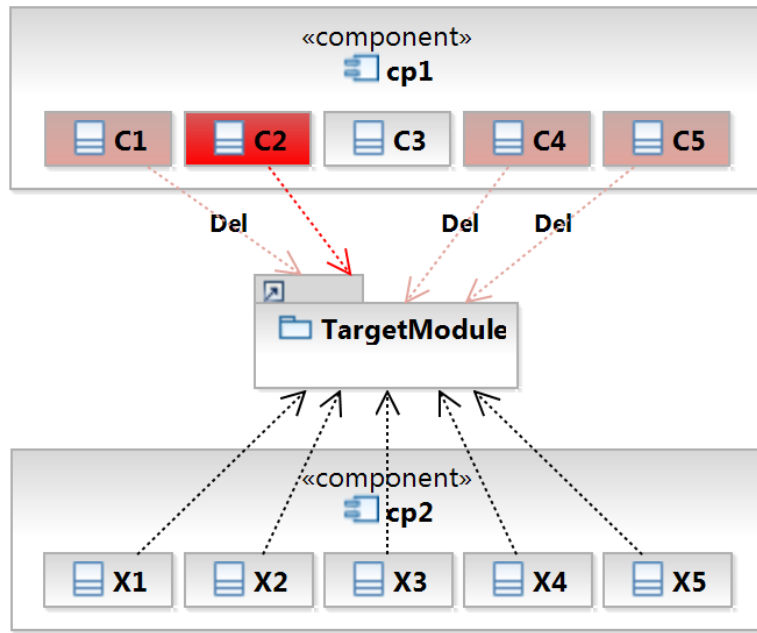


Figure 3.3. Example of divergence (C_2 depends on *TargetModule*). The label *Del* denotes a dependency removed in a previous version of the class

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 retrieves 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

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

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 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 .

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

3.3.2 Heuristic #2

Similarly to the previous case, this second heuristic restricts the analysis to dependencies defined by few classes of a component that were removed in the past (in other classes from the component). However, this heuristic has two important differences in contrast to the first one: (a) it is based on dependencies to a specific target class (instead of an entire module), which also includes the type of the dependency; and (b) it does not require the existence of a heavy-user for the dependency under analysis.

Figure 3.4 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 with *TargetClass*, as observed in the history of classes C_1 , C_4 , and C_5 .

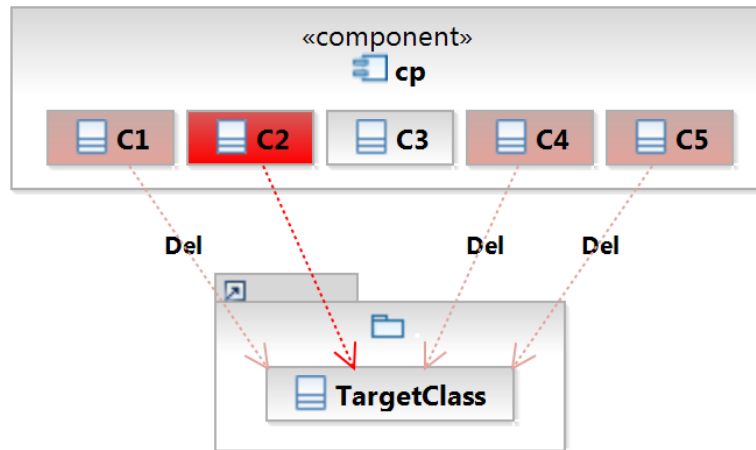


Figure 3.4. Example of divergence (C_2 depends on *TargetClass*). The label *Del* denotes a dependency removed in a previous version of the class

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:

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

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} .

3.3.3 Heuristic #3

This heuristic is based on the assumption that a common consequence of divergences is the creation of asymmetrical cycles between components. More specifically, as illustrated in Figure 3.5, 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 a few references in the reverse direction. The underlying assumption is that the components were originally designed to communicate unidirectionally and the dependencies in the “wrong” direction are likely to represent architectural violations (and might not be exceptions authorized by the architecture, e.g., for performance issues). 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., 2009a].

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:

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

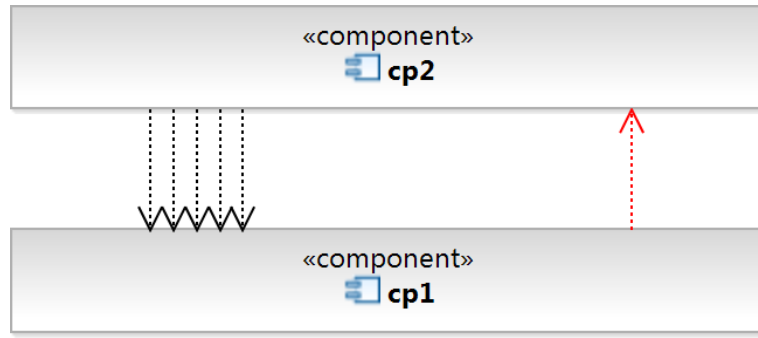


Figure 3.5. Divergences due to asymmetrical cycles

Using this definition, the heuristic is formalized as follows:

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

Basically, divergences are pairs of classes (x, c) where x is a class in component cp_1 (i.e., the component under analysis) that has 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} ; and (b) they are not dominant, since there are more dependencies in the reverse direction, as specified by the *Dependency Direction Weight* lower than 0.5.

3.4 Formal Definition

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

3.4.1 Notation

The definition of the heuristics relies on the following notation:

- $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$ is the set of all classes in the system under analysis.
- $CP = \{cp_1, cp_2, \dots, cp_n\}$ 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.

3.4.2 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:

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

$ClassComp(cp)$ 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:

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

$DepAdd(c, t, cp)$ 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:

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

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

$$\begin{aligned} Absences(cp) = \{ (x, c, t) \mid & comp(x) = cp \wedge \neg depends(x, c, t, H) \wedge \\ & DepScaRate(c, t, cp) \geq A_{sca} \wedge \\ & DepInsRate(c, t, cp) \geq A_{ins} \} \end{aligned}$$

3.4.3 Detecting Divergences

3.4.3.1 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:

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

$DepCompMod(m, cp)$ 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:

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

$DepDelRate(m, cp)$ 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:

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

3.4.3.2 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:

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

$DepDel(c, t, cp)$ 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:

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

3.4.3.3 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:

$$rf(cp_1, cp_2) = | \{ (x, c) \mid comp(x) = cp_1 \wedge comp(c) = cp_2 \wedge depends(x, c, _, H) \} |$$

$DepDirWeight(cp_1, cp_2)$ 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:

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

3.5 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 $cp = comp(x)$. Basically, this formula represents the arithmetic mean of the scattering rate and the insertion rate of the dependency that characterizes the absence. Therefore, the warnings denoting absences must be presented according to their ranking 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 this situation. In this case, the dependencies responsible for the “wrong” part of the second cycle should be ranked before the dependencies in the first cycle.

3.6 Tool Support

We implemented a prototype tool, called ArchLint, that supports the four heuristics for detecting architectural violations. As presented in Figure 3.6, ArchLint’s implementation follows a pipeline architectural pattern [Garlan and Shaw, 1996, Garlan, 2000] 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]. 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 Chapter 3. Basically, the heuristics are performed as SQL queries. Additionally, this module ranks the architectural violations evidences—as described in Section 3.5—and reports them to the architect of the system under analysis.

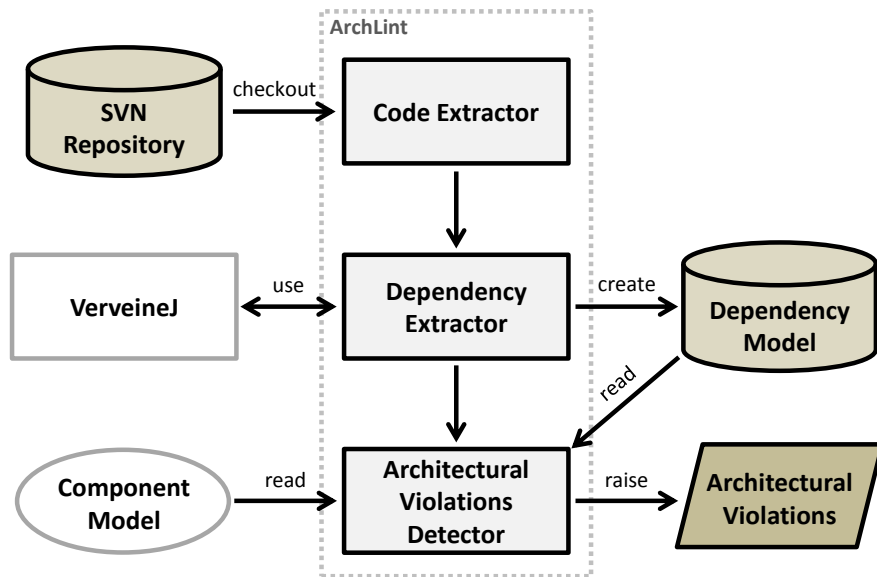


Figure 3.6. ArchLint architecture

3.7 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:

²<https://gforge.inria.fr/projects/verveinej>.

- 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 such as ArchLint. Moreover, based on our initial experiments with the proposed heuristics [Maffort et al., 2013a], 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 certainly 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 3.5.

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 argue that a tool such as 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 refactoring in the system to fix the detected violations. The architect may also decide to perform another iteration of the conformance process, with more flexible thresholds. This process should stop 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 anymore to experiment with new thresholds.

Figure 3.7 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 3.5—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.

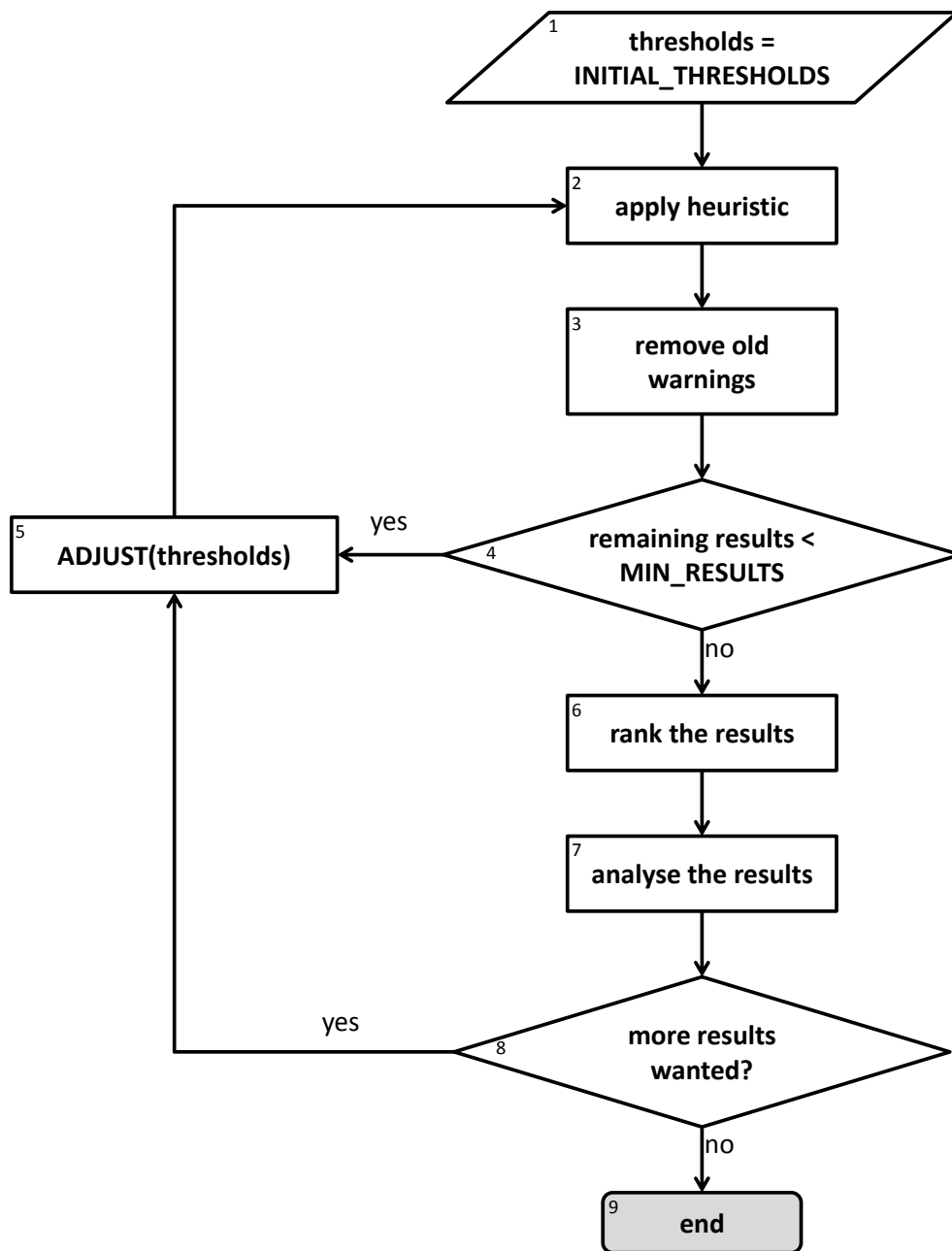


Figure 3.7. Architecture conformance using the proposed heuristics

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.

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 3.8 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%.

3.8 Final Remarks

Architectural deviations are a serious threat to the long term survival of software systems [Hochstein and Lindvall, 2005, Garcia et al., 2009]. The reason is that the accumulation of architectural violations in the source code can demand a lot of effort even when dealing with simple code changes [Macia et al., 2012]. On the other hand, the application of current techniques for architectural conformance checking requires a considerable effort [Knodel et al., 2008, Terra and Valente, 2009]. For instance: (i) reflexion models may require successive refinements of the high-level model in order to adequately express the full spectrum of absences and divergences that may be present in a large system; and (ii) domain-specific languages require a detailed definition of constraints among the classes of a system.

To address this shortcoming, we describe an approach that combines static and historical source code analysis techniques to provide an alternative technique for architecture conformance. The proposed approach includes four heuristics to discover suspicious dependencies in the source code, i.e., dependencies that may denote absences (missing expected dependencies) or divergences (existing unwanted dependencies). Furthermore, the proposed approach includes a technique to report the warnings sorted by their potential to denote true violations. To automate the violations detection

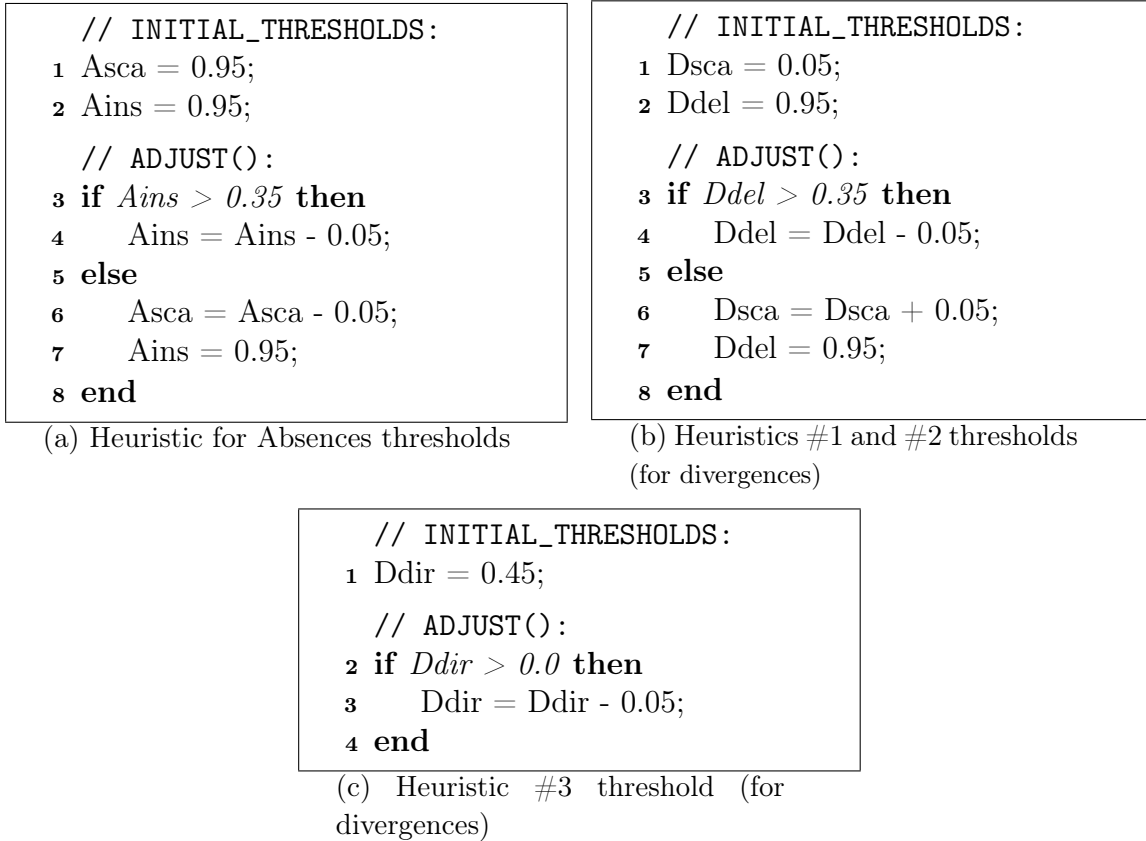


Figure 3.8. Initial thresholds values and thresholds adjustment procedures for each heuristic

triggering, we implemented a prototype tool, called ArchLint, that supports the four heuristics for detecting architectural violations and the ranking strategy.

In the next chapter, we conduct an evaluation of our approach by reporting and discussing results on its usage in four real-world systems. We also summarize our main findings and the lessons learned after designing and evaluating the heuristics-based approach.

Chapter 4

Evaluation

This chapter is organized as follows. Section 4.1 and Section 4.2 describe two studies on using our approach to evaluate the architecture of two proprietary information systems. Section 4.3 and Section 4.4 present a study that evaluates the architecture of two open-source systems. Next, Section 4.5 discusses the main findings of our work. Finally, Section 4.7 concludes the chapter with a general discussion.

4.1 First Study: SGA System

To start evaluating our approach, we conducted a first study using a real-world information system. Our central goal is to perform a first experiment with the conformance process described in Section 3.7. Specially, in this section we report the number of iterations required by this process, the precision achieved after each iteration, and the effectiveness of the strategy proposed to rank the warnings raised by a given heuristic.

4.1.1 Study Setup

In this first study we followed the architecture conformance process defined in Section 3.7 to detect violations in the architecture of an EJB-based information system used by a major Brazilian university, which for confidentiality reasons we will just call SGA. The SGA system automates many administrative activities, including functionalities for human resource management, finance and accounting management, and material management, among others. In the study, 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. All extracted versions were considered for computing the functions *DepInsRate* and *DepDelRate*, described in Sections 3.2 and 3.3. The last revision considered in the study has 1,864 classes and interfaces, organized in 100 packages, comprising around 273 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 *JavaServer 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.

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 4.1 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 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.

Table 4.1. 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*

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 3.7. Particularly, for each heuristic, we considered the initial thresholds and the thresholds adjustment procedure suggested in Figure 3.8. Moreover, SGA’s architect was only requested to evaluate the warnings generated by iterations that produce at least 10 new warnings (MIN_RESULTS constant). 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 study. 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 in 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 heuristic 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.

4.1.2 Results

In this section, we present the results achieved after following the proposed conformance process to detect absences (Section 4.1.2.1) and divergences (Sections 4.1.2.2, 4.1.2.3, and 4.1.2.4) in the architecture of the SGA system.

4.1.2.1 Results for Absences

Table 4.2 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 4.2. 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 are 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.

Table 4.2. 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

As can be observed in Table 4.2, we decided to stop the process after seven iterations, including iterations #1 and #4 that did not generate enough warnings for evaluation. In the remaining five iterations, the architect evaluated 128 warnings, with an overall precision of 86.7%. In Table 4.2, 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 4.2 present a tendency to decrease after each iteration. However, in the last iteration the ranking strategy achieved 90% of the effectiveness of a perfect ranking algorithm.

We finished with 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 one:¹

Component:	<code>IService</code>
Class:	<code>br.sga.doc.ejb.facade.DictionaryService</code>
Missing Dependency:	<code>javax.ejb.Remote ClassAnnotation</code>
<i>DepScaRate; DepInsRate:</i>	0.990; 0.800

¹To improve the thesis's comprehension, we translated the class names from Portuguese to English.

In the SGA system, the architect explained that interfaces in the `IService` 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 were later maintained to include the annotation (*DepInsRate*). The lack of this annotation does not impact 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.

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

Component:	<code>BusinessEntity</code>
Class:	<code>br.sga.core.domain.FederatedUnit</code>
Missing Dependency:	<code>br.sga.core.domain.AuditInfo</code> Inheritance
<i>DepScaRate; DepInsRate:</i>	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 a 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).

4.1.2.2 Results for Divergences - Heuristic #1

Table 4.3 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.

Table 4.3. 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

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

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 DAOs establish a dependency with the `IService` class ($DepScaRate$). Moreover, in the past, 75% of the classes that established a dependency like that in a given version were later refactored to remove the dependency ($DepDelRate$). Finally, the `br.sga.ejb.facade` package 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`).

4.1.2.3 Results for Divergences - Heuristic #2

Table 4.4 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 not productive anymore, demanding the evaluation of many false positives per true warning discovered.

Table 4.4. Detecting divergences in the SGA system using Heuristic #2

Iteration	$D_{sca}; D_{del}$	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
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

Despite the lower precision, by analyzing the *nDCG* values in Table 4.4, 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 one:

```

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 to remove the dependencies to this class coming from `ManagedBean`. In fact, 88.8% of the classes that once had this dependency were later refactored 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 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 refactoring in the system responsible for the high *Dependency Deletion Rate* was motivated by a design decision not related to removing architectural violations.

4.1.2.4 Results for Divergences - Heuristic #3

Table 4.5 shows the results achieved by the third heuristic for divergences. In this case, as defined in Figure 3.8, 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 has been produced.

Table 4.5. 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	—	—	—	—

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.

4.1.2.5 Overall Results for Divergences

Table 4.6 presents the precision achieved by our approach for divergences, considering the warnings evaluated for the three heuristics. As can be observed, both heuristic #1 and heuristic #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%.

Table 4.6. 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%

4.1.3 Comparison with Reflexion Models

This section compares our results with reflexion models (RM) [Murphy et al., 2001b, Murphy et al., 1995], which is a well-known and lightweight approach for architecture conformance. 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 Figure 4.1, 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 (JavaServer 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 (dependencies that are not expected by the architect) and absences (dependencies that are expected but not found).

Figure 4.2(a) compares the results for divergences achieved by RM and by our approach. As mentioned in Section 4.1.2, 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` establishes a dependency with `JavaIO`, but with the wrong class in this component. Specifically, an architectural rule states that `ManagedBean` can only establish 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`.

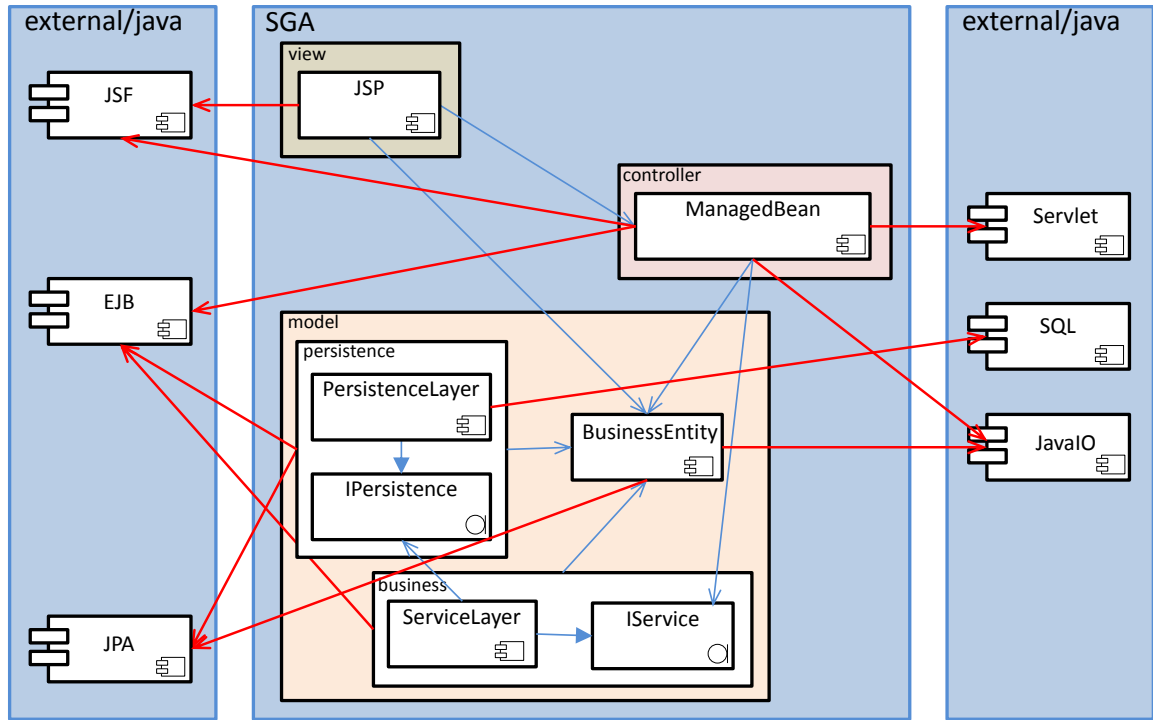


Figure 4.1. Enriched high-level model for the SGA system

After this modification, we must update the dependency from `ManagedBean` to reach just the `IOException` subcomponent. In fact, this frequent need to refine reflexion models motivated the extension of the original proposal with hierarchical modules [Koschke and Simon, 2003].

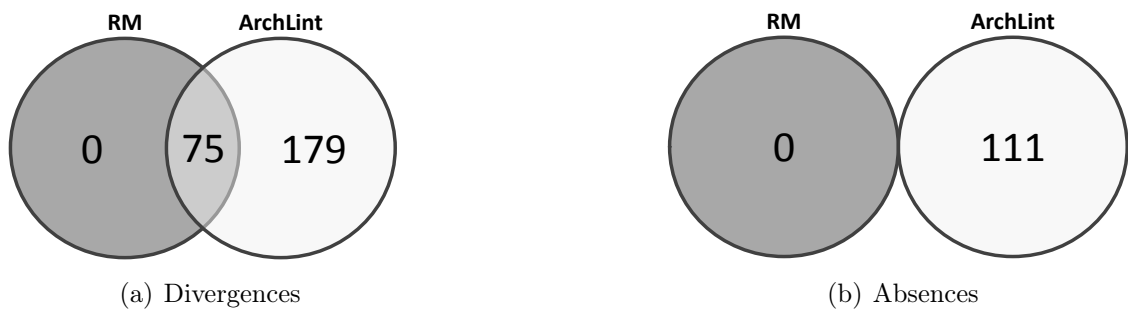


Figure 4.2. Absences and divergences detected by RM and the proposed heuristics

Figure 4.2(b) compares the results for absences achieved by RM and by our approach. As reported in Section 4.1.2, 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 Figure 4.1, the high-level model prescribes that 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%).

4.1.4 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 the first thresholds evaluated by the architect in the original study (see Table 4.2). 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 4.7 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).

Table 4.7. 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

To explain these results, we first characterize the refactorings that have an impact in the proposed heuristics. The heuristic for absences monitors a refactoring that inserts a missing dependency in the target class, which we will refer to *Insert Missing Dependency* refactoring. In the case of divergences, the heuristics monitor a refactoring that removes an undesirable dependency from a target class, which we will refer to *Remove Undesirable Dependency* refactoring. Figure 4.3 reports the distribution of these refactorings in our dataset, in four years. We can observe that both refactorings happened most of the times in the first year of SGA’s evolution. For example, 53% of the *Insert Missing Dependency* refactorings were performed in 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 refactorings 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 refactorings 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 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 refactorings 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, heuristic #1

for divergences was not deeply impacted by removing the commits of the first year due to its evaluation with flexible thresholds.

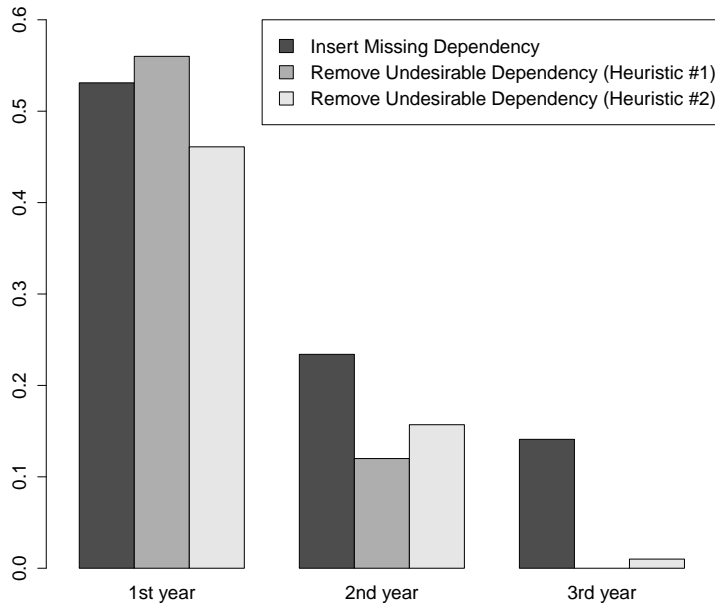


Figure 4.3. Distribution of the refactoring operations by year

4.2 Second Study: M2M System

In this section, we report the application of our approach in a real-world proprietary system, which we are just calling M2M, for confidentiality reasons.

4.2.1 Study Setup

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. Moreover, M2M integrates Brazilian government systems, which allows automatic loading of information of these systems, and assists in resource management, settlement expenses, and control electronic auctions.

In this study, we considered 61,785 revisions available in the system’s control version repository (all available revisions), from November, 2010 to October, 2013.

The last revision considered in the study 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, whose relational database has 107 GB.

Similarly to SGA, we asked M2M’s architect to define the system’s high-level component model. Table 4.8 presents the components suggested by the architect and in the high-level components the regular expressions proposed by the architect that define the classes in each component, besides the respective number of classes. We can observe that the regular expressions in M2M maps 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`).

Table 4.8. High-level components in the M2M system

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

In practice, the regular expressions in Table 4.8 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 have produced at least 10 new warnings. In this case, the architect carefully examined the warnings and classified them as true or false positives.

4.2.2 Results for the M2M system

Table 4.9 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%.² Moreover, to discover the violations we executed seven iterations, raising 279 warnings with an overall precision of 53.8%. Section 4.2.3 presents a detailed description of the warnings detected by each heuristic.

Table 4.9. Precision considering the warnings raised in M2M system

	Iterations	Warnings	Precision	
			Mean	Overall
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%

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).
- *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).

²*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.

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

4.2.3 M2M Conformance Process

In this section, we show the results achieved after each iteration when detecting architectural violations in the M2M system.

Results for Absences

Table 4.10 presents the results achieved by each iteration when detecting absences. As can be observed, two iterations are performed, achieving a precision ranging from 77.8% to 82.1%. In total, 112 warnings are detected with an overall precision of 82.1%. In Table 4.10 we can also observe that the criteria used to rank the warnings was quite effective, producing *nDCG* results higher than 0.97.

Table 4.10. Detecting absences in the M2M system

Iteration	D_{dir}	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

In the M2M system we performed only two iterations to detecting absences because the architect established a limit of around 100 warnings to evaluate. He considered that such true warnings should be first addressed by the development team, before continue looking for new warnings.

Results for Divergences - Heuristic #1

As reported before, the Heuristic #1 for detecting divergences did not report warnings in the M2M system.

Results for Divergences - Heuristic #2

Table 4.11 shows the results achieved by the second heuristic for divergences. In this case, we performed nine iterations, with three iterations including evaluation

by the architect. As can be observed, we achieved a precision ranging from 18.5% to 90.0%. In summary, we detected 119 warnings with an overall precision of 18.5%.

Table 4.11. Detecting divergences in the M2M system using Heuristic #2

Iteration	$A_{sca}; A_{ins}$	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
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

In Table 4.11, there is an expressive decrease in the precision after each iteration. For example, in iteration #4 we achieved a precision of 90% and in the iteration #9 (the last iteration) the precision was 18.5%. Similarly, the *nDCG* values also present a decrease tendency after each iteration. The smallest result, achieved in the last iteration, is 50% of the effectiveness of a perfect ranking algorithm.

Results for Divergences - Heuristic #3

Table 4.12 shows the results achieved by the third heuristic for divergences. In this case, five iterations are performed, with two evaluation steps. As can be observed, in the first evaluation by the architect (iteration #3) we found 12 warnings, using $D_{dir} = 0.10$, which resulted in a precision of 41.7%. Finally, in the next evaluation (iteration #5), when we defined $D_{dir} = 0.00$, we found 36 warnings with a precision of 86.6%. In total, we detected 48 warnings with an overall precision of 75.0%.

As can be observed in Table 4.12, despite the precision lower than 50% in the iteration #3, the *nDCG* result shows that the proposed ranking strategy was quite effective.

Table 4.12. Detecting divergences in the M2M system using Heuristic #3

Iteration	D_{dir}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
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

4.3 Third Study: Lucene System

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

4.3.1 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 4.13 lists the components defined by the Lucene’s HLM.

Table 4.13. 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.*

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 third 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 indicate 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 there must exist a dependency from **Search** to **Index**. However, **Search** is a component with 351 classes and therefore a single class from **Search** that relies on **Index** is sufficient to hide eventual absences in the remaining classes of the component. Certainly, it is possible to refine the HLM by creating a nested component in **Search** with exactly the classes that must depend on **Index** and establishing an edge between each of such classes and **Index**. However, the proliferation of nested components increases the complexity and contrasts to the lightweight profile normally associated with RM-based techniques. For this reason, we decided to do not evaluate our approach for absence detection in Lucene.

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

4.3.2 Results for the Lucene system

Table 4.14 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 at each iteration ranging from 7.0% to 98.5%. Section 4.3.3 presents a detailed description of the warnings detected by each heuristic.

Table 4.14. Precision considering the warnings raised in Lucene system

	Iterations	Warnings	Precision	
			Mean	Overall
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%

An analysis of the missing divergences—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 exceeds 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.

For computing recall, we consider as false negatives the violations reported by the reflexion model but that are not detected by our approach. In this way, reflexion model detected 312 violations and our approach detected 264 violations. Therefore, the result for recall is 84.62% (264/312).

4.3.3 Lucene Conformance Process

In this section, we show the results for divergences achieved after each iteration searching for detecting architectural violations in the Lucene system. The heuristic for absences did not report warnings in Lucene, as mentioned before.

Results for Divergences - Heuristic #1

Table 4.15 shows the results after each iteration of the conformance process, when configured to provide warnings using the first heuristic for divergences. As can be observed, we performed 12 iterations, comparing the warnings raised by our approach with the violations detected using reflexion models. We achieved a precision in each iteration ranging from 0.0% to 100.0%. As overall result, we analyzed 168 warnings with a precision of 55.4%.

Table 4.15. Detecting divergences in Lucene using Heuristic #1

Iteration	$D_{sca}; D_{del}$	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
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

Results for Divergences - Heuristic #2

Table 4.16 shows the results achieved by the second heuristic for divergences. In this case, we performed nine iterations, with four evaluation steps. In summary, we achieved a precision in each iteration ranging from 3.1% to 12.5%. In total, we analyzed 114 warnings with an overall precision of 7.9%.

Table 4.16. 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

Results for Divergences - Heuristic #3

Table 4.17 shows the results achieved by the third heuristic for divergences. In this case, we performed seven iterations, with six evaluation steps. As result, we achieved a precision in each iteration ranging from 90.9% to 100.0%. In total, we analyzed 164 warnings with an overall precision of 98.8%.

Table 4.17. 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

4.4 Fourth Study: ArgoUML System

Similarly to Section 4.3, in this section we report the application of the proposed heuristics in an open-source system named ArgoUML.

4.4.1 Study Setup

In this system, similarly to the Lucene system, our evaluation is also fully based on a Reflexion Model (RM) independently proposed by Bittencourt et al. [Bittencourt, 2012]. Table 4.18 lists the components defined by the ArgoUML's HLM.

Table 4.18. High-level components in ArgoUML

Component	Regular Expression
Application	org.argouml.application.*
Diagrams	org.argouml.uml.diagram.*
Notation	org.argouml.notation.*
Explorer	org.argouml.ui.explorer.*
CodeGeneration	org.argouml.language.*
ReverseEngineering	org.argouml.uml.reveng.*
Persistence	org.argouml.persistence.*
Profile	org.argouml.profile.*
Help	org.argouml.help.*
ModuleLoader	org.argouml.moduleloader <or> org.argouml.application.modules <or> org.argouml.application.api
GUI	org.argouml.ui.*
Model	org.argouml.model.*
Internationalization	org.argouml.i18n.*
TaskManagement	org.argouml.taskmgmt.*
Configuration	org.argouml.configuration.*
SwingExtensions	org.argouml.swingext.*
OCL	org.argouml.ocl.*
Critics	org.argouml.cognitive.*
JavaCodeGeneration	org.argouml.language.java.*

As reported in Section 4.3, the HLM was carefully designed for architecture conformance purposes, for this reason we considered the computed reflexion models as a reliable oracle for evaluating the precision of the heuristics. More specifically, in this fourth study, we replaced the architect with a reflexion model. Moreover, we decided to stop the iterative process followed for each heuristic when we targeted around 100 warnings per heuristic.

In the case of absences, the reflexion model also did not indicate absences in ArgoUML for the same reasons appointed to the Lucene system, i.e., in RM a single class in a component satisfying the prescribed architectural rule is sufficient to hide all absences in this component.

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

4.4.2 Results for the ArgoUML system

Table 4.19 reports the precision achieved by the heuristics for divergences. The overall precision was 53.3%. In 10 iterations, our approach raised 152 warnings with a mean precision in the iterations used for each heuristic ranging from 14.8% to 100.0%. Section 4.4.3 presents a detailed description of the warnings detected by each heuristic.

Table 4.19. Precision considering the warnings raised in ArgoUML system

	Iterations	Warnings	Precision	
			Mean	Overall
Divergence - Heuristic #1	6	105	60.0%	58.1%
Divergence - Heuristic #2	2	31	14.8%	12.9%
Divergence - Heuristic #3	2	16	100.0%	100.0%
All Heuristics	10	152	58.3%	53.3%

Similarly to Lucene system, some divergences were missed by our approach but they were detected by the reflexion model. We observed that many divergences with a high scattering and a low deletion rate were missed. For example, in ArgoUML, it is common to observe divergences that are not spatially and historically confined in their source components. Therefore, we argue that ArgoUML's architecture might have evolved during the time frame considered in our study. As 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.

In ArgoUML, recall is computed in the same way as in Lucene, i.e., we consider as false negatives the violations reported by the reflexion model that are detected by our approach. In this way, reflexion model detected 148 violations and our approach detected 81 violations. Therefore, the result for recall is 54.7% (81/148).

4.4.3 ArgoUML Conformance Process

In this section, we show the results achieved after each iteration when searching for architectural violations in the ArgoUML system. As stated previously, the heuristic

for absences did not report warnings in ArgoUML system.

Results for Divergences - Heuristic #1

Table 4.20 shows the results achieved after each iteration of the conformance process using the Heuristic #1 for divergences. As can be observed, we performed 24 iterations achieving a precision ranging from 19.0% to 100.0%. As overall result, we analyzed 105 warnings with a precision of 58.1%.

Table 4.20. Detecting divergences in ArgoUML using Heuristic #1

Iteration	$D_{sca}; D_{del}$	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.05;0.90	2	2	—	—	—	—
2	0.05;0.80	5	3	—	—	—	—
3	0.05;0.70	9	4	—	—	—	—
4	0.05;0.60	11	2	11	63.6%	63.6%	0.93
5	0.05;0.50	24	13	13	46.2%	54.2%	0.62
6	0.05;0.30	26	2	—	—	—	—
7	0.05;0.25	29	3	—	—	—	—
8	0.05;0.00	54	25	30	60.0%	57.4%	0.70
9	0.10;0.85	6	4	—	—	—	—
14	0.10;0.80	11	2	—	—	—	—
17	0.10;0.75	14	3	—	—	—	—
18	0.10;0.65	23	5	14	71.4%	60.3%	0.61
19	0.10;0.60	27	2	—	—	—	—
20	0.10;0.50	42	2	—	—	—	—
21	0.10;0.40	59	17	21	19.0%	50.6%	1.00
22	0.10;0.30	63	2	—	—	—	—
23	0.10;0.25	69	3	—	—	—	—
24	0.10;0.20	80	11	16	100.0%	58.1%	1.00

Results for Divergences - Heuristic #2

Table 4.21 presents the results for the second heuristic for divergences. In this case, we performed seven iterations, with two evaluation steps. As result, we achieved a precision ranging from 9.5% to 20.0%. In short, we inspect 31 warnings with an overall precision of 12.9%.

Table 4.21. Detecting divergences in ArgoUML using Heuristic #2

Iteration	$D_{sca}; D_{del}$	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.05;0.95	2	2	—	—	—	—
2	0.05;0.90	7	5	—	—	—	—
3	0.05;0.85	8	1	—	—	—	—
4	0.05;0.80	10	2	10	20.0%	20.0%	0.47
5	0.05;0.75	13	3	—	—	—	—
6	0.05;0.70	18	5	—	—	—	—
7	0.05;0.65	31	13	21	9.5%	12.9%	0.82

Results for Divergences - Heuristic #3

Table 4.22 reports the results for the third heuristic for divergences. As can be observed, we performed seven iterations, with two evaluation steps. In total, we found 16 warnings, and they are all ranked as true positives, i.e., with an overall precision of 100.0%.

Table 4.22. Detecting divergences in ArgoUML using Heuristic #3

Iteration	D_{dir}	Warnings			Precision		nDCG
		Iter.	New	Eval.	Iter.	Overall	
1	0.30	2	2	—	—	—	—
2	0.25	7	5	—	—	—	—
3	0.20	15	8	15	100.0%	100.0%	1.00
4	0.15	15	0	—	—	—	—
5	0.10	15	0	—	—	—	—
6	0.05	16	1	—	—	—	—
7	0.00	16	0	1	100.0%	100.0%	1.00

4.5 Discussion

In this section, we discuss the main lessons learned in the studies reported in this chapter:

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.

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 4.4(a) and 4.4(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), and 1.7% (Lucene). On the other hand, there are more differences in terms of the deletion 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.

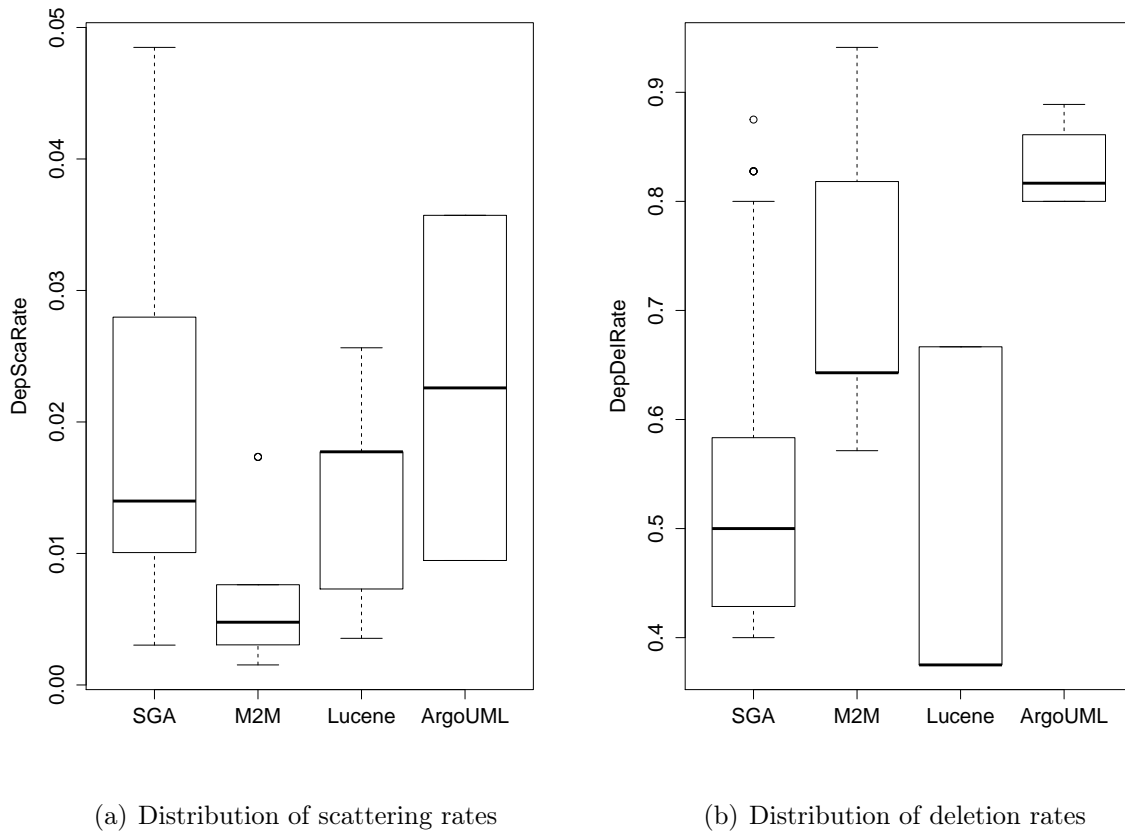


Figure 4.4. Thresholds distribution in heuristic #2 for divergences

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 proposed 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 normally 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.

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 Figure 4.5(a), 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 Figure 4.5(b). Finally, in Lucene, we found an expressive intersection between heuristics #1 and #3, as showed in Figure 4.5(c). 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.

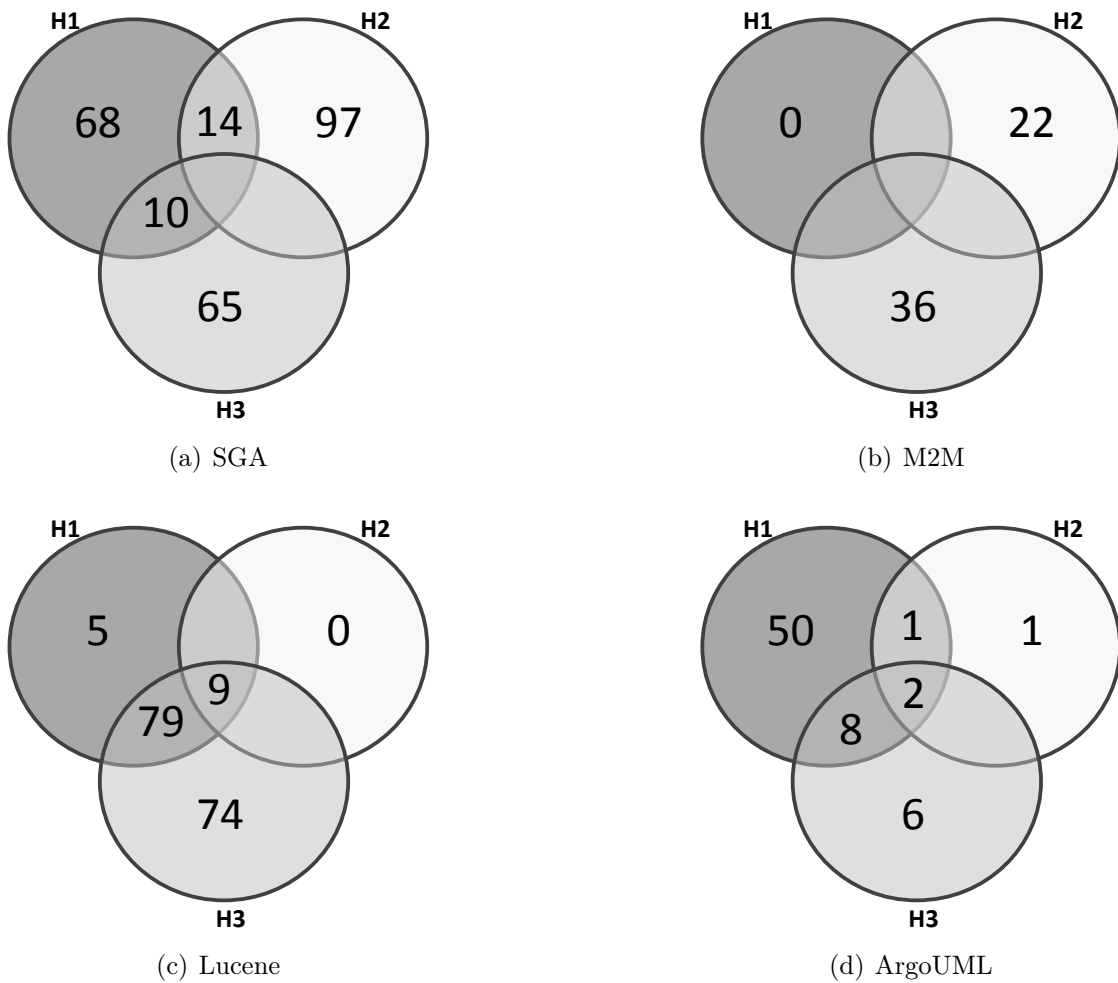


Figure 4.5. Warnings raised by more than one heuristic for detecting divergences

What are the most common dependency types responsible for violations?

As defined in Section 3, the heuristics for absence and the second heuristic for divergence consider a violation regarding a specific dependency type. Table 4.23 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 4.1.2.1). 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.OneToOne` annotation.

Table 4.23. Most common dependency types in the SGA system

	Absence	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%

4.6 Threats to Validity

In the case of SGA and M2M systems, we relied on an architect to design our initial model and to classify our warnings. Therefore, as any human-made artifact, the model and the classification are subjected to errors and imprecision. However, we interviewed a senior architect, with a complete domain of SGA’s and M2M’s architecture and implementation. Furthermore, one can argue that this architect might be influenced to design a model favoring our approach. However, we never explained to the architect the heuristics followed to discover architectural violations.

In the case of Lucene and ArgoUML, our evaluation is fully based on a Reflexion Model (RM) independently proposed by Bittencourt et al. [Bittencourt, 2012]. We reused the component specification from the high-level models defined as the input for the proposed heuristics. For this reason, it is possible that the Lucene’s and ArgoUML’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 establish a benchmark for architecture conformance.

4.7 Final Remarks

In this chapter, we evaluated the architecture of two industrial-strength information systems, for which we detected 539 architectural violations, with an overall precision of 62.7% and 53.8%. We also evaluated our approach with two open-source systems, for which we detected 345 architectural violations, achieving an overall precision of 53.3% and 59.2%.

In conclusion, we claim we were able to provide an alternative and iterative technique for architecture conformance that does not require successive refinements in architectural models (as reflexion models) neither requires the extensive specification of architectural constraints (as domain-specific languages). On the other hand, the proposed approach can generate false positive warnings, as common in most bug finding tools based on static analysis.

Chapter 5

Extracting Architectural Patterns

In this chapter, we start by presenting our motivation to perform an exploratory study on using data mining techniques to reveal architectural patterns (Section 5.1). Next, Section 5.2 presents an overview of the proposed approach to infer patterns for detecting absences and divergences, respectively. Section 5.3 presents an evaluation of the proposed approach in a real system. Section 5.4 presents a comparative discussion between the approach based on data mining technique and the approach based on heuristics.

5.1 Motivation

The architecture of a system prescribes the organization of its components, their relationships, constraints, and the principles that guided its design and evolution over time. 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 its development and evolution.

However, as previously reported, during the development of a software product, anomalies regarding the proposed architectural model are normally introduced. In practice, the introduction of architectural violations is very common [Knodel and Popescu, 2007], and it usually makes more complex subsequent maintenance tasks since the concrete architecture does not match the planned and documented architecture anymore [Sarkar et al., 2009a].

In this thesis, we presented in Chapter 3 a set of heuristics for detecting architectural violations, which are based on our previous experience in the area. Specifically, these heuristics were designed from abstract scenarios where architectural

violations frequently occur, considering the assumption that architectural violations are frequently corrected.

However, the proposed heuristics do not cover the entire spectrum of architectural scenarios where violations might occur. For example, as described before, the proposed heuristics consider only direct dependencies between classes, modules, and components. Nevertheless, scenarios based on co-dependency analysis and/or causal relationships among dependencies are not considered by the proposed technique. Co-dependency analysis can be used, for example, to assess multiple dependencies that occur simultaneously. For example, in a particular system using *JPA* (Java Persistence API) persistence framework, the classes in the *Domain* component are mapped to tables in the database using *Entity* annotations. Additionally, these classes frequently also have an annotation *Id* on an attribute of type *Long*. Therefore, the absence of this annotation might denote an architectural anomaly.

As another example, in some systems, classes are instrumented according to certain contracts to make them able to provide services to other classes. In the aforementioned hypothetical system, classes in the *Domain* component are annotated with *Entity* to make it possible for a specific class of the *JPA* framework, called *EntityManager*, perform persistence operations in these classes. Thus, there is a well-know reason for that specific *Domain* classes to receive an *Entity* annotation. By contrast, classes annotated with *Entity* that are not accessed by the persistence framework are unnecessarily fulfilling an contract. In the presented example, the introduction of unnecessary code can lead to problems, such as performance degradation, unnecessary memory allocation, a decrease in readability and maintainability of the source code, etc.

It is well-know that data mining-based techniques should be used when you want to discover how often two or more items from a set occur simultaneously. Furthermore, it is possible to identify co-occurrence dependencies between these items, allowing to infer causal relationships in these dependencies. Therefore, this chapter investigates a data mining based approach that assumes that the inception of architectural violations in software products is a common event and that some violations are detected and corrected in future revisions by means of inspections and/or quality assurance activities. Our ultimate goal is to evaluate whether a data mining based approach would be more effective than the heuristic-based approach described in Chapter 3.

5.2 Data Mining Based Approach

This chapter describes the methodology we followed for detecting architectural violations in object-oriented software systems. The proposed methodology relies on data mining techniques over historical dependencies between the classes of a target system. This historical information is retrieved from version control system repositories. Basically, the idea is to mine structural and historical dependencies between the classes of the target system.

Figure 5.1 illustrates in details the approach we followed for detecting evidences of 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 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) the dependencies database and (b) a high-level component specification. In our approach, we assume that classes are statically organized in modules (packages in the Java terminology) 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 *Architectural Miner* populates a Prolog database describing the structural and historical relations available in the source code. After that, the *Architectural Miner* relies on 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* module relies on such architectural patterns to detect evidences of architectural violation.

The proposed methodology identifies evidences of architectural violations by relying on low frequency hypotheses and past refactoring tasks performed on structural dependencies. The assumption is that dependencies violating architectural patterns are rare events in the space-time domain, i.e., they appear in a small number of classes and are eventually corrected during the system evolution. In other words, as in Chapter 3, this methodology is based on the idea that architectural patterns are frequently followed and violations represent a small percentage of the cases. For example, if most classes of a source component access a specific class \mathcal{C} and many classes in this component that did not access \mathcal{C} at first were modified to access it afterwards (as observed in the system history), then we can suppose that there is an architectural pattern prescribing that classes from the source component must access \mathcal{C} .

Our approach is based on a data mining technique called *frequent itemset*

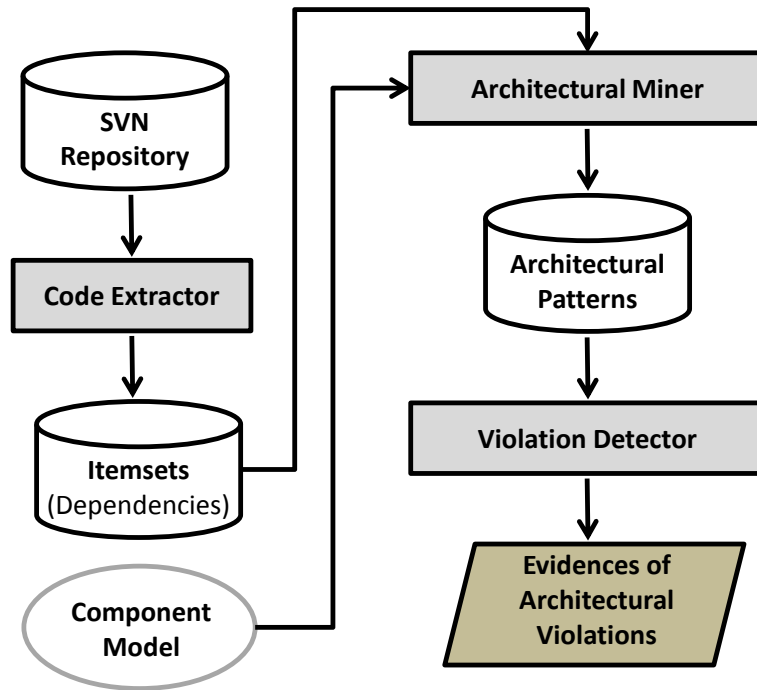


Figure 5.1. Data Mining proposed approach

mining [Agrawal and Srikant, 1994], which efficiently finds frequent itemsets in a transaction dataset, where an *itemset* is a set of items. The frequent itemset mining algorithm enumerates itemsets that occur frequently in a dataset. Therefore, this technique defines the *support* as the number of occurrences of a subset of items (sub-itemset). A sub-itemset is considered frequent when its support is greater than a specified threshold called *minimum support*. Thus, support counts the number of times a sub-itemset occurs in the itemsets database.

After the frequent itemsets have been mined, we can compute *association rules* [Zaki and Meira Jr., 2011, Agrawal et al., 1993]. From 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. An association rule is usually expressed as $\mathcal{A} \Rightarrow \mathcal{C}$, where \mathcal{A} and \mathcal{C} are itemsets. Each association rule has a *confidence*, a value that represents the probability of a database transaction covered by an antecedent term \mathcal{A} (pre-condition) be covered by a consequent term \mathcal{C} (consequence).

The investigation reported in this chapter assumes that association rules are effective to detect violations in architectural patterns. In other words, assuming that the confidence of the rule (pattern) is very high, as 99% for example, an itemset containing the antecedent term \mathcal{A} but not the consequent term \mathcal{C} can be regarded as

violating the pattern, i.e., it represents a strong evidence of architectural violation.

To calculate frequent itemsets and to generate association rules, we use a FP-tree-based mining algorithm, called FPGrowth [Zaki and Meira Jr., 2011]. 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 remainder of this section is organized as follows: Section 5.2.1 presents the methodology proposed to detect evidences of absences; Section 5.2.2 describes the methodology proposed for divergences.

5.2.1 Mining for Absences

As presented in Section 3.2, an absence happens whenever a dependency is defined by the planned architecture but it does *not exist* in the source code [Murphy et al., 1995, Passos et al., 2010].

Similarly to the heuristic-based approach, to detect absences using data mining techniques, 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. More specifically, we mine for patterns representing dependencies that 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 according to the component model provided as input. Second, from the classes in each component, we identify evolutionary architectural patterns. For instance, Figure 5.2 illustrates an example of absence. In this case, the planned architecture prescribes that classes located in module *DTO* must use services provided by a specific class located in *JPA* module, such as the *Entity* class. In this case, an absence is counted for each class in *DTO* that does not follow this rule. In the second step of the proposed procedure, we check how frequently classes in the *DTO* module 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 behind the evolutionary architectural patterns is to reinforce the violation evidences suggested by the first step. The assumption is that absences are

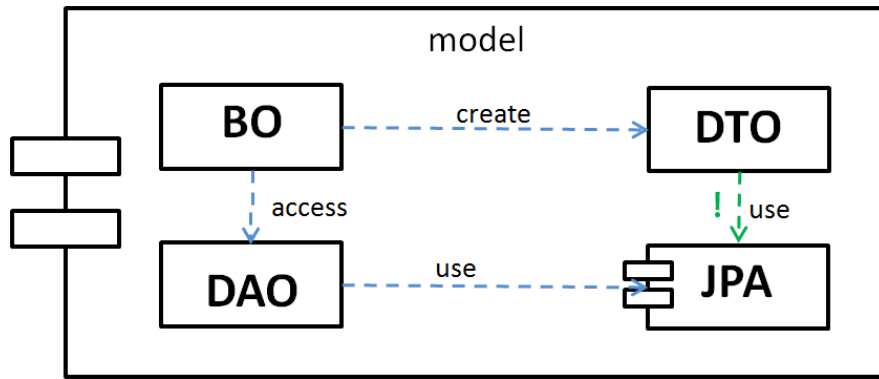


Figure 5.2. Example of absence (*DTO* must use *JPA*)

frequently detected and fixed, i.e., classes created without a dependency prescribed by the planned architecture are frequently fixed in future revisions.

To find correlations among the dependencies, it is initially necessary to compute the frequent itemset mining dataset. For this purpose, we rely on a dataset of Prolog facts, which describes both the dependencies and the 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 informed by the architect of the system under analysis. The *module* predicate defines the packages, in Java terminology, of the system. The *class* predicate represents a class in the system. The *dependency* predicate defines a dependency relation between two classes (*BaseClassId* depends on *TargetClassId*). In this 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 was detected in some version of the system. A short example of this Prolog database is presented next:

```

1: component(5,'domain').
2: component(12,'jpa').
3: ...
4: module(10,5,'br.sga.aaa.core.domain').
5: module(170,12,'javax.persistence').
6: ...
7: class(531558,10,'Auditing').

```

```

8: class(540800,10,'Functionality').
9: class(117,170,'Entity').
10: ...
11: dependency(21,531558,117,false,false,true).
12: dependency(2004,540800,117,true,true,true).

```

As can be observed in line 11, the class *br.sga.aaa.core.domain.Auditing* (id=531558) was created without a dependency with *javax.persistence.Entity* (id=117). Moreover, this dependency also does not exist in the current version of the system (attribute *ExistCurrently* = *false*). Nevertheless, this dependency was inserted in the past (attribute *AddAny* = *true*). The dependency presented in line 12, between *br.sga.aaa.core.domain.Functionality* (id=540800) and *javax.persistence.Entity* (id=117) was detected in the first revision of the class *br.sga.aaa.core.domain.Functionality* in the version control repository (attribute *CreatedWith* = *true*). Moreover, this dependency was preserved over time and it was also detected in the current version of the system (attribute *ExistCurrently* = *true*).

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

```

BaseComponent(bcomp),BaseModule(bmod),BaseClass(bclass)
[,TargetComponent(tcomp),TargetModule(tmod),TargetClass(tclass)]*

```

By mining the itemset database using the FPGrowth algorithm, we can find the frequent sub-itemsets and generate the association rules representing the corresponding *architectural patterns*. Basically, these patterns 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) should depend on the class *Entity* (consequent term). In other words, according to 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 number 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 corresponds to a row in the itemset database as follows:

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

We compute the association rules expressing this *dependency evolutionary patterns* using the FPGrowth algorithm, using a given support (A_{deps}) and confidence (A_{depc}) threshold. For instance, suppose the following pattern:

```
{BaseComponent('domain'),
    TargetClass('Entity'),
    CreatedWith(false)}=> {ExistCurrently(true)}
```

This pattern states that classes on the component *domain* created without dependency with the class *Entity* were frequently refactored to include this dependency, which also exists in the current version of the system.

The second step results are combined with the results obtained in the first step. For example, suppose that in the first step the classes in the *domain* component that do not depend on *Entity* were classified as evidences of absences. Moreover, suppose that in the second step we concluded that classes in *domain* created without a dependency with *Entity* frequently (i.e., with a high support and confidence) were refactored to include this dependency during their evolution, which therefore reinforces the evidence detected in the first step.

5.2.2 Mining for Divergences

As described in Section 3.3, 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].

Likewise the heuristic for absences, we assume that divergences happen in a *small percentage* of cases. However, a standard frequent itemset mining technique is not suitable for detecting minorities. For this reason, we in fact mine for dependencies that 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. For instance, Figure 5.3 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* module. In the second step of the proposed procedure, we check how frequently classes in the component *BO* that had a dependency with classes on *JPA* module in the past have removed this dependency, so that it does not exist anymore in the current version of the system.

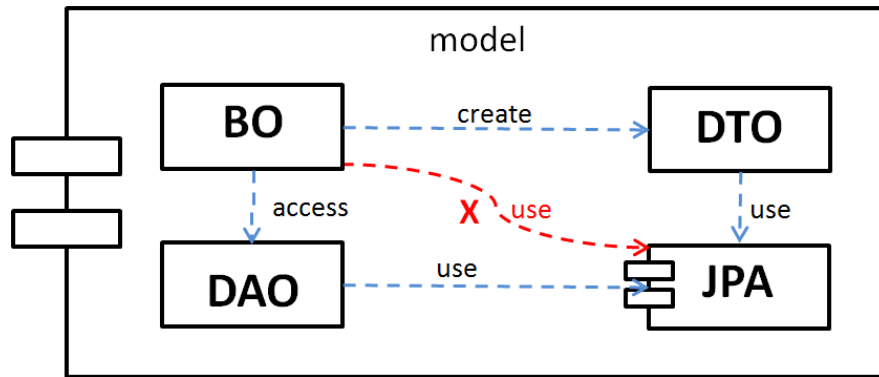


Figure 5.3. Example of divergence (*BO* cannot use *JPA*)

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

```
BaseComponent(bcomp),BaseModule(bmod),BaseClass(bclass)
[,TargetComponent(tcomp),TargetModule(tmod),TargetClass(tclass)]*
```

Using the 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 *domain* component frequently do not depend on *HttpServlet*.

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

Therefore, classes in *domain* that depend on *HttpServlet* represent an evidence of divergence.

In the second step, we rely on 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 association rules representing what we decide to call the *dependency evolutionary patterns*. For instance, suppose a dependency evolutionary pattern as follows:

```
{BaseComponent('domain'),
  TargetClass('HttpServlet'),
  AddAny(true)}=> {ExistCurrently(false)}
```

This pattern states that classes in the component *domain* that added a dependency with the class *HttpServlet* (attribute *AddAny* = *true*) frequently removed this dependency, so that it no longer exist in the current version of the system (attribute *ExistCurrently* = *false*).

Finally, these results are combined with the results obtained in the first step. For instance, suppose that it was previously inferred that the classes in the *domain* component that depend on *HttpServlet* represent evidences of divergences. Moreover, suppose that in this second step we discovered that such classes frequently were refactored to remove the dependencies with *HttpServlet*. In this case, the evidence detected in the first step is reinforced by this second finding.

5.3 Evaluation

To evaluate the data mining based approach for detecting absences and divergences, we performed a study with the SGA system, i.e., the same system described in Section 4.1. The last revision considered in our study has 1,852 classes and interfaces, organized in 104 packages, comprising around 127 KLOC.

5.3.1 Dataset

To detect absences and divergences, we initially retrieved 4,923 revisions of the SGA system, which are 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 5.2.1 and 5.2.2. Finally, the architect of the SGA system inspected the selected violations in order to classify them as true or false positives.

5.3.2 Thresholds for Absences

As reported in Section 5.2.1, 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 the historical dependency evolutionary patterns. Table 5.1 shows the values used for such thresholds:

Table 5.1. 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 have a confidence of at least 90%. For the architectural evolutionary patterns, we consider thresholds of 10% for support and 60% for confidence. In other words, we consider as evidences of architectural violation classes that do not respect a rule followed by at least 10% of the other classes, among all classes in the system. Furthermore, only classes whose historical rules have a confidence higher than 60% are considered as violations.

5.3.3 Thresholds for Divergences

The detection of divergences relies on four thresholds: D_{dps} and D_{dpc} , denoting respectively the support and confidence of the structural architectural patterns, and D_{deps} and D_{depc} , denoting respectively the support and confidence of the historical patterns. Table 5.2 shows the thresholds values used for divergences.

Similarly to the absence detection, for divergences we consider architectural patterns with support of 10% and confidence of 90%. On the other hand, for the historical patterns, we consider the thresholds of 10% and 25% for support and confidence, respectively. Furthermore, we select as divergences the classes that violate both patterns. More specifically, we select classes that depend on a class when at least

Table 5.2. Divergences thresholds

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

90% of the classes in the same component do not follow this rule. Additionally, at least 25% of the classes that have established this dependency were later refactored to remove it.

5.3.4 Results

We applied our methodology in the SGA system using the thresholds defined in Sections 5.3.2 and 5.3.3. The triggered violations were inspected by the SGA architect, who classified them as true or false violations.

As we can observe in Table 5.3, we detected 261 evidences of absences, and 101 evidences 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 49.3% for absences and divergences, respectively. As total, the architect inspected 334 warnings, which 137 were considered true-positives, resulting in a global precision of 41.0%.

Table 5.3. Architectural violations in the 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%	49.3%	41.0%

5.4 Discussion

In this chapter, we conducted an exploratory study to explore the feasibility of using frequent itemset mining techniques to detect architectural violations. As reported in Section 5.3, this evaluation was conducted using the same information system considered in the evaluation of the heuristic-based techniques proposed in Chapter 3. As reported, we achieved an overall precision of 86.7% for absences and 56.5% for

divergences, when using the proposed heuristics. On the other hand, the frequent itemset mining based approach achieved an overall precision of 38.7% and 49.3% for absences and divergences, respectively.

Therefore, our heuristic-based technique achieved a higher precision than frequent itemset mining both for absences and divergences. Next, we compare the two techniques in more details, by discussing the pros and cons of the data mining approach investigated in this chapter.

Pros: The data mining based methodology has the following advantages:

- It allows to detect complex patterns of dependencies, formed by multiple classes, independently on any prior knowledge of software architecture or on the architectural scenarios where violations frequently occur. In other words, frequent itemset mining techniques detect co-occurrence patterns among items in a dataset of existing transactions (according to support and confidence thresholds). For instance, as stated previously, it is common in a system using *JPA* persistence framework, that classes that use the *Entity* annotation also use the annotation *Id* on an attribute of type *Long*. Therefore, the use of *Entity* without *Id*, or vice-versa, might denote an architectural anomaly. Furthermore, the use of *Id* on an attribute that is not of type *Long*, might also represent an anomaly.
- We can rely on association rules generated by frequent itemset mining to reveal architectural patterns. These patterns can be used as documentation artifacts, supporting and guiding the development team on understanding the dependencies between classes, modules, and components of the system. For instance, consider this association rule: $\{\text{BaseComp}(\text{'domain'}), \text{TargetClass}(\text{'Entity'})\} \Rightarrow \{\text{TargetClass}(\text{'Id'})\}$. This rule prescribes that classes in the component *Domain* that depend on the class *Entity* should also depend on class *Id*.
- We can consider other information in the conformance analysis, such as historical information, information on the components of the system, information on the structure and hierarchy of modules, class name patterns, direct dependencies, and indirect dependencies (e.g., based on inheritance rules). For example, consider this association rule: $\{\text{BaseComp}(\text{'domain'}), \text{ClassNamePart}(\text{'Impl'}), \text{PackageNamePart}(\text{'dao.impl'})\} \Rightarrow \{\text{TargetClass}(\text{'Entity'}), \text{TargetClass}(\text{'Id'})\}$. This rule prescribes that classes in the component *Domain*, whose name contains the substring “Impl”, and whose package name contains the substring “dao.impl” must depend on classes *Entity* and *Id*.

- It is feasible to consider not only usage but also non-usage patterns regarding dependencies. For example, we can infer that classes of a particular component *rarely* use the class `java.sql.Statement`, and that such classes *frequently* depend on `javax.persistence.Query`. More important, these dependencies are usually mutually exclusive, i.e., classes that depend on `java.sql.Statement` do not depend on `javax.persistence.Query` and vice-versa. The following association rule illustrates this example: `{TargetClass('javax.persistence.Query')=true} => {TargetClass('java.sql.Statement')=false}`.

Cons: However, a data mining based methodology has the following disadvantages:

- A frequent itemset mining strategy only considers frequency hypothesis, by means of the *support* and *confidence* thresholds described in Section 2.5. Therefore, heuristic-based techniques take advantage over frequent itemset mining based ones because the latter do not rely on any previous knowledge on the problem domain. This condition may contribute to heuristics-based approaches present greater precision than data mining based approaches.
- Our initial investigation indicated the need of further improvements in the performance of the data mining technique. To illustrate this fact, considering the dataset described in Section 5.3.1, that uses only direct dependencies between classes, the heuristic-based technique required 33.2 minutes to display the architectural violations reports. On the other hand, the frequent itemset mining based technique required 495.8 minutes to display its results (14.9 times slower).
- Specially for low support and confidence thresholds, a data mining algorithm may require an extremely high amount of memory and produce a large number of association rules. As an example, using the support and confidence thresholds presented in Section 5.3, 174,737 association rules are produced for absences and 1,752,365 for divergences.

5.5 Final Remarks

In many scenarios it is necessary to discover how often two or more items of interest occur simultaneously or the relationships between co-occurrences of items. It is well-known that data mining-based techniques are suitable when you want to discover how often two or more items from a set occur simultaneously. Furthermore, it is possible

to identify multiple dependencies among these items. Therefore, in this chapter, we investigated a data mining based approach that can be applied to analyze structural and historical architectural patterns among classes.

We conducted an evaluation in the architecture of the SGA's system, when we detected 137 architectural violations, with an overall precision of 41.0%. As can be observed, our heuristic-based approach achieved a higher precision than the data mining based approach, both for absences and divergences.

In Section 5.4, we discussed the pros and cons of the data mining based approach. Despite the disadvantages, we consider that frequent itemset mining is a promising technique and should be more deeply evaluated. Particularly, frequent itemset mining techniques can make use of a broad source of information, which can help to generate more complete architectural patterns.

Chapter 6

Conclusion

This chapter is organized as follows. We start by summarizing the results of this thesis (Section 6.1). Next, we review our contributions (Section 6.2). Additionally, we also indicate the limitations of our proposed architecture conformance process (Section 6.3). Finally, we present further work (Section 6.4).

6.1 Summary

Architectural conformance checking is a fundamental activity for controlling the quality of software systems. This activity aims to reveal deviations between the actual and planned software architectures [Passos et al., 2010]. However, the application of the current techniques for architecture conformance usually requires a considerable effort [Knodel et al., 2008, Passos et al., 2010]. Specifically, reflexion models may require successive refinements in the high-level model to reveal the whole spectrum of absences and divergences in large and extensively maintained systems [Koschke and Simon, 2003]. On the other hand, domain-specific languages may require the extensive and detailed definition of constraints.

To address this shortcoming, this thesis proposed an architecture conformance technique that relies on a combination of static and historical source code analysis to produce evidences of absences and divergences. We provide an iterative technique for architecture conformance checking that does not require successive refinements in high-level architectural models neither requires the specification of an extensive list of architectural constraints. We also designed and implemented an open-source tool called ArchLint that supports our approach and hence reveals architectural erosion symptoms in Java systems. Additionally, we conducted an evaluation of the proposed conformance checking process in four real-world systems, which provided us a positive

feedback on the applicability of our approach. In this evaluation, ArchLint was able to indicate 884 true violations, with overall precision results ranging from 53.3% to 86.7%.

6.2 Contributions

We present the main contributions of our research both for practitioners and for software engineering researchers. First, for practitioners, especially the ones who are not experts on the system under evaluation, we envision that a 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 (e.g., FindBugs, PMD, etc.), ArchLint can be promoted as a complementary tool that elevates to an architectural level the warnings typically raised by such tools. Finally, for researchers the approach proposed in this thesis 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.

Specifically, this research provides the following contributions:

- We provide a review of the state-of-the-art and state-of-the-practice with respect to automatic anomaly detection in source code, architectural conformance checking approaches, and data mining techniques with potential application to architectural violation detection (Chapter 2);
- We introduces an alternative and iterative technique to architectural conformance checking based on a combination of static and historical source code analysis (Chapter 3). This technique includes four heuristics for detecting absences and divergences. It also includes a ranking strategy for ordering the produced warnings according to their probability to denote true architectural violations.
- We implemented a prototype tool called ArchLint that implements our approach and hence provides architectural violation evidences (Section 3.6).
- We evaluated the use of the proposed iterative architectural conformance checking process in four real-world systems (Chapter 4).

- We provide an exploratory study on applying data mining techniques to mine architectural patterns (Chapter 5). We also implemented a prototype tool called ArchLintMiner that supports this data mining-based approach.

6.3 Limitations

Our work has the following limitations:

- The proposed approach may miss true warnings when the system under evaluation is facing a major erosion process. This scenario may cause a relevant impact on the structural and historical functions, such as *DepScaRate*, *DepInsRate*, and *DepDelRate*;
- The proposed approach assumes that violations are usually detected and fixed. This assumption recommends its use especially in mature systems. A possible workaround in less mature systems is to rely on flexible thresholds, e.g., $DepInsRate = 0.0$ and $DepDelRate = 0.0$;
- The proposed heuristics do not cover the entire spectrum of scenarios where architectural violations may occur. They are based on the best of our knowledge and our experience in software architecture conformance. Moreover, they only consider direct dependencies among classes, modules, and components;
- We have not evaluated our approach in scenarios where it is not possible to map the classes of the target system to their respective components through regular expressions;
- We have not measured recall for the SGA and the M2M systems because a detailed inspection in the code is required to find the full set of absences and divergences, including not only true positives, but also false negatives;
- We have not evaluated our ranking strategy using advanced ranking techniques, such as giving weights to different elements as proposed by Engler et al. [Engler et al., 2001b] or using correlation rankings [Kremenek et al., 2004].

6.4 Further Work

The heuristic-based approach for architecture conformance proposed in this thesis must be complemented by the following future work:

- By evaluating other systems to refine and to investigate new heuristics, and to demonstrate the application of our approach in other contexts, using different architectural patterns;
- By working on the integration of ArchLint with ArchFix [Terra et al., 2013], which is a recommendation tool that suggests refactorings for repairing architectural anomalies triggered by static architecture conformance checking approaches;
- By working on the integration of ArchLint with a Domain-Specific Language, such as DCL [Terra et al., 2013].
- By using well-know high quality systems to retrieve architectural patterns, which are more likely to be correct. Once this patterns are available, we can compare other system against them, which should be used to evaluate systems with a small repository of versions.

The data mining-based approach can be complemented by the following future work:

- By investigating other techniques for detecting common patterns of structural dependencies, such as formal concept analysis [Ganter and Wille, 1999];
- By conducting a sensitivity analysis to discover the best combination of values for the thresholds required by data mining algorithms;
- By evaluating its usage on systems using architectural patterns different from the one followed by the SGA system;
- By extending the study to consider specific correlations between dependencies as well as to consider the dependency types, such as attributes, annotations, inheritance, etc.

Bibliography

- [Agrawal et al., 1993] Agrawal, R., Imieliński, T., and Swami, A. (1993). Mining association rules between sets of items in large databases. In *International Conference on Management of Data (MOD)*, pages 207–216.
- [Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499.
- [Allen, 1997] Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.
- [Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.
- [Araujo et al., 2011] Araujo, J. E., Souza, S., and Valente, M. T. (2011). Study on the relevance of the warnings reported by Java bug-finding tools. *IET Software*, 5(4):366–374.
- [Artho and Biere, 2001] Artho, C. and Biere, A. (2001). Applying static analysis to large-scale, multi-threaded Java programs. In *13th Australian Conference on Software Engineering (ASWEC)*, pages 68–75.
- [Baeza-Yates and Ribeiro-Neto, 2011] Baeza-Yates, R. and Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology Behind Search*. Addison-Wesley Professional, 2nd edition.
- [Baldwin and Clark, 1999] Baldwin, C. Y. and Clark, K. B. (1999). *Design Rules: The Power of Modularity*. MIT Press.

- [Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley, 2nd edition.
- [Bell, 1999] Bell, T. (1999). The concept of dynamic analysis. *Software Engineering Notes*, 24(6):216–234.
- [Birkhoff, 1940] Birkhoff, G. (1940). *Lattice Theory*. American Mathematical Society.
- [Bittencourt, 2010] Bittencourt, R. A. (2010). Conformance checking during software evolution. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 289–292.
- [Bittencourt, 2012] Bittencourt, R. A. (2012). *Enabling Static Architecture Conformance Checking of Evolving Software*. PhD thesis, Universidade Federal de Campina Grande.
- [Brito et al., 2013] Brito, H., Marques-Neto, H., Terra, R., Rocha, H., and Valente, M. T. (2013). On-the-fly extraction of hierarchical object graphs. *Journal of the Brazilian Computer Society*, 19(1):15–27.
- [Chang et al., 2007] Chang, R.-Y., Podgurski, A., and Yang, J. (2007). Finding what’s not there: a new approach to revealing neglected conditions in software. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 163–173.
- [Clements, 2003] Clements, P. (2003). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.
- [Copeland, 2005] Copeland, T. (2005). *PMD Applied*. Centennial Books.
- [Couto et al., 2013] Couto, C., Maffort, C., Garcia, R., and Valente, M. T. (2013). COMETS: A dataset for empirical research on software evolution using source code metrics and time series analysis. *ACM SIGSOFT Software Engineering Notes*, pages 1–3.
- [Couto et al., 2012] Couto, C., Montandon, J. E., Silva, C., and Valente, M. T. (2012). Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, pages 1–17.
- [Darwin, 1988] Darwin, I. F. (1988). *Checking C Programs with Lint*. O’Reilly.
- [Davey and Priestley, 2002] Davey, B. and Priestley, H. (2002). *Introduction to Lattices and Order*. Cambridge University Press.

- [de Moor, 2007] de Moor, O. (2007). Keynote address: .QL for source code analysis. In *7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 3–14.
- [Ducasse et al., 2011] Ducasse, S., Anquetil, N., Bhatti, M. U., Hora, A., Laval, J., and Girba, T. (2011). MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, RMOD - INRIA Lille - Nord Europe, Software Composition Group - SCG.
- [Ducasse and Pollet, 2009] Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.
- [Eichberg et al., 2008] Eichberg, M., Kloppenburg, S., Klose, K., and Mezini, M. (2008). Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering (ICSE)*, pages 391–400.
- [Engler et al., 2001a] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001a). Bugs as deviant behavior: a general approach to inferring errors in systems code. *Operating Systems Review*, 35(5):57–72.
- [Engler et al., 2001b] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001b). Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72.
- [Evans, 1996] Evans, D. (1996). Static detection of dynamic memory errors. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 44–53.
- [Evans et al., 1994] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. (1994). LCLint: a tool for using specifications to check code. In *2nd Symposium on Foundations of Software Engineering (FSE)*, pages 87–96.
- [Fayyad et al., 1996] Fayyad, U. M., Piatetsky-Shapiro, G., and Smyth, P. (1996). Advances in knowledge discovery and data mining. pages 1–34.
- [Feiler, 2014] Feiler, P. H. (2014). Aadl and model-based engineering. *Ada Lett.*, 34(3):17–18.
- [Fowler, 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

- [Fowler, 2002] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- [Frawley et al., 1992] Frawley, W. J., Piatetsky-Shapiro, G., and Matheus, C. J. (1992). Knowledge discovery in databases: an overview. *AI Magazine*, 13(3):57–70.
- [Ganter and Wille, 1999] Ganter, B. and Wille, R. (1999). *Formal concept analysis: mathematical foundations*. Springer.
- [Garcia et al., 2009] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Identifying architectural bad smells. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 255–258.
- [Garlan, 2000] Garlan, D. (2000). Software architecture: a roadmap. In *Conference on The Future of Software Engineering (COFES)*, 22nd International Conference on Software Engineering (ICSE), pages 91–101.
- [Garlan et al., 1994] Garlan, D., Allen, R., and Ockerbloom, J. (1994). Exploiting style in architectural design environments. In *2nd Symposium on Foundations of Software Engineering (FSE)*, pages 175–188.
- [Garlan et al., 1997] Garlan, D., Monroe, R., and Wile, D. (1997). ACME: an architecture description interchange language. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 1–15.
- [Garlan and Shaw, 1996] Garlan, D. and Shaw, M. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.
- [Gorton and Zhu, 2005] Gorton, I. and Zhu, L. (2005). Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *27th International Conference on Software Engineering (ICSE)*, pages 514–523.
- [Gruska et al., 2010] Gruska, N., Wasylkowski, A., and Zeller, A. (2010). Learning from 6,000 projects: lightweight cross-project anomaly detection. In *19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–130.
- [Gurgel et al., 2014] Gurgel, A., Macia, I., Garcia, A., von Staa, A., Mezini, M., Eichberg, M., and Mitschke, R. (2014). Blending and reusing rules for architectural degradation prevention. In *13th International Conference on Modularity*, pages 61–72.

- [Götzmann, 2007] Götzmann, D. N. (2007). Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen., Bachelor thesis, Saarland University, Available from <http://code.google.com/p/colibri-java/> (accessed July 06, 2012).
- [Hamou-Lhadj and Lethbridge, 2004] Hamou-Lhadj, A. and Lethbridge, T. C. (2004). A survey of trace exploration tools and techniques. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 42–55.
- [Han et al., 2000] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *19th International Conference on Management of Data (SIGMOD)*, pages 1–12.
- [Hochstein and Lindvall, 2005] Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: A survey. *Information and Software Technology*, 47(10):643–656.
- [Hou and Hoover, 2006] Hou, D. and Hoover, H. J. (2006). Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423.
- [Hou et al., 2004] Hou, D., Hoover, H. J., and Rudnicki, P. (2004). Specifying framework constraints with FCL. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 96–110.
- [Hovemeyer and Pugh, 2004] Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106.
- [Jerding and Rugaber, 1997] Jerding, D. and Rugaber, S. (1997). Using visualization for architectural localization and extraction. In *4th Working Conference on Reverse Engineering (WCRE)*, pages 56–65.
- [Johnson, 1977] Johnson, S. C. (1977). Lint: A C program checker. Technical report 65, Bell Laboratories.
- [Kazman and Carrière, 1999] Kazman, R. and Carrière, S. J. (1999). Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138.
- [Knodel et al., 2008] Knodel, J., Muthig, D., Haury, U., and Meier, G. (2008). Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

- [Knodel et al., 2006] Knodel, J., Muthig, D., Naab, M., and Lindvall, M. (2006). Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294.
- [Knodel and Popescu, 2007] Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 44–54.
- [Koschke and Simon, 2003] Koschke, R. and Simon, D. (2003). Hierarchical reflexion models. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 36–45.
- [Kremenek et al., 2004] Kremenek, T., Ashcraft, K., Yang, J., and Engler, D. (2004). Correlation exploitation in error ranking. In *12th Symposium on Foundations of Software Engineering (FSE)*, pages 83–93.
- [Li and Zhou, 2005] Li, Z. and Zhou, Y. (2005). PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *13th Symposium on Foundations of Software Engineering (FSE)*, pages 306–315.
- [Macia et al., 2012] Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., and von Staa, A. (2012). Supporting the identification of architecturally-relevant code anomalies. In *28th International Conference on Software Maintenance (ICSM)*, pages 662–665.
- [Maffort et al., 2013a] Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013a). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231.
- [Maffort et al., 2012] Maffort, C., Valente, M. T., and Bigonha, M. (2012). Detecção de violações arquiteturais usando histórico de versões. In *XI Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15.
- [Maffort et al., 2013b] Maffort, C., Valente, M. T., Bigonha, M., Hora, A., and Anquetil, N. (2013b). Mining architectural patterns using association rules. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 375–380.
- [Maffort et al., 2013c] Maffort, C., Valente, M. T., Bigonha, M., Silva, L. H., and Aparecido, G. (2013c). ArchLint: Uma ferramenta para detecção de violações arquiteturais usando histórico de versões. In *IV Congresso Brasileiro de Software: Teoria e Prática (Sessão de Ferramentas)*, pages 1–6.

- [Maffort et al., 2014] Maffort, C., Valente, M. T., Terra, R., Bigonha, M., Anquetil, N., and Hora, A. (2014). Mining architectural violations from version history. *Empirical Software Engineering Journal (EMSE)*, pages 1–41.
- [Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *5th European Software Engineering Conference (ESEC)*, pages 137–153.
- [Mens et al., 2006] Mens, K., Kellens, A., Pluquet, F., and Wuyts, R. (2006). Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3):140–156.
- [Mileva et al., 2011] Mileva, Y. M., Wasylkowski, A., and Zeller, A. (2011). Mining evolution of object usage. In *25th European Conference on Object-Oriented Programming (ECOOP)*, pages 105–129.
- [Murphy et al., 1995] Murphy, G., Notkin, D., and Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28.
- [Murphy et al., 2001a] Murphy, G., Notkin, D., and Sullivan, K. (2001a). Software reflexion models. *IEEE Transactions on Software Engineering*, 27(4):364–380.
- [Murphy et al., 2001b] Murphy, G., Notkin, D., and Sullivan, K. J. (2001b). Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27:364–380.
- [Nierstrasz et al., 2005] Nierstrasz, O., Ducasse, S., and Gırba, T. (2005). The story of moose: an agile reengineering environment. In *European software engineering conference held jointly with the ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13*, pages 1–10.
- [Passos et al., 2010] Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonca, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.
- [Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52.
- [Pressman, 2010] Pressman, R. (2010). *Software engineering: a practitioner’s approach*. McGraw-Hill.

- [Priss, 2006] Priss, U. (2006). Formal concept analysis in information science. *Annual Review of Information Science and Technology (ARIST)*, 40(1):521–543.
- [Rocha et al., 2013] Rocha, H., Couto, C., Maffort, C., Garcia, R., Simoes, C., Passos, L., and Valente, M. T. (2013). Mining the impact of evolution categories on object-oriented metrics. *Software Quality Journal*, 21(4):529–549.
- [Sangal et al., 2005] Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176.
- [Santos et al., 2014] Santos, G., Valente, M. T., and Anquetil, N. (2014). Remodularization analysis using semantic clustering. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 224–233.
- [Sarkar et al., 2009a] Sarkar, S., Maskeri, G., and Ramachandran, S. (2009a). Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Software*, 82:1891–1905.
- [Sarkar et al., 2009b] Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K., and Sivagnanam, S. (2009b). Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35.
- [Schmerl et al., 2006] Schmerl, B. R., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006). Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466.
- [Silva et al., 2014] Silva, L., Valente, M. T., and Maia, M. (2014). Assessing modularity using co-change clusters. In *13th International Conference on Modularity*, pages 49–60.
- [Sullivan et al., 2001] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108.
- [Tan et al., 2002] Tan, P.-N., Kumar, V., and Srivastava, J. (2002). Selecting the right interestingness measure for association patterns. In *8th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 32–41.

- [Terra and Valente, 2008] Terra, R. and Valente, M. T. (2008). Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 256–263.
- [Terra and Valente, 2009] Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094.
- [Terra et al., 2013] Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2013). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–36.
- [van Gorp and Bosch, 2002] van Gorp, J. and Bosch, J. (2002). Design erosion: problems and causes. *Journal of Systems and Software*, 61:105–119.
- [Wasylkowski and Zeller, 2009] Wasylkowski, A. and Zeller, A. (2009). Mining temporal specifications from object usage. In *24th International Conference on Automated Software Engineering (ASE)*, pages 295–306.
- [Wasylkowski et al., 2007] Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *15th Symposium on The Foundations of Software Engineering (FSE)*, pages 35–44.
- [Wille, 2009] Wille, R. (2009). Restructuring lattice theory: an approach based on hierarchies of concepts. In *7th International Conference on Formal Concept Analysis (IFCA)*, pages 314–339.
- [Zaki and Meira Jr., 2011] Zaki, M. J. and Meira Jr., W. (2011). *Fundamentals of Data Mining Algorithms*. Cambridge University Press.
- [Zaki et al., 1997] Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W. (1997). New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 283–286.
- [Zimmermann et al., 2004] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE)*, pages 563–572.
- [Zimmermann et al., 2005] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.