Supporting the Identification of Architecturally-Relevant Code Anomalies

Isela Macia, Roberta Arcoverde, Elder Cirilo, Alessandro Garcia, Arndt von Staa
Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio),
Rio de Janeiro – RJ – Brazil
{ibertran, rarcoverde, ecirilo, afgarcia, arndt}@inf.puc-rio.br

Abstract—**Code anomalies are likely to be critical to the systems' maintainability when they are related to architectural problems. Many tools have been developed to support the identification of code anomalies. However, those tools are restricted to only analyze source code structure and identify individual anomaly occurrences. These limitations are the main reasons why state-of-art tools are often unable to identify architecturally-relevant code anomalies, i.e. those related to architectural problems. To overcome these shortcomings we propose SCOOP, a tool that includes: (i) architecture-code traces in the analysis of the source code, and (ii) exploits relationships between multiple occurrences of code anomalies to detect the architecturally-relevant ones. Our preliminary evaluation indicated that SCOOP was able to detect anomalous code elements related to 293 out of 368 architectural problems found in 3 software systems.**

Keywords-code anomaly; pattern; architectural problem

I. INTRODUCTION

Code anomalies, also known as code smells, are recurring code structures which usually make systems more difficult to understand and maintain [1]. Code anomalies are even more severe to the system maintainability when they are related to one or more architectural problems [5][7] - hereafter called *architecturally-relevant code anomalies*. The reason is that the progressive introduction of architectural problems in the implementation can result in spending a lot of effort even when dealing with simple code changes. In more critical cases, architectural problems can lead to the system discontinuation [5]. Therefore, developers need to be provided with mechanisms that allow them to detect those critical code anomalies as early as possible [7].

However, detecting architecturally-relevant code anomalies is not trivial. Recent studies have shown that looking at individual occurrences of code anomalies is not enough for identifying those that impact on the architectural design [6][7]. The main reason is that a single architecture-level element is usually realized by multiple implementation-level elements. Thus, architectural problems are often associated with inter-related anomalous code elements [7][8]. Additionally, developers cannot distinguish which anomalous code elements are architecturally harmful without considering the role they play in the architectural design.

To support code anomalies detection, a wide range of code analysis tools have been developed [9][11][12][15]. These tools are usually focused on the extraction and combination of static code measures to detect potential maintainability problems. However, such state-of-art tools are limited to support the detection of architecturally-relevant code anomalies. First, they only identify individual code anomalies, without analyzing particular relationships between them. Second, they are restricted to analyzing the source code structure, leading to well-known false positives and false negatives [10], and guiding developers in wrong directions when performing maintenance tasks.

In this paper we present SCOOP [13], a tool that detects architecturally-relevant code anomalies. Developers might benefit from using SCOOP typically when performing maintenance tasks, like refactoring or modifying existing features. By identifying architecturally-relevant code anomalies, developers can invest their refactoring efforts into solving anomalies that could possibly cause architecture problems, improving refactoring effectiveness.

For identifying architecturally-relevant code anomalies, SCOOP's novelty consists in (i) using architecture information, instead of static analysis only, and (ii) analyzing groups of anomalies for identifying patterns of code anomaly occurrences. SCOOP is implemented as an Eclipse plug-in, allowing developers to use it during the implementation of Java systems. Preliminary results of SCOOP's usage indicated that it was able to detect code anomalies related to 293 out of 368 architectural problems found in 3 software systems. These results indicate that our tool can improve the effectiveness of existing ones [9][11][12][15] and help developers prioritize refactorings during maintenance tasks.

II. RELATED WORK

A number of tools have been proposed for identifying code anomalies [9][11][12][15]. To this end, most of these tools rely on the use of detection strategies [10] - expressions composed by metrics and thresholds. Each detection strategy aims at identifying occurrences of a specific code anomaly [10]. However, conventional detection strategies suffer from several shortcomings that hinder their support on the identification of architecturally-relevant code anomalies [6].

The main limitation of conventional detection strategies is that they only consider the source code structure, disregarding architectural design information. Additionally, conventional detection strategies only identify individual anomaly occurrences. That is, they do not analyze structural relationships between code anomalies, which could be indicators of the anomaly's adverse influence on the architecture [6]. These limitations lead conventional detection strategies to not inform developers accurately about the presence of architecturally-relevant code anomalies and, hence, such anomalies could remain indefinitely in the systems.

Another shortcoming that affects the helpfulness of conventional detection strategies is that they report long lists of code anomalies, which are often not considered as threats [6][10][12]. These lists usually cover many parts of the system, without revealing architecturally-relevant code anomalies. Therefore, by employing existing tools, developers may be wasting a lot of time (i) removing anomalies that are not related to architectural threats, and (ii) manually and exhaustively inspecting all the detected code anomalies in order to identify the most relevant ones.

There are other mechanisms that consider history-sensitive information of the system's implementation to identify critical code anomalies [8][16]. They tend to improve the accuracy of conventional detection strategies by analyzing several systems' versions. The problem with these mechanisms is that they cannot detect code anomalies early enough (i.e. in the first system' versions); therefore, their analysis could be performed too late, when removing the critical anomaly might be impeditive [7].

III. SCOOP Tool

The anomaly detection process of SCOOP is based on two novel steps. The first step (Section III.A) comprises the detection of code anomaly occurrences, based on the analysis of traces between architecture elements and its corresponding source code (i.e. architecture-code traces). The second step analyzes different kinds of relationships between code anomalies (e.g. access to the same architectural elements) in order to distinguish which ones are likely to affect more the architecture (Section III.B). The novelty in this step is that SCOOP is not limited to the detection of individual anomalies.

A. Detection of Code Anomalies

SCOOP relies on detection strategies in order to identify code anomaly occurrences, similarly to state-of-art code analysis tools (Section II). However, as mentioned above, detection strategies solely based on source code structure are unable to accurately identify architecturally-relevant code anomalies [6]. Therefore, in SCOOP we enhance the source code analysis by exploiting properties of architecture-code traces. This allows SCOOP to perform more sophisticated queries on the source code than conventional strategies can do and, consequently, improve the effectiveness of state-of-art code analysis tools (Section II).

Two properties of architecture-code traces are currently supported by SCOOP. First, we consider mappings between elements of a component-and-connector view and code elements (e.g. classes). Second, we exploit mappings between architectural concerns and code elements. In this context, an architectural concern is a responsibility or property of a software system that is realized by architecture elements. Both kinds of traces are represented in SCOOP by using Prolog facts [3][14] in two different files, where each file represents a kind of trace. An example of these representations is shown below.

Listing 1. Representation of architecture-code traces in SCOOP.

```
<<component-and-connector mappings>>
mapping('MODEL', class('mp.core.ui.datamodel', 'Album').
mapping('VIEW', class('mp.core.ui.screens, 'SplashScreen').
<<architectural concern mappings>>
mapping('PERSISTENCE', class('mp.core.ui.datamodel', 'Album').
mapping('SERVICE', class('mp.core.ui.screen, 'Midlet').
```

The first trace provides information regarding the architectural role a given code element plays. For example, considering the MVC architecture a code element might belong to three different architecture roles - Model, View and Controller (Listing 1). The analysis of this trace allows SCOOP to identify anomalous code elements that have numerous interactions with adjacent components. Additionally, the analysis of the second trace supplies information related to which architectural concerns a given code element is in charge of realizing - like persistence or integration of services (Listing 1). This allows SCOOP to decide whether an anomalous code element realizes multiple relevant concerns. It is important to note that, when the architectural information is not available, both kinds of traces can be automatically recovered from the source code by third-party tools [4].

Moreover, SCOOP embeds a domain-specific language (DSL), allowing developers to define their own detection strategies and threshold values. Therefore, developers are able to implement different detection strategies for distinct projects and to calibrate their sensitivity accordingly. Moreover, the language is enriched with support for selecting architecture-sensitive metrics, such as Number of Architectural Concerns implemented by a given Class (NACC). An example of detection strategy defined using the DSL provided by SCOOP is presented in Listing 2.

Listing 2. Detection strategy enriched with architecture-code traces.

```
codeanomaly<class> ShotgunSurgery: ((CM > 5) and (CC > 3))
                                    or ((NACC > 1) and (NCCC > 2))
CM: Changing Methods
CC: Changing Classes
NACC: Number of Architectural Concerns per Class
NCCC: Number of Client Components per Class
```

As it can be noticed, SCOOP in this case is not only looking for coupling between classes (in terms of client classes and affected methods), like in conventional detection strategies [10]. It also looks for classes that introduce accidental couplings between architectural components, represented by the clause highlighted in gray. To this end, SCOOP verifies if the class realizes different architectural concerns (using the NACC clause) and has clients in different components (using the NCCC clause). Thus, SCOOP can detect code anomaly occurrences that would be otherwise either ignored by state-of-art tools or intertwined with long lists of irrelevant code anomalies (Section II).

SCOOP also provides a set of pre-defined detection strategies, as explained in Section V.

B. Detection of Code Anomaly Patterns

SCOOP also exploits relationships between code anomalies in order to guide developers towards those that are more likely to impact the system's architecture. Groups of structurally-related code anomalies can influence the system's architecture in different ways, ranging from anomalous elements that belong to the same architectural element to those scattered over the architecture design. These groups of code anomalies are referred as *code anomaly patterns*. In the following we present 3 out of 11 patterns currently detected by SCOOP, due to space constraints. An explanation of all the patterns, examples of their occurrences in a project as well as specific algorithms used to detect their occurrences can be found online [13].

Hereditary Anomaly refers to code anomalies that infect the parent element (e.g. parent class) of an inheritance tree and, additionally, are propagated to descendants in this tree. The impact of this pattern on the architecture design is given by the proliferation of the anomaly' negative effect. For example, when the infected parent class introduces couplings between different components via its methods' parameter types, all its descendants will strengthen such relationships. This situation can lead to the introduction of an undesirable coupling degree between architectural components and, hence, adverse ripple effects scattered over different components. For detecting this pattern, SCOOP verifies, for each infected parent class, whether their children or the overwritten methods are infected with the same anomaly.

Replicated External Network groups code anomalies within a component that depend on the same external code elements (i.e., elements located in other component). The occurrences of this pattern may suggest the lack of a shared component's interface. Therefore, any internal code element in the component can establish communication with external ones. This problem can be the source of tight coupling degree between components and unexpected relationships between code elements. The detection of this pattern involves analyzing all the anomalous elements in a given component and verifying whether they depend on a common external set of elements. Note that the detection of this pattern necessarily involves analyzing architecture-code traces. Consequently, the occurrences of this pattern could not be accurately detected by current state-of-art tools.

Multiple-Anomaly Syndrome occurs when a given code element is simultaneously infected by several anomalies. This pattern may warn developers about the existence of several architectural problems depending on the type of the code anomalies. For instance, co-occurrences of Large Class [1] and Feature Envy [1] may suggest that the responsibilities are inadequately distributed over the architecture elements. This occurs when the complexity of the infected class increases because some of its methods deal with functionalities that are modularized by other architectural components. To detect this pattern, SCOOP verifies, for each anomalous code element, whether it is infected by several anomalies or not. Note that the architectural relevance of this pattern strongly depends on to what extent the architecture-code traces are exploited by the strategies that detect single anomalies (Listing 2).

IV. Design and Implementation Details

Figure 1 depicts a representation of SCOOP's design and how its main elements are related to each other.

SCOOP's Inputs. SCOOP gets as input four different sources of information (Figure 1): the source code that will be analyzed; a well-formatted file – either an XML or a CSV– containing conventional code metrics calculated for each code element; a file – defined using our DSL – representing the detection strategies to be used to identify the code anomalies and, optionally, architecture-code traces. We decided to get externally-collected code metrics as input, as many tools already calculate them [15][17]. Our DSL was implemented using XText [19], a framework that provides edition features in order to reduce mistakes in the specification of detection strategies. Finally, the last input is a well-formatted file that describes the architecture-code traces. Details regarding how a developer can provide all the inputs required by the tool are provided in the next section.

SCOOP's Engine. The aforementioned inputs are then processed by SCOOP in order to detect architecturally-relevant code anomalies. SCOOP uses BAT (Bytecode Analysis Toolkit) [3] to generate a representation of structural properties of the code elements as logical facts in Prolog [14]. The architecture-code traces are also stored in a Prolog-based representation. This particular representation allows SCOOP to perform queries involving different sources of information. Also, the use of Prolog can foster the integration with other programming languages, as we can develop translators from these languages to Prolog. This characteristic is particularly interesting as recent studies have shown that most software projects are currently implemented in four different programming languages [3].

After generating the prolog representation, SCOOP is able to collect metrics supported by our DSL (e.g. NCCC) that are not received as input. To this end, SCOOP uses the JPL Java library [8] that provides APIs to perform Prolog queries. Listing 3 illustrates a query used to determine dependencies, in terms of field declarations, between classes

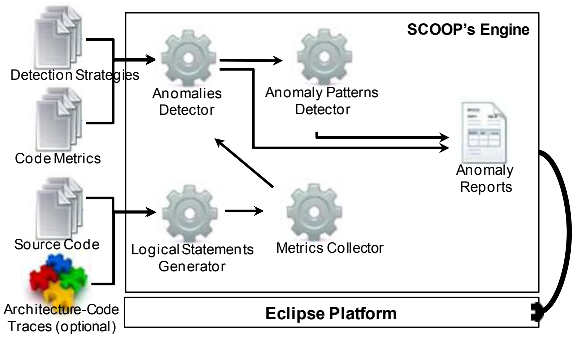

Figure 1. SCOOP Design.

Listing 3. Prolog query used by SCOOP to detect dependencies.

```
field_dcl(pkg,src,dst,fld) :- field(class(pkg,src), fld,dst),
                              mapping(comp_src, class(pkg,src),
                              mapping(comp_dst,class(pdst, dst)),
                             comp_src \== comp_dst.
```

that belong to different components. Such query returns those classes *src* that declare fields *fld* whose type *dst* belong to an external component. Other queries are also performed by SCOOP to compute dependencies between components - like inheritance and method call relationships between classes. Once all the required metrics used in the detection strategies are collected, SCOOP identifies single code anomalies. Finally, it searches for the selected set of pre-defined anomaly patterns (Section III.B). This whole process results in two different lists: single code anomalies and patterns, where the latter are more likely to represent architecturally-relevant code anomalies.

V. USAGE SCENARIO

To run SCOOP, the user needs to provide the tool's inputs (Section IV). The input files should be located in the root directory of the project under analysis. The only mandatory input is the measures file. This file should be named *metrics.csv* and can be generated by well known code analyzers, such as Together [15] or Understand [17], used in SCOOP's current implementation. The architecture-code traces can be generated by modeling tools [2][18], or described by the user in a well-formatted file named *mappings.pl* (Listing 1). The definition of architecture-code traces is not mandatory; however, in this case, SCOOP will only rely on static analysis of the source code to identify code anomalies and their patterns.

SCOOP Configuration. For configuring the detection of code anomalies, SCOOP provides a file named *strategies.rule*, located in the plug-in folder. This file contains a set of pre-defined detection strategies that will be used to identify code anomalies. The user can navigate to that file to include a new set of detection strategies or adjust the pre-defined ones in terms of metrics and thresholds.

For configuring the detection of code anomaly patterns, the user navigates to Eclipse properties view by right-clicking on the project under analysis. On the *SCOOP* configuration tab, the user can select which code anomaly patterns will be detected according to their needs. The user can also modify the default threshold values associated with each code anomaly pattern. It is important to highlight that, although possible, the manual configuration of SCOOP and its detection strategies are not mandatory; users can rely on the set of pre-defined strategies and thresholds.

Results Analysis. For running SCOOP, the users must right-click on the project under analysis and select the *Execute SCOOP* option. After running SCOOP, the user will get as result two lists of code elements, corresponding to single code anomalies and code anomaly patterns, shown in two different views. The *Single Code Anomalies* view shows the list of all single code anomaly instances grouped by their types. The type of each code anomaly is defined in the strategy that detects its instances. The *Code Anomaly Patterns* view depicts the list of all the detected pattern instances, which are also grouped by the pattern's type. By double-clicking on each detected instance (i.e. single code anomaly or pattern instances), the user navigates to the source code of the corresponding anomalous code element. Images of the configuration screens as well as both result views can be found online [13].

VI. CONCLUSIONS AND ONCOMING WORK

In this paper we presented SCOOP, a tool for detecting architecturally-relevant code anomalies. Our initial results indicated that SCOOP was able to detect a significant number of architectural problems by analyzing code anomaly patterns, providing concrete evidences of the usefulness of the tool [13]. Currently, we are working on a module for ranking relevant anomalies based on different code properties, such as change and fault-proneness, helping developers to prioritize refactorings more effectively. Future work also includes automatic extraction of static code metrics and recovery of architecture design, currently provided as inputs to SCOOP. Also, it is our intention to evaluate the performance of SCOOP (i.e. running time and memory consumption) when detecting the architecturally-relevant code anomalies.

REFERENCES

[1] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[2] ConcernMapper. Website. http://www.cs.mcgill.ca/~martin/cm.

[3] M. Eichberg et al. Defining and Continuous Checking of Structural Program Dependencies. In Proc. of the 30th ICSE, 2008.

[4] J. Garcia et al. Enhancing architectural recovery using concerns. In Proc. of the 26th ASE, 2011.

[5] L. Hochstein and M. Lindvall. Combating architectural degeneration: A survey. Info. & Soft. Technology, 2005.

[6] I. Macia et al. Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? In Proc. of the 10th AOSD, 2012.

[7] I. Macia et al. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In Proc. of 16th CSMR, 2012.

[8] JPL: http://www.swi-prolog.org/packages/jpl/java_api/index.html

[9] L. Mara et al. Hist-Inspect: A Tool for History-Sensitive Detection of Code Smells. In Proc. of the 10th AOSD, 2011

[10] R. Marinescu. Detection Strategies: Metrics-based rules for detecting design flaws. In Proc. of the 20th ICSM, 2004.

[11] R. Marinescu et al. inCode: Continuous Quality Assessment and Improvement. In Proc of the 14th CSMR, 2010.

[12] N. Moha et al. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE TSE, 2010.

[13] SCOOP: http://www.inf.puc-rio.br/~ibertran/SCOOP.

[14] Swi-Prolog: http://www.swi-prolog.org/

[15] Together: http://www.borland.com/br/products/together.

[16] N. Tsantalis and A. Chatzigeorgiou. Ranking Refactoring Suggestions based on Historical Volatility. In Proc of the 15th CSMR, 2011.

[17] Understand: http://www.scitools.com/

[18] Vespucci: http://www.opal-project.de/vespucci_project/index.html

[19] XText: http://www.eclipse.org/Xtext/.