Automating the Architecture Evaluation of Enterprise Information Systems

Felipe Pinto[1,2], Uirá Kulesza[1] and Eduardo Guerra[3]
[1]*Federal University of Rio Grande do Norte (UFRN), Natal, Brazil*
[2]*Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN), Natal, Brazil*
[3]*National Institute for Space Research (INPE), São José dos Campos, Brazil*

Keywords: Scenario-based Evaluation, Architecture Evaluation, Annotation, Quality Attribute, Scenario.

Abstract: Traditional scenario-based architectural analysis methods rely on manual review-based evaluation that requires advanced skills from architects and evaluators. They are applied when the architecture has been specified, but before its implementation has begun. The system implementation is one additional and fundamental element that should be considered during the software architecture evaluation. In this paper, we propose an approach to add information, which ideally should come from traditional evaluation methods, about scenarios and quality attributes to the source code using metadata in order to allow the automatic analysis producing a report with information about scenarios, quality attributes, source code assets and potential tradeoff points among quality attributes. The paper also presents the preliminary results of the approach application to an enterprise web information system and an e-commerce web system.

1 INTRODUCTION

Over the last decade several software architecture evaluation methods based on scenarios and quality attributes have been proposed (Clements, 2002) (Bengtsson, 2004). These methods use scenarios in order to exercise the software architecture what allow the gain of architectural-level understating and of predictive insight to achieve desired quality attributes (Kazman, 1996).

Traditional scenario-based methods produce a report as output which contains information about risk analysis regarding architecture decisions. ATAM (Clements, 2002), produce information about *tradeoff points*. A *tradeoff point* is an architectural decision that affects more than one quality attribute. For example, changing the level of encryption could have impact on both security and performance.

All these methods are applied manually and rely on manual review-based evaluation that requires advanced skills from architects and evaluators. They are classically applied when the architecture has already been specified, but before implementation has begun. The system implementation is one additional element that can be useful when suitably analyzed, for example, if the software evolves causing critical architectural erosion (Silva, 2012) implying on the need of executing the process of evaluation again because the architecture designed has several differences to the architecture implemented (Abi-Antoun, 2009).

We believe that the usage of system implementation during the architecture evaluation can enable the automation of this process and the reuse of architectural information and tests. In this context, we propose an approach that introduces additional information, which ideally should come from traditional architecture evaluation methods, about scenarios and quality attributes to the application code using metadata. Further, it executes an automated tool to perform the analysis producing a report with relevant information about scenarios, quality attributes and code asset, such as: (i) the scenarios affected by particular quality attributes; and (ii) the scenarios that potentially contain *tradeoff points* and should have more attention from the architecture team

The rest of this paper is organized as follows: Section 2 introduces the approach; Section 3 presents the tool developed; Section 4 shows two case studies where we have applied our approach; Section 5 discusses some related works and, finally, Section 6 concludes the paper.

2 APPROACH OVERVIEW

This section presents an overview of our approach. The main goal is to automate the architecture evaluation by adding extra information with metadata to the application source code. The approach presented here is independent of programing language or platform. Figure 1 summarizes the approach steps.

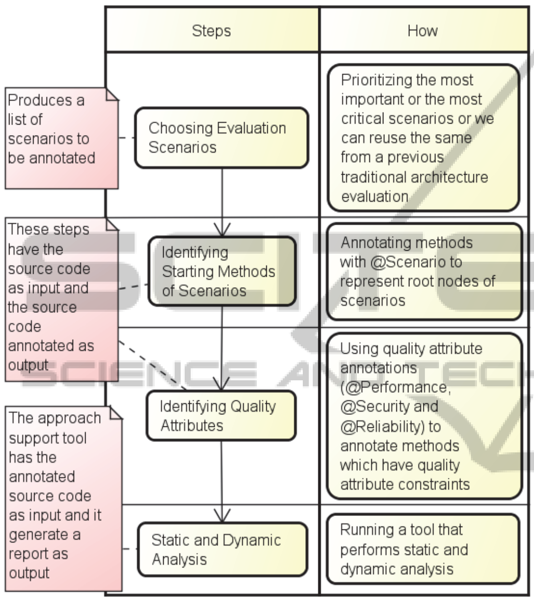
Steps
How
Produces a list of scenarios to be annotated
Choosing Evaluation Scenarios
Prioritizing the most important or the most critical scenarios or we can reuse the same from a previous traditional architecture evaluation
These steps have the source code as input and the source code annotated as output
Identifying Starting Methods of Scenarios
Annotating methods with @Scenario to represent root nodes of scenarios
Identifying Quality Attributes
Using quality attribute annotations (@Performance, @Security and @Reliability) to annotate methods which have quality attribute constraints
The approach support tool has the annotated source code as input and it generate a report as output
Static and Dynamic Analysis
Running a tool that performs static and dynamic analysis

Figure 1: Approach overview.

In Figure 1, the column “Steps” presents the step description and the column “How” shows an example of how it is accomplished on our developed tool. Next subsections detail each one of the steps which are presented considering the developed tool that uses annotation to define metadata information. In languages that do not support annotations, the metadata can be defined externally, such as on databases or XML files.

2.1 Choosing Evaluation Scenarios

The first step of our approach is to choose the scenarios from the target architecture to be evaluated. In order to perform this step, we can reuse information produced by previous activities from the development process. In particular, the elicited relevant scenarios gathered during the application of traditional architecture evaluation methods, such as ATAM or others (Muhammad, 2004), can be reused during this step.

2.2 Identifying Scenarios

In this step we identify the starting points of the execution of the chosen scenarios in the application source code under evaluation. A scenario execution defines paths of execution which can be abstracted to a call graph where each node represents a method and each edge represents possible invocations.

Our challenge in this step is to define how identifying scenarios or paths of execution in the application source code. A simple solution is just identify the method which represents the call graph root node and after that based on the invocations of this node to identify the complete call graph related to this root node.

In order to allow the introduction of this information in the source code, our tool defines an annotation named `@Scenario` which defines an attribute to identify it uniquely. Figure 2 shows an example of this annotation.

2.3 Identifying Quality Attributes

The identification of quality attributes in the application source code is similar to the identification of starting methods. We have to add the metadata to the element that we are interested. The tool currently defines method annotations considering the following quality attributes: `@Performance`, `@Security` and `@Reliability`. These annotations are the ones implemented by the developed tool, the approach can be generalized to evaluate other quality attributes.

Figure 2 shows the annotations and their respective attributes. Performance annotation has two attributes: name and time limit. Name is a string that uniquely identifies it and time limit is a long integer that specifies a maximum time expected in milliseconds. The annotated method must complete its execution in a shorter time compared to the time limit value. As a consequence, we can monitor if an annotated method has improved or decreased its performance in the context of an evolution among different releases of the system.

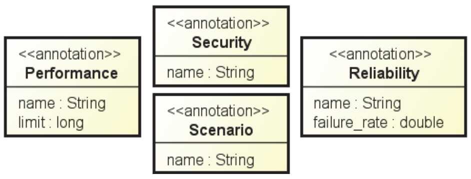

Figure 2: Approach annotations.

Security annotation has currently one attribute, a string that uniquely identifies it. It is useful because allows determining which execution paths of scenarios have potential to contain *tradeoff points*. For example, increasing the level of encryption could improve the security of the system, but on the hand it requires more processing time. That is, if a path of scenario execution is associated to more than one quality attribute, we need to observe and monitor it carefully because it has potential to contain *tradeoff points*.

Similarly to performance, the reliability annotation has a unique string attribute and a double attribute that specifies the failure rate. It represents the maximum failure rate expected for an annotated method from zero to one. Zero means that it never fails and one that fails in all the cases. Currently, it is used to check if a particular scenario has potential *tradeoff points*.

2.4 Static and Dynamic Analysis

The last step of our approach involves the execution of static and dynamic analysis implemented in a tool. This tool parses the metadata from the source code and performs analysis automatically in order to enable the automate architecture evaluation based on the configured scenarios and quality attributes.

During the static analysis the tool parses the annotations and builds a call graph of the methods involved in the execution paths of the system scenarios. After that, using this information, the tool can: (i) discover the quality attributes associated to a particular scenario or which one has potential to have *tradeoff points*; (ii) discover which methods, classes or scenarios could be affected due to a particular quality attribute; (iii) perform traceability of scenarios and quality attributes in the source code.

The dynamic analysis also benefits from our code annotations in order to perform the architecture evaluation during the system execution. It allows monitoring the quality attributes and, also, dynamic reflective calls are capture only by dynamic analysis. The following analysis can be currently accomplished using our approach: (i) calculating the performance time or failure rate from a particular annotated method or from a complete path of scenario execution; (ii) verifying if the constraints defined by quality attribute annotations are respected over the different evolution releases of the system; (iii) logging of several information captured during the runtime; (iv) adding more useful information to detect and analyze *tradeoff points*.

3 APPROACH TOOL SUPPORT

This section introduces a tool that we have developed to support our approach. It has been accomplished as two independent components: (i) the static analysis is implemented as an Eclipse plugin; and (ii) the dynamic analysis is made available as an executable JAR file.

3.1 Tool Support for Static Analysis

The static analysis tool allows executing the architecture evaluation over Eclipse projects. It currently parses source code from Java projects. Figure 3 shows a partial class diagram.

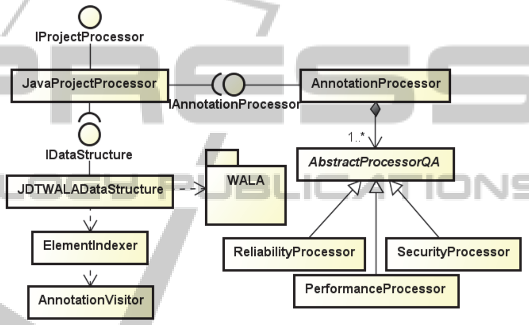

Figure 3: UML class diagram showing tool processors.

The `JavaProjectProcessor` class calls other classes in order to build the call graph of the system under architectural evaluation. We have used the CAST (Common Abstract Syntax Tree) front-end of WALA (Watson Libraries for Analysis) static analysis framework (WALA, 2012) to build the call graph of the scenarios of interest. `AnnotationProcessor` class aggregates a set of different concrete strategy classes to process the different quality attribute annotations. Each one of them is responsible for the processing of a particular kind of annotation. During the annotation parsing, the `AnnotationProcessor` class also builds the list of scenarios annotated to complement the data structures built previously.

`JavaProjectProcessor` class also uses the `JDTWALADataStructure` to access and manipulate the application call graph and the indexes. The `JDTWALADataStructure` class uses `ElementIndexer` to build indexes of methods, classes and annotations to be used during the analysis. Actually, the annotation index is created by the `AnnotationVisitor` class that reads the source code looking for annotations.

Figure 4 summarizes the static analysis process. `JavaProjectProcessor` uses `JDTWALADataStructure` to build the call graph and the indexes. `ElementIndexer` is used to build the method index and the annotation index, but it creates an object `AnnoationVisitor` that parsers the source code looking for annotations. Then, `AnnotationProcessor` processes the scenario annotations and builds a list of scenarios. Finally, it processes each quality attribute annotation calling every `AbstractProcessorQA` specializations.

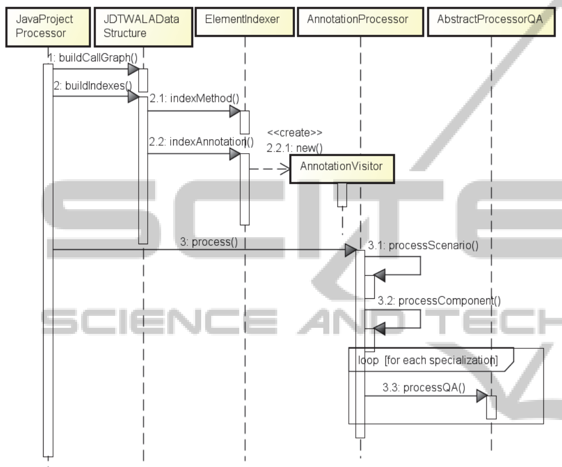

Figure 4: UML sequence diagram to static analysis.

Our static analysis tool uses a model to represent the relationships among the system assets, such as classes, methods, scenarios and quality attributes. Figure 5 shows a partial class diagram of this model. The `ScenarioData` has a starting root method and `MethodData` has a declaring class. Each quality attribute is a specialization of the `AbstractQAData` which in turn keeps a reference to its related method. Finally, every `MethodData` instance has also an attribute signature that references the method node in the WALA call graph.

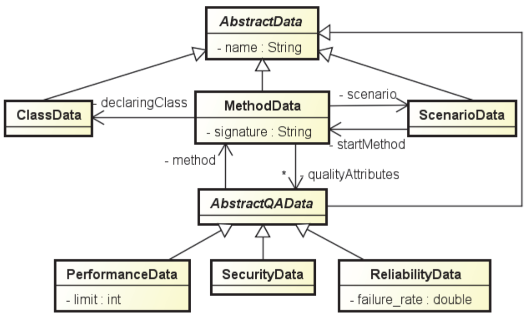

Figure 5: Class diagram of the static analysis model.

3.2 Tool Support for Dynamic Analysis

Our dynamic analysis tool has been implemented using AspectJ language by defining aspects that monitor the execution of annotated methods. Essentially, the tool builds a dynamic call graph during application execution intercepting the approach annotations. In this way, if an annotated method is called, a specific aspect for each kind of annotations is automatically invoked. When a method is intercepted, the aspects register and monitor the method execution by gathering information about their name, the current execution thread and the parameters values. These information is then stored in the dynamic call graph in order to help the decision making about what to do when something is wrong, for example, logging the `@Reliability` annotated methods who has thrown or handled an exception.

The current version of our tool has implemented aspects to intercept scenarios and quality attributes annotations (performance, security and reliability). These aspects use concrete strategy objects, which have a common interface in order to make possible the aspects to call them. In that way the developers can define their own strategies for dealing with the quality attribute which are generally dependent on the domain and application.

In our tool, we have implemented default strategies to gather and store information about the execution of the relevant architecture scenarios. In addition, we have also implemented specific strategies for our case studies, which will be presented in Section 4.

4 APPROACH EVALUATION

We have applied our approach in two different systems. In the first one, we have explored the static analysis in an academic enterprise large-scale web system developed for our institution, and in the second one the dynamic analysis in an e-commerce web system. Our main goal was to conduct an initial evaluation of the approach in order to verify its feasibility and how the developed tool behaves in practice.

4.1 Static Analysis in Action

We have applied the static analysis tool of our approach to enterprise web systems from SINFO/UFRN. SINFO is the Informatics Superintendence at Federal University of Rio

Grande do Norte (UFRN) in Brazil. It has developed several enterprise large-scale information systems (SINFO, 2012) which perform full automation of university management activities. Due to the quality of these systems, several Brazilian federal institutions have licensed and extended them to their needs.

Our main goal was to verify the approach feasibility of static analysis in practice. In this sense, the tool should extract useful information in order to help developers answering some questions, such as: (i) what scenarios does a specific method belong to? (ii) what kinds of quality attributes can affect a specific scenario? (iii) what are the scenarios that contain potential *tradeoff points* among quality attributes?

4.1.1 Choosing Evaluation Scenarios

In the first step we have chosen some specific scenarios: (i) *sending message* – scenario used for sending messages (emails); (ii) *authenticated document generation* – scenario used to generate authenticated documents; (iii) *user authentication* – scenario used to authenticate users in the web application; (iv) *mobile user authentication* – scenario used to authenticate users from a mobile device.

4.1.2 Identifying Scenarios

In this step the starting execution method for each chosen scenario were identified. They are, respectively: (i) `sendMessage()`; (ii) `execute()`; (iii) `userAuthentication()`; (iv) `mobileUserAuthentication()`.

4.1.3 Identifying Quality Attributes

The methods and quality attributes selected were: (i) `getJdbcTemplate()` with `@Performance` – it was considered to be relevant for performance requirements because it is accessed by several database operations; (ii) `enqueue()` with `@Security` – it is used by the system to enqueue messages that will be sent over the network; (iii) `createRegistry()` with `@Security` – it is used to create the registry of an authenticated document to ensure its legitimacy; (iv) `toMD5()` with `@Security` – it is used to create an MD5 hashing of strings, for example, passwords; (v) `initDataSourceJndi()` with `@Reliability` – it is used to initialize the access to the database and was considered critical for reliability because if the database initialization fails, the system is not going to work adequately.

4.1.4 Executing the Static Analysis Tool: Preliminary Results

The tool execution has extracted useful and interesting information in order to help us answering the questions highlighted on section 4.1.

Considering the first question – (i) *what scenarios does a specific method belong to?* – the tool can determine that the `getJdbcTemplate()` method, for example, belongs to the following scenarios: user authentication, mobile user authentication and authenticated document generation. This is possible because the tool builds a static call graph of each scenario and calculates if a call to a particular method exists in some of the possible paths of execution.

Regarding the second question – (ii) *what kinds of quality attributes can affect a specific scenario?* – the tool verifies all the paths for a specific scenario checking which ones have any quality attribute. Our tool has identified, for example, all the quality attributes related to the *User Authentication* scenario: (i) performance quality attribute – because the method `getJdbcTemplate()` belongs to a possible path; (ii) the reliability quality attribute because the method `initDataSourceJndi()` also belongs to a possible path; and (iii) finally, the security quality attribute for the same reason, the method `toMD5()` is used to encrypt the user password.

Finally, for answering the third question – (iii) *what are the scenarios that contain potential tradeoff points among quality attributes?* – the tool looks for scenarios affected by more than one quality attribute because they contain potentially *tradeoff points* among their quality attributes. The tool has identified that: (i) user authentication and mobile user authentication are potential scenarios to have *tradeoff* because they are affected by performance, security and reliability; (ii) authenticated document generation is another potential *tradeoff point* because it addresses the reliability and security quality attributes; on the other hand (iii) the sending message does not represent a *tradeoff point* because it is only affected by the security quality attribute. These results are summarized in Table 1.

The information identified automatically by our tool is useful to indicate to the architects and developers which specific scenarios and code assets they need to give more attention when evaluating or evolving the software architecture through the conduction of code inspections or the execution of manual or automated testing. In that way, our preliminary evaluation in large-scale enterprise systems has allowed us to answering the expected questions previously highlighted and demonstrated the feasibility of our static analysis approach.

Table 1: Some information about tradeoffs in scenarios.

Scenario:	**User Authentication**
Performance:	getJdbcTemplate()
Security:	toMD5()
Reliability:	initDataSourceJndi()
Tradeoff:	Potential
Scenario:	**Mobile User Authentication**
Performance:	getJdbcTemplate()
Security:	toMD5()
Reliability:	initDataSourceJndi()
Tradeoff:	Potential
Scenario:	**Authenticated Document Generation**
Performance:	-
Security:	createRegistry()
Reliability:	initDataSourceJndi()
Tradeoff:	Potential
Scenario:	**Sending Message**
Performance:	-
Security:	enqueue()
Reliability:	-
Tradeoff:	No

4.2 Dynamic Analysis in Action

The evaluation of the dynamic analysis was performed by applying our tool to the EasyCommerce web system (Torres, 2011); (Aquino, 2011) which is an e-commerce web system that has been developed by graduate students at our institution. It implements a concrete product of an e-commerce software product line described in (Lau, 2006).

The main aim of our evaluation was to extract execution context information in order to analyze the aspects behaviour in practice to achieve the following dynamic analysis: (i) monitoring of scenario execution and the annotated methods; (ii) calculation of the performance time (`timeSpent`) of methods and scenarios; and (iii) detection of executed paths with potential *tradeoff points*.

4.2.1 Choosing Evaluation Scenarios

We have chosen some of the scenarios that represent the main features of EasyCommerce: (i) *registration of login information* – it records the user information about login, such as user name and password; (ii) *registration of personal information* – it records personal information about the user, such as name, address, birthday, document identification; (iii) *registration of credit card information* – it records information about users credit card such as card number and expiration date; (iv) *search for products* – It allows searching for products by its name, type or features; (v) *include product item to cart* – it allows users adding a product item to their shopping cart.

4.2.2 Identifying Scenarios

In this step the starting execution method for each chosen scenario were identified. They are, respectively: (i) `registerLogin()`; (ii) `registerUser()`; (iii) `registerCreditCard()`; (iv) `searchProducts()`; (v) `includeItemToCart()`.

4.2.3 Identifying Quality Attributes

We have chosen some methods belonging to the scenarios that appear to have potential to be relevant to specific quality attributes. The selected ones were: (ii) `save()` with `@Performance` – it is used by the system to save all its objects, because of that it should run as fast as possible; (ii) `save()` with `@Reliability` – considering that this method is executed many times and it represents a critical action is fundamental to analyze its robustness. (iii) `registerLogin()`, `registerUser()`, and `registerCreditCard()` with `@Security` – these methods manipulate user confidential information and they are in some way related to security.

4.2.4 Executing the Aspects of Dynamic Analysis: Preliminary Results

We have executed the selected scenarios of EasyCommerce web system together with the aspects of dynamic analysis in order to perform the evaluation of the results and benefits which are discussed next.

Our approach defines a specific strategy to analyze the annotated scenarios through an aspect. For such cases, our aspect builds a dynamic call graph structure used: (i) to monitor de scenarios execution; (ii) to calculate the time to execute completely the scenario or a particular method; (iii) to get some information about the system execution context, such as the date and time of execution.

The current stored information provided by our scenario aspect can help architects and developers to identify: (i) all the cases where a method has taken more time to execute than the specified value in the `@Performance` annotation; (ii) the execution time for a given scenario; and (iii) the quality attributes addressed in particular methods or scenarios.

Table 2 shows information collected by scenario aspect which shows some obtained results from the execution of the scenarios register of login, register of personal information and register of credit card information. Executing these scenarios we have one occurrence of performance in `save()`, three occurrences of security in `registerLogin()`, `registerUser()` and `registerCreditCard()` and one occurrence of reliability in `save()`.

Table 2: Sample of data collected by dynamic analysis.

Registration of login information	
Execution time (ms):	3
Performance:	-
Security:	registerLogin()
Reliability:	-
Registration of personal information	
Execution time (ms):	2
Performance:	-
Security:	registerUser()
Reliability:	-
Registration of credit card information	
Execution time (ms):	150
Performance:	save()
Security:	registerCreditCard()
Reliability:	save()

Analyzing the dynamic call graph generated the tool can inform which scenarios contain potential *tradeoff points*. For example, `registerCreditCard()` calls `record()` that calls `save()`. The `save()` method has been annotated with the performance and reliability quality attributes. The `registerCreditCard()` method has been annotated with the security quality attribute. Thus, we have a scenario with three quality attributes involved and because of that a potential *tradeoff points* among them.

The dynamic analysis process in this study has met our expectations because it has allowed us extracting useful information of the execution context, such as, monitoring of scenarios and quality attributes, calculating the performance of scenarios and specific methods and last, but not least, detecting executed paths with potential *tradeoff points*.

5 RELATED WORK

To the best of our knowledge, there is no existing proposal that looks for the automation of architecture evaluation methods in the same way of ours and we have not found approaches really close to ours. In this section, we summarize some research work that address architectural evaluation or propose analysis strategies similar to ours.

Over the last years, several architecture evaluation methods, such as ATAM, SAAM, ARID (Clements, 2002) and ALMA (Bengtsson, 2004) have been proposed. They rely on manual reviews before of the architecture implementation. Our approach complements these existing methods by providing automated support to static and dynamic analysis over the source code of the software system. It contributes to the continuous evaluation of the software architecture during the system implementation and evolution.

Also, some recent research work have proposed adding extra architectural information to the source code with the purpose of applying automated analysis or document the software architecture. (Christensen, 2011) uses annotations to add information about components and design patterns with the purpose of document the architecture. (Mirakhorli, 2012) presents an approach for tracing architecturally significant concerns, specifically related to architectural tactics which are solutions for a wide range of quality concerns. These recent research work, however, do not explored the combined usage of adding information related to scenarios or quality attributes.

6 CONCLUSIONS

We presented an approach to automating the software architecture evaluation using the source code as input of this process that consists on adding metadata to the source code providing extra information, such as, scenarios and quality attributes. It provides support to the execution of static and dynamic analysis that aims the automatic evaluating of the software architecture. Finally, it has been applied in two systems: a large-scale enterprise information system and an e-commerce web system. The preliminary obtained results of the approach usage have allowed us to provide and quantifying several and useful information about architecture evaluation based on scenarios and quality attributes.

The approach presented is still under development and we are currently evolving it in order to apply to other large-scale enterprise information systems. We have also identified several possibilities for future work, for example, it is possible to detect which paths of execution are more often followed and their performance to suggest to the developers or architects team try to improve them. Another possibility is to verify if all the possible paths of execution for all scenarios prioritized on the architecture evaluation have been effectively executed and tested. It is also possible to check missing paths (Liu, 2011) when a path exists in the static call graph and it does not exist in the dynamic call graph meaning a not tested path or dead code.

REFERENCES

Abi-Antoun, M., Aldrich, J. 2009. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. *SIGPLAN Not. 44*, 10 (October 2009), 321-340.

Aquino, H. M. (2011). *A Systematic Approach to Software Product Lines Test*. 2011. MSc Dissertation, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil, 2011.

Bengtsson, P., Lassing, N., Bosch, J., Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*. 69, 1-2 (January 2004).

Christensen, H. B., Hansen, K, M. 2011. Towards architectural information in implementation (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 928-931.

Clements, P., Kazman, R., Klein, M. 2002. *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley.

Kazman, R., Abowd, G., Bass, L., Clements, P. 1996. Scenario-Based Analysis of Software Architecture. *IEEE Softw. 13*, 6 (November 1996), 47-55.

Lau, S. Q. 2006. *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates*, MASc Thesys, University of Waterloo.

Liu, S., and Zhang, J. 2011. Program analysis: from qualitative analysis to quantitative analysis (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*.

Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.. 2012. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*.

Muhammad and Ian Gorton. 2004. Comparison of Scenario-Based Software Architecture Evaluation Methods. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*.

Silva, L., Balasubramaniam, D., 2012. Controlling software architecture erosion: A survey. *J. Syst. Softw. 85*, 1 (January 2012), 132-151.

SINFO. (2012). *Informatics Superintendence*, UFRN: http://www.info.ufrn.br/wikisistemas, May 2012.

Torres, M. 2011. *Systematic Assessment of Product Derivation Approaches*. MSc Dissertation, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil, 2011.

WALA, T. J. Watson Libraries for Analysis: http://wala.sourceforge.net, December 2012.