Extracting Implemented Module Dependencies with the ArR Tool

Christian Schmitz[1], **Gabriel Alves**[1], **Vanius Zapalowski**[1], **Ingrid Nunes**[1,2], **Daltro José Nunes**[1]

[1]Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

[2]TU Dortmund, Dortmund, Germany

{christian.schmitz,goalves,vzapalowski,ingridnunes,daltro}@inf.ufrgs.br

Abstract. *Software architecture is fundamental to the development of any software system. Nevertheless, architecture documentation is commonly outdated or nonexistent. Reverse engineering techniques to recover software architecture emerged from the need for supporting architecture documentation of existing systems, which is a task that demands significant effort if done entirely manually. We thus developed the ArR tool, an Eclipse plug-in that provides software engineers with support to recover architecture rules based on implemented source code dependencies. Our tool relies on the use of a data mining technique applied to (sub-)modules dependencies extracted from source files of Java projects providing a measurement and a visualization of such dependencies.*
Video: `https://youtu.be/WEqDVX5Q0Pk`

1. Introduction

Software architecture has a major role in guiding the software development process to success. However, architectural documentation is commonly nonexistent or outdated [Lutellier et al. 2015]. Without proper architectural documentation, developers tend to evolve software in an uncontrolled way, which decreases software quality and, consequently, increases maintenance costs [Su and Yeh 2012]. Software architecture recovery, i.e. extracting and documenting the architecture of implemented software, thus should be performed to make architectural information available and avoid these problems.

The task of recovering software architecture is complex and demands project-specific knowledge because architectural modules must be identified as well as how they can or should communicate (i.e. architectural rules) [Ducasse and Pollet 2009]. Software architecture recovery methods aim to support the specification of architectural modules and rules by automating (part of) these activities, such as grouping software elements (e.g. classes) with similar names so as to form a (sub-)module. However, these activities are still time-consuming and error-prone given that they still require expert validation.

Many approaches and tools for software architecture recovery have been proposed, but most of them focus on providing inspection of architectural conformance. Furthermore, few approaches were developed as part of an integrated development environment (IDE), which facilitates their use as part of the software development process. We thus developed the *ArR (Architectural Rule Recovery)* tool, aiming to support the documentation of architectural rules based on implemented dependencies between modules. Our tool is an Eclipse plug-in that adds a new Eclipse *view* with a set of architectural support features, which help developers identify implemented architectural rules. With ArR, it is possible to analyze source code dependencies to derive representative module dependencies using data mining techniques, reducing the effort needed to recover software architecture rules.

We next detail the ArR tool, presenting its main features and how they are implemented. In Section 3, we discuss existing tools related to supporting architectural rule recovery and extraction of source code dependencies. Finally, we present final remarks and future work in Section 4.

2. The ArR Tool

ArR is a tool, implemented as an Eclipse plugin, to support architectural rule recovery of software written in Java. Our tool extracts implemented source code dependencies and analyzes them using a data mining algorithm to automatically extract implemented module dependencies. These dependencies are relationships between modules (which in Java should correspond to packages), in the form `A` $\rightarrow$ `B`, meaning that the module `A` *can or must depend on* module `B`. This dependency relationship must occur not occasionally, but with a certain frequency. Consequently, implemented dependencies indicate architectural rules adopted in the code, excluding unusual dependencies, which are considered forbidden dependencies and, consequently, architecture violations, if they occur.

The extraction and visualization of these dependencies are the main contributions of our tool because these two features reduce the time demanded to analyze recovered dependencies, which may not necessarily be planned or intended—a module that depends on another may not be allowed to do so. Moreover, our tool is complementary to most of the architecture recovery tools because they require specification of architectural rules [Miranda et al. 2016, Melo et al. 2014]. Therefore, ArR's output may be used as input to these other tools. In the remainder of this section, we detail the ArR features, as well as how they are implemented and used external libraries.

2.1. Features and User Interface

In Figure 1, we present an overview of ArR. The textual view of our tool is in the square labeled with (i); its graphical view is the square labeled with (ii); its action buttons are in the smaller square labeled with (iii); the Eclipse package explorer is labeled with (iv), from which menu items from ArR can be accessed; and a window with options to customize how the data mining algorithm should run is labeled with (v). The view consists of a list of implemented module dependencies provided as the output of the execution of the data mining algorithm. This list is composed of three columns: the first column shows dependent modules; the second column shows dependee modules, and the third column shows the support metric [Agrawal and Srikant 1994] (an indication of the dependency intensity). Each row corresponds to a recovered module dependency, with is support value. In Figure 1-i, there are action buttons with functionalities, which are described below according to their appearance, from left to right.

Run ArR in Project. In this functionality, the user is asked to inform an Eclipse project in the package explorer (Figure 1-iv). Our tool runs the architecture recovery process using as input the selected project and presents the extracted module dependencies in the ArR view (Figure 1-i). Our tool uses the implementation of the Apriori [Agrawal and Srikant 1994] algorithm, which is an association rule mining algorithm provided by Sequential Pattern Mining Framework (SPMF) library.

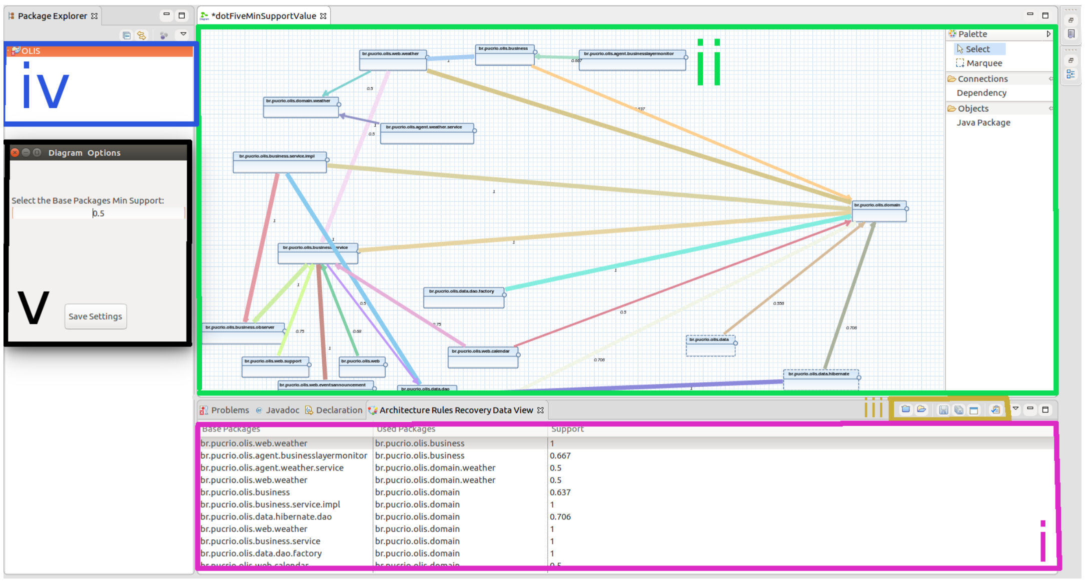


Diagram Options

Select the Base Packages Min Support: 0.5

Base Packages	Used Packages	Support
br.pucrio.olis.web.weather	br.pucrio.olis.business	1
br.pucrio.olis.agent.businesslayermonitor	br.pucrio.olis.business	0.667
br.pucrio.olis.agent.weather.service	br.pucrio.olis.domain.weather	0.5
br.pucrio.olis.web.weather	br.pucrio.olis.domain.weather	0.5
br.pucrio.olis.business	br.pucrio.olis.domain	0.637
br.pucrio.olis.business.service.impl	br.pucrio.olis.domain	1
br.pucrio.olis.data.hibernate.dao	br.pucrio.olis.domain	0.706
br.pucrio.olis.web.weather	br.pucrio.olis.domain	1
br.pucrio.olis.business.service	br.pucrio.olis.domain	1
br.pucrio.olis.data.dao.factory	br.pucrio.olis.domain	1
br.pucrio.olis.web.calendar	br.pucrio.olis.domain	0.5

Figure 1. ArR Main Screen.

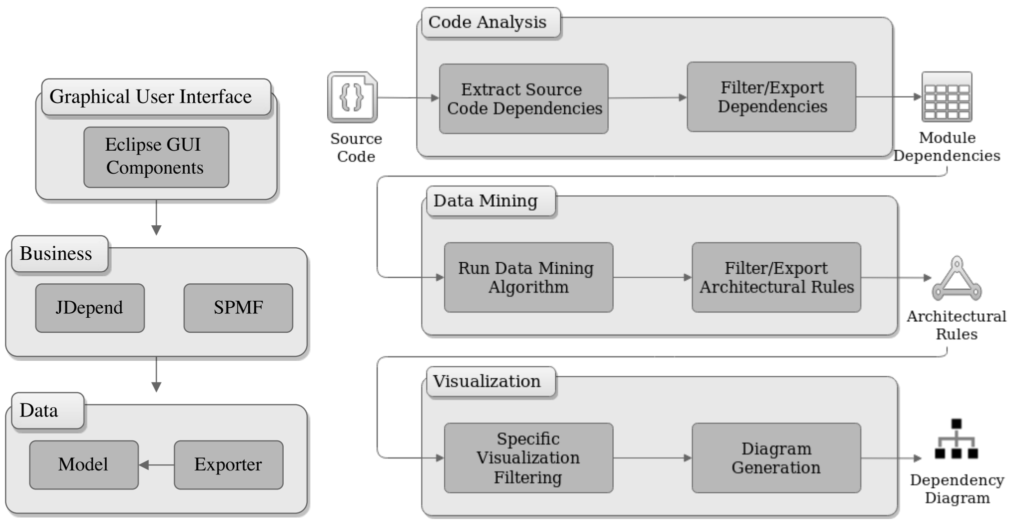
Export Implemented Source Code Dependencies. This functionality allows exporting implemented source code dependencies (among fine-grained elements, such as classes) used as input by the data mining algorithm in a Comma Separated Values (CSV) file. With this data available in a separated file, it can be used by experts to calibrate algorithm parameters used to derive architectural rules.

Export Module Dependencies. This functionality allows exporting every module dependencies displayed in the ArR textual view to a CSV file following the column order of the ArR view. This information can be used as input to architecture conformance check approaches [Murphy et al. 2001] or to tools that provide graphical visualizations of architectural rules [Zapalowski et al. 2014].

Diagram Generation. In this functionality, a window with a diagram of the module dependencies of the system appears to the user. These diagrams contain the module dependencies generated after running ArR's recovery process. It allows the user to see the implemented module dependencies together in a graphical representation. This diagram is detailed in the Section 2.4.

2.2. Architecture and Implementation

Given that ArR is an Eclipse plug-in, it is written in Java and follows the architecture usually adopted by Eclipse plug-ins, which is illustrated in Figure 2(a). We implemented our tool dividing it into three key modules. The Graphical User Interface (GUI) module is composed of Eclipse graphical components customized to our needs, such as menu options and views presented in Figure 1. The Business module handles the ArR action buttons and most of the architectural rule recovery process, which is detailed next. Finally, the Data module was built based on Eclipse and SPMF [Fournier-Viger et al. 2014] data models to manage source code, module dependencies and data mining objects. This model includes the Exporter module that manipulates data to provide most of the export actions previously mentioned.



(a) Architecture. (b) Architectural Rule Recovery Process.

Figure 2. ArR Layered Architecture and Recovery Process.

2.3. Architectural Rule Recovery Process

The process adopted to perform architectural rule recovery in our tool is based on two steps: (i) the static analysis of source code dependencies; and (ii) the execution of a data mining algorithm to provide module dependencies. An overview of the architectural rule recovery process is represented in Figure 2(b).

2.3.1. Extracting Static Dependencies

The first step to recovering rules consists of extracting static dependencies based on source code available. Our tool manipulates these dependencies to create a dependency matrix, which is used as input by data mining algorithms. To extract these dependencies, we integrated the JDepend[1] tool with our plugin. The JDepend API parses the Java Bytecode searching for class-to-package dependencies. To mine architectural rules, we assume that packages are representations of possible architectural modules, given that they are typically used to organize different system concerns. We abstract the class-to-package dependencies to package-to-package dependencies, according to the class packages, which are then represented as module-to-module dependencies generating an architectural module dependency matrix. Package-to-package dependencies are commonly used to specify architectural rules in architecture recovery approaches [Murphy et al. 2001], because they are coarse-grained and thus more concise to be specified. Moreover, our goal here is to mine rules that are implemented, rather than planned.

Given this dependency matrix, we filter source code dependencies discarding dependencies to modules not available in the project, such as those to external libraries. This matrix is used in the mining architectural dependencies step of our tool. Additionally, it is possible to translate this matrix to a CSV file, which can be exported by the user.

[1]Available at `http://clarkware.com/software/JDepend.html`.

2.3.2. Mining Architectural Dependencies

With the aim of automatically deriving architectural rules, our tool applies a data mining algorithm based on the dependency matrix created in the previous step. The application of such algorithms reduces the amount of module dependencies, thus supporting the analysis of implemented architectural rules providing metrics of each rule usage. This information guides developers in the task of determining which module-to-module dependency is, in fact, an implemented architectural rule (i.e. allowed module dependency). Currently, our tool executes the algorithm named Apriori implemented in the SPMF library. Apriori is an algorithm of association rule, i.e., given a dataset of dependencies, it analyses the frequency of each dependency and provides a metric of the frequency of each dependency, namely support. The support metric measures the frequency of the dependency based on the number of classes that a package has and depend on another package. For instance, assuming that a package A with 10 classes, from which 2 of them depend on package B, a module dependency of $A \rightarrow B\ [support = 20\%]$ can be derived.

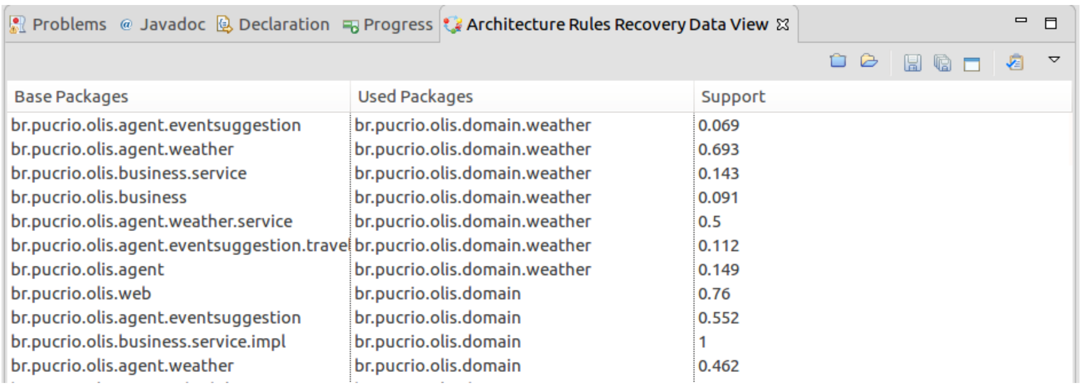
The algorithm output represents the following abstraction: potential architectural rules as a pair of dependent module and dependee modules, and algorithm-specific values (for instance, support in the case of Apriori). As shown in Section 2.1, our output format is a list of implemented dependencies between (sub-)modules consisting of a *Base Package* (TP), *Used Packages* (UP) and the *Support* rate of each dependency mined. Thus, after running data mining algorithm, the algorithm output is formatted to our standard format and presented in the screen as previously explained in Section 2.1. As in the Extracting Static Dependencies step, it is possible to export results to a CSV file.

2.4. Use Case

An example of a scenario in which the ArR tool is helpful to developers is when they must implement a new feature in software, but there is no available documentation associated with the software architecture indicating allowed module dependencies. ArR thus helps in the task of recovering this project information to show implemented architectural module dependencies to developers to help them implementing the new feature in conformance with its architecture. Alternatively, our tool can be used to refactor the code, because the current implementation patterns adopted in the code, shown by the extracted rules, may be inadequate according to software engineering principles.

We present the ArR first usage scenario with OnLine Intelligent Services (OLIS) source code as an example. OLIS is a web-based application to handle personal calendar and events. It has 11.4 KLOC, 211 classes, and 45 packages. The outputs that are shown in Figure 3 were obtained using the tool with the ArR source code, setting the minimum support (algorithm-specific parameter) value to 5%, i.e. our tool ran the Apriori algorithm considering that each selected implemented architectural rule must occur at least in 5% of the cases, in the module-to-module dependency matrix. Each line in the ArR view represents an architectural rule and its support value.

The results presented in Figure 3 show 11 implemented module dependencies (11 lines in ArR view). To explain extracted implemented dependencies, we detail the dependencies in lines 1 and 2. The module dependency of line 1 is associated with `olis.agent.eventsuggestion` and `olis.domain.weather` modules. It has a support of 6.9% because only two classes of `olis.agent.eventsuggestion`,



Base Packages	Used Packages	Support
br.pucrio.olis.agent.eventsuggestion	br.pucrio.olis.domain.weather	0.069
br.pucrio.olis.agent.weather	br.pucrio.olis.domain.weather	0.693
br.pucrio.olis.business.service	br.pucrio.olis.domain.weather	0.143
br.pucrio.olis.business	br.pucrio.olis.domain.weather	0.091
br.pucrio.olis.agent.weather.service	br.pucrio.olis.domain.weather	0.5
br.pucrio.olis.agent.eventsuggestion.trave	br.pucrio.olis.domain.weather	0.112
br.pucrio.olis.agent	br.pucrio.olis.domain.weather	0.149
br.pucrio.olis.web	br.pucrio.olis.domain	0.76
br.pucrio.olis.agent.eventsuggestion	br.pucrio.olis.domain	0.552
br.pucrio.olis.business.service.impl	br.pucrio.olis.domain	1
br.pucrio.olis.agent.weather	br.pucrio.olis.domain	0.462

Figure 3. Textual output of ArR using as input OLIS filtered by 5%.

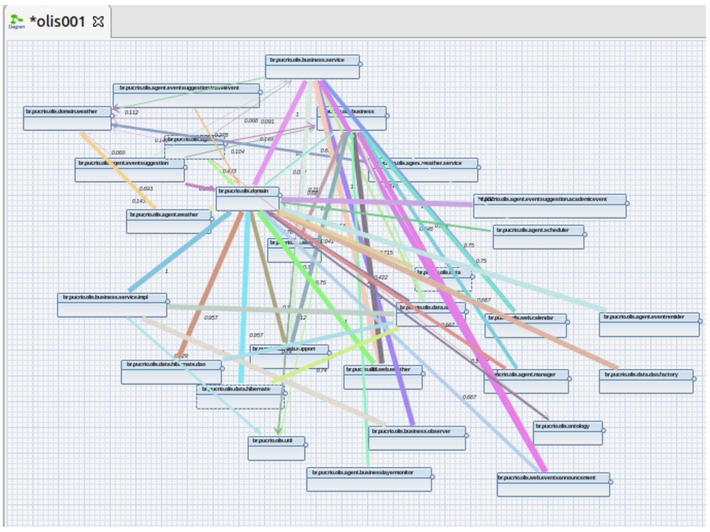
which has 29 classes, use the `olis.domain.weather`. Module dependency of line 2 presents the direct dependency between `olis.agent.weather` $\rightarrow$ `olis.domain.weather` with support of 69.3%. It has 69.3% of support because 9 of the 13 classes of `olis.agent.weather` module use `olis.domain.weather`.

Based on the extracted dependencies presented in Figure 3, developers may conclude that `olis.agent.weather` $\rightarrow$ `olis.domain.weather` is a rule because this module dependency has a high support. On the other hand, the rule `olis.agent.eventsuggestion` $\rightarrow$ `olis.domain.weather` may be an architectural violation due to its low support. Alternatively, it can be a *can depend on* rule, rather than *must depend on*. Thus, developers can use ArR too reason about the relevance of each extracted rule analyzing the rules provided by ArR.

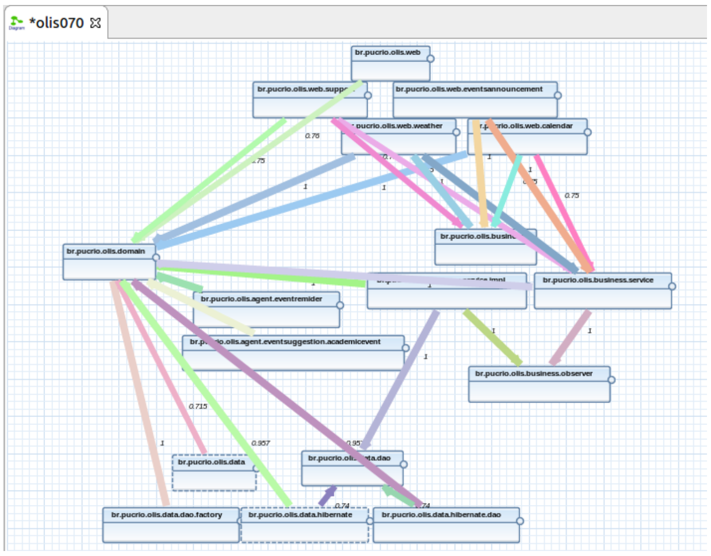
To facilitate the task of understanding the module dependencies of the project, we developed the graphical visualization of the module dependencies presented in the textual view of ArR. We generated two diagrams of OLIS as presented in Figure 4. The intensity of the dependencies presented in these diagrams is represented by their thickness. Figure 4(a) gives the module dependencies with support greater than 1%. In this diagram, there are many dependencies that hamper the task of understanding the architecture of OLIS. Figure 4(b), in turn, has the module dependencies greater than 70%. This diagram is clearer to present only highly frequent dependencies between modules. Furthermore, given the organization of module, we can see the more important module and sub-modules. We can notice that: (i) the `web` modules are concentrated in the upper right of diagram; (ii) mid-right modules are related to `business` tasks; (iii) the module in the bottom handle `data` activities; (iv) the mid-left module, which is the most used, is the `domain`; and (v) the two modules in the middle, between `business` and `domain`, handle `agent` tasks.

3. Related Work

As discussed, most of the existing approaches in the context of architecture conformance take as input a set of architectural rules, which describe the conceptual architecture, and perform a conformance check between such rules *vs.* the implemented architecture [Ducasse and Pollet 2009, Terra and Valente 2009]. Tools [Sartipi 2001, Gall et al. 1996, Miranda et al. 2016, Melo et al. 2014] have been developed to implement such approaches, but most of them are standalone applications that require much


*olis001
br.pucrio.olis.business.service
br.pucrio.olis.domain
br.pucrio.olis.business.service.impl
br.pucrio.olis.agent.scheduler
br.pucrio.olis.agent.eventreminder
br.pucrio.olis.data.dao.factory
br.pucrio.olis.ontology
br.pucrio.olis.util
br.pucrio.olis.business.observer
br.pucrio.olis.web.eventsannouncement

(a) Support 1%.


*olis070
br.pucrio.olis.web
br.pucrio.olis.web.eventsannouncement
br.pucrio.olis.web.calendar
br.pucrio.olis.domain
br.pucrio.olis.agent.eventremider
br.pucrio.olis.business.service
br.pucrio.olis.business.observer
br.pucrio.olis.data
br.pucrio.olis.data.dao.factory
br.pucrio.olis.data.hibernate
br.pucrio.olis.data.hibernate.dao

(b) Support 70%.

Figure 4. Graphical output of ArR .

effort to manually specify and maintain architectural rules.

Even in the tools that have a set of predefined architectural rules [Sartipi 2001], it is complex to maintain these in conformance with the source code because the code usually evolves and the documentation does not. Thus, the process of listing implemented architectural rules is needed to update the set of conceptual architectural rules.

4. Final Remarks and Future Work

ArR was built with the intent of helping software engineers by decreasing the effort needed to maintain or recover architectural documentation of software systems. Our main contribution is the provision of a tool-supported way of obtaining implemented rules, which are related to the allowed dependencies between modules. Our tool extracts source code dependencies and provides data mining algorithm for deriving implemented architectural rules. Also, a metric associated with dependency intensity is presented to allow developers decide whether the extracted module dependency corresponds to a rule of the type *can* or *must* depend on, or an architecture violation. Unusual dependencies are already discarded by the data mining algorithm, thus considered violations of forbidden dependencies.

Although our tool currently provides useful features to mine architectural rules, presenting source code dependencies related to the implemented architectural rules to developers would help them in the task of identifying architectural violations. Moreover, automating the task of comparing different versions of the system given a predefined saved diagram would provide a support that is needed to keep a system in conformance with its architecture.

Acknowledgements

Ingrid Nunes thanks for research grants CNPq ref. 303232/2015-3, CAPES ref. 7619-15-4, and Alexander von Humboldt, ref. BRA 1184533 HFSTCAPES-P.

References

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *VLDB '94*, pages 487–499.

Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu., C., and Tseng, V. S. (2014). SPMF: a Java Open-Source Pattern Mining Library. *Journal of Machine Learning Research (JMLR)*, 15:3389–3393.

Gall, H., Jazayeri, M., Klsch, R., Lugmayr, W., and Trausmuth, G. (1996). Architecture recovery in ares. In *Int. Software Architecture Workshop*, pages 111–115.

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidović, N., and Kroeger, R. (2015). Comparing software architecture recovery techniques using accurate dependencies. In *ICSE '15*, pages 69–78.

Melo, I., Serey, D., and Valente, M. T. (2014). Uma ferramenta para verificaćão de conformidade visando diferentes percepć ões de arquiteturas de software. In *CBSoft 2014, Tools Track*, pages 1–8.

Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016). Architectukre conformance checking in dynamically typed languages. *Journal of Object Technology*, 1(1):1–35.

Murphy, G., Notkin, D., and Sullivan, K. (2001). Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380.

Sartipi, K. (2001). Alborz: a query-based tool for software architecture recovery. In *IWPC 2001*, pages 115–116.

Su, C.-T. and Yeh, D. (2012). Software architecture recovery and re-documentation tool of a hospital information system. In *ICCCE 2012*, pages 143–146.

Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094.

Zapalowski, V., Nunes, I., and Nunes, D. J. (2014). Archviz: a tool to support architecture recovery research. In *CBSoft 2014, Tools Track*, pages 13–20.