A Regression Testing Approach for Software Product Lines Architectures

Paulo Anselmo da Mota Silveira Neto1,3, Ivan do Carmo Machado2,3, Yguaratã Cerqueira Cavalcant1,3, Eduardo Santana de Almeida2,3, Vinicius Cardoso Garcia3, Silvio Romero de Lemos Meira1,3

1Federal University of Pernambuco (UFPE) – Recife, Pernambuco, Brazil
2Federal University of Bahia (UFBA) – Salvador, Bahia, Brazil
3Reuse in Software Engineering (RiSE)

Abstract—In the Software Product Lines (SPL) context, where products are derived from a common platform, the reference architecture can be considered the main asset. In order to maintain its correctness and reliability after modifications, a regression testing approach based on architecture specification and code was developed. It aims to reduce the testing effort, by reusing test cases, execution results, as well as, selecting and prioritizing an effective set of test cases. Taking advantage of SPL architectures similarities, this approach can be applied among product architectures and between the reference and product architecture. This study also presents an evaluation performed in order to calibrate and improve the proposed approach.

Keywords—Software Product Lines; Regression Testing; Reference Architecture.

I. INTRODUCTION

In order to achieve the ability to build individualized products, to meet individual customer needs, companies need high investments, which consequently leads to higher prices for final products. As an alternative, many companies started to introduce the common platform development, in order to assemble a greater variety of products, by reusing common parts, still meeting customer requirements and expectations, at a lower cost. Common platforms are the core of the SPL engineering. In SPL, projects, it is called the reference architecture, which provides a common high-level structure for all product line applications [1].

Software architectures, the central role of SPL projects, are becoming the central part during the development of quality systems [2], being the first model and base to guide the implementation [3] and providing a promising way to deal with large systems [4]. Nevertheless, it evolves over time to meet customer needs, environment changes, improvements or corrective modifications. Thus, to be confident that these modifications are in conformance with the architecture specification, and also that they did not introduce unexpected errors, and the new features work as expected, regression test is performed [5].

Considering testability in architecture design, the testing activity can be made more efficiently and effectively [6], since when a modification occur few paths will be affected. Thus, few tests need to be rerun, few obsolete test cases will be removed and few new test cases need to be designed. Moreover, when the changes are so common [7], maintainability is one important criteria when developing software [8]; if the regression testability is not considered in early phases, more expensive and harder are the modifications and testing and retesting activities.

The adoption of a regression test selection technique is useful in some scenarios and domains. Basically, it selects a set of test cases from existing test suites to test the original version, avoiding the execution of all test cases. For example, in avionics, the reduction of one test case may save thousands in testing resources [9]. However, the test selection technique is only justifiable when the cost to select test cases is lesser than running the entire test suite.

In SPL context, each product can be seen as variants of each other, which makes regression testing inviting. In this sense, an architecture regression testing approach was defined by taking advantage of regression test benefits and SPL architectures similarities, through selecting and prioritizing sets of effective and efficient test cases, based on previously collected information. There are some scenarios, considering SPL (Core Asset Development (CAD) and Product Development (PD)), where the use of the proposed regression testing approach is useful, such as: (i) during reference architecture evolution and modification, (ii) when changes in the product architecture should be propagated through the overall product line, (iii) maintenance of the conformance among product architectures and the reference architecture, and (iv) to address the problems raised from a typical selective retest technique. An experimental study was also performed in order to evaluate the proposed approach.

II. REGRESSION TESTING

Although unit, integration and system test levels have their importance to detect specific defects, regression testing is a way to efficiently test the conformance after modifications. Instead of submitting the modified software to all test
levels again, regression testing is applied, reducing costs and detecting faults earlier. Next, an overview regarding to regression test is presented, as means to provide a bases to better understand the proposed approach.

A. Corrective vs Progressive Regression Testing

Characterized by its huge cost and expensive implementation, maintenance initiates after the product release and aims to correct, keep the software updated, as well as to fit with the new environment needs. According to [10], around 20% of all maintenance work is spent fixing mistakes, the remaining 80% are spent adapting the system according to the external environment needs, making enhancements requested by users and reengineering an application for future use. One way to reduce the maintenance cost can be achieved by an efficient and effective regression testing.

Based on the four categories of maintenance [11] and ISO/IEC 14764 [12]: adaptive, perfective, corrective and preventive, two types of regression testing can be identified.

**Corrective Regression** which is often performed after some corrective action. It is applied when specifications are unmodified (e.g. when the code is not in conformance with the specification). When the modification affects only some instructions and design decisions (e.g. changing only the way used to implement the variability mechanism - such as: inheritance, parameterization and design patterns - without modify the specification), it makes the test cases from the previous test plan be more reusable. However, when they involve possible changes to the control and data flow structures, some existing test cases can be no longer valid to test that portion of the software. Since program failures can occur at any time, this type of regression testing should be applied for every correction.

**Progressive Regression** is typically performed after adaptive and perfective maintenance. It is used when specifications are modified (e.g. the addition of a new feature or functionality). This specification modification is caused by new enhancements or new data requirements, which should be incorporated into the system. In order to handle the testing of this modification, new test cases need to be designed. This type of regression testing is performed during regular intervals since adaptive or perfective maintenance is typically done at a fixed interval.

**B. Test Case Classes**

Let P be a program, let P’ be a modified version of P, and let T be a test suite created to test P. The main idea behind regression testing techniques is to select a subset of tests (T’) of T to make confidence that P’ was correctly modified, still working properly as previously and no new errors were inserted [13].

In [14] and [15], the authors categorize test cases created in the previous phase (integration testing) from the previous test plan in the following classes:

**Reusable Tests**: Responsible for testing an unmodified portion of the specification and architecture structure. They are still valid but do not need to be executed again to guarantee the regression testing safety;

**Retestable Tests**: This class includes all tests that should be repeated when the software structure is modified, even though the specification regarding the software structure is not modified. They are still valid and need to be rerun;

**Obsolet Test**: Correspondent to the test cases that cannot be executed on the new version, since they become invalid for the new context. According to [14], there are three reasons for that: (i) The structural tests (based on the control and data structures) are designed to increase the structural coverage of software. Since the structure can be changed in different versions of the software, some test cases become obsolete; (ii) Due to some changes in a specific software component, some test cases may not be testing the same structure, despite they correctly specify the input/output relation; (iii) When the test cases specify an incorrect input/output relation. It happens when a specification is modified and the related tests are not according updated.

**Unclassified Tests**: Involve the test cases which may either be retestable or obsolete. According to [14], two new classes of test cases can be included in the test plan: (i) New-structural tests, include tests which aims to test the modified software structure. Often they are designed to improve the structural coverage; (ii) New-specification tests, comprehend the test cases that evaluate the new code generated from the modified portion of the specification.

![Figure 1. Corrective and Progressive Regression Test Plan (14).](image)

In order to better understand the relation between the types of regression testing and they correspondent test classes, Figure 1 shows this relation. The above portion of the Figure shows that after performing a modification, obsolete test cases are removed and new-structural tests are added to the new test plan. The below portion of the Figure shows that besides removing obsolete test cases, new-structural and new-specification test cases need to be designed to test the modified version of the software. Since in this type of regression testing the specification is modified, new-specification test cases are designed.
C. Regression at Integration Level

As mentioned by McGregor [16], regression testing is a technique rather than a testing level. Burstein et. al. [17] define it as being “the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes”. Considering this standpoint, regression testing can be performed after any testing level. In our context, it will be performed after integration testing, since the purpose of the approach is to verify the integration among modules and components which compose the SPL architecture.

After unit testing level, where the components are individually tested, integration testing comes in scene. The product map, which is a SPL artifact build during scoping phase that groups all products and its respective features, is analyzed in order to identify commonalities and variabilities among SPL members. By analyzing the feature model and the feature dependency diagram, the Test Architect can identify and comprehend how the feature interacts. This information is important during test design and test suite composition, since it explicitly shows the product features and the interaction among each other. During design, it serves as input to define modules and components that composes the reference architecture and, during testing, it guides the design of test cases which evaluate the interaction between components and modules (integration testing).

The architecture diagrams, composed by components and modules, are also important for testing, since they provide the way in which the components and modules interact. Furthermore, this information will guide the analysis step (Section II-D).

VP2 represents the functionality that aims to create the savings account. The bind of VP1, VP2 or both, should be provided by the product specific architecture specification. It is important to highlight that a test case can be designed to verify different scenarios and configurations, depending on the feature(s) that was/were bound.

D. Regression Testing in SPL Architectures

There are three different scenarios where the regression testing approach is attractive. Figure 3 shows all of them, which are following described:

Scenario 1: Given a Reference Architecture (RA), composed by the integration of components (A,B,C and D), which has its conformance verified during integration testing (Section II-C). Imagine that a component A should be modified to A’ in order to reflect a change, due to evolution or even corrective action. A new version of the reference architecture (RA) is developed. Considering these two versions, the original V1 (A,B,C and D) and the new one V2 (A’,B,C and D), they need to be applied against a regression testing approach to gather confidence that the new version is flawless and it is working properly.

Scenario 2: Given the reference architecture (RA) and the product architecture PA1. Considering these two architectures as different versions V1 (A,B,C and D) and V2 (A,B,C,D,E and H), the regression testing approach can be useful. This scenario can also be considered a testing strategy commonly used in SPL testing by exploiting the existing commonalities among SPL members. While a product is individually tested, the subsequent products are tested using a regression testing approach [18].

Scenario 3: Given two product line architectures PA4 and PA5, considering both as two different versions V1 (A,B,C,D,G,L and M) and V2 (A,B,C,D,G,L and N), the regression testing approach can be applied in this context. By observing the Figure 3, the reader can be induced to think that since the PA4 was previously tested during integration testing, PA5 can be verified reusing the common tests and only considering the integration of the last component N. It is a wrong consideration because the integration of the last component could bring faults in the previously tested structure. For this reason, the application of a regression testing approach is crucial to understand the impact of the last integration.

Figure 2. A Sequence Diagram with two variation points.

Based on previous information (feature model, product map and feature dependency) and architecture views, the integration tests can be designed. Whereas the behavioral view (sequence diagrams) provides information about the functionalities of the architecture, the structural view (component and module class diagrams) give us information about the architecture structure. Figure 2 shows a sequence diagram in a scenario where a user requests for an account creation and the system can create two types of accounts special and savings account. Two variation points (optional features) can be viewed: the first VP1 represents the functionality responsible for creating the special account; the second one VP2 represents the functionalty that aims to create the savings account. The bind of VP1, VP2 or both, should be provided by the product specific architecture specification. It is important to highlight that a test case can be designed to verify different scenarios and configurations, depending on the feature(s) that was/were bound.

Figure 3. Similarities among product architectures.
The regression testing approach defined was considered in two ways. Firstly, during CAD it aims to test the conformance of the RA after a modification in a component or module which is part of it. Secondly, during the PD with the purpose to test a product architecture in respect with the reference architecture or others product architectures considering their common features.

III. A Regression Testing Approach for SPL Architectures

The purpose of regression testing in architecture is to ensure that no new defects are introduced into previous tested architecture and it continues to work properly. To be confident that the architecture is working properly, its specification can be used as test oracle to identify when the tests pass or fail.

The main inputs are the two versions (modified and original) of the architecture code, the test cases, test scripts and test suites from integration testing level saved to be further reused, all of these artifacts are considered mandatory in this approach. Architectural specifications as behavioral and structural views, as well as the feature model, the product map, the feature dependency diagram can be useful to extract information from the architecture, serving as guides to identify portions that need to be retested. Using the structural view, the relation among classes and components are clearly specified and identified. From this view and using the use cases previously specified, sequence diagrams are built in order to better represent the relation among the components and classes, making easier the creation of integration tests to be used in the regression approach. The feature model and the feature dependency diagram are used to understand the relationship among features, i.e. in cases where a feature excludes another one and even the presence of optional features (Figure 2). This information should be considered when designing integration test cases.

Specific product architectures can be instantiated based on product maps and decision models which contain information such as mandatory, optional and variant features for each product. Based on this information, the test architects are able to instantiate an architecture by selecting specific features and components. Thus, when components or modules are modified, regression testing must be performed on the application architecture, as means of evaluating its correctness [19].

A. Approach Steps

In this section, the proposed regression testing approach is described. The overall approach can be viewed in Figure 4. Although they are presented as sequentially initiated, this process represents an incremental and iterative development step, since feedback connections enable refinements along the approach. This flow illustrates the approach workflow comprising its activities, inputs, outputs, tasks and involved roles.

1) Planning and Analysis: The planning is performed in order to understand how a correction or evolution impact the architecture. By manually analyzing the architecture specification, the modified classes and methods are identified, and the relevant tests can be designed or selected based on this information. It serves as guide to support the next steps, restricting the coverage of the modified version that should be examined.

After processing a change request or receiving an architecture evolution request, the test architect starts the analysis phase. Figure 5 shows an illustrative example of a class diagram with five classes of a bank management system. In the context where a customer sends a change request describing an issue found in the credit function, the Test Architect will analyze the class diagram in order to identify the impacted classes. Considering that a modification was done in method credit from class Account, it may cause problems (regarding to business rules) in method credit (Figure 6) from class SpecialAccount. If the class SavingAccount has a similar implementation as SpecialAccount, for example using super credit(value), this class will be also impacted. Considering this scenario, the Test Architect can see that the method credit in the classes Facade, Account and SpecialAccount need to be investigated more carefully. In this excerpt, some classes were removed from Figure 5 in order to facilitate the visualization and understanding.

Based on the category of the modification (Section II-A), two types of regression testing (Section II-A) can be per-
behaviors. Thus a graph representation for both versions of modified portions of the architecture (the new and old versions) are generated.

The graphs can be control flow graph (CFG), program dependence graph, control dependence graph or a Java Inter-class Graph (JIG) depending on the test selection technique adopted. CFGs are suitable for representing the control flow in a single procedure, but it cannot handle interprocedural control flows or features of Java programming language such as polymorphism, dynamic binding, inheritance and exception handling [9]. The more language features the graph representation, the more refined will be the analysis, increasing the code coverage and decreasing the number of undetected faults.

Apiwattanapong et. al. [22] propose the Enhanced Control-Flow Graphs (ECFG) to suitably represent object-oriented constructs and model their behavior. They also present JDiff tool that generates ECFG representation for two different versions of a program and compare these versions. This tool considers both the program structure and semantics of the programming-languages constructs [22].

In our approach, the use of this type of tool is optional and depends on the specificity of the fault. In some cases, a simple textual comparison is enough to found the critical path (or fault). When using a textual diff tool, it is important to select a person with high experience and knowledge in the architecture (domain) in order to identify the problems. When textual differentiation is enough, this step and the next one can replaced by a diff tool, resulting in faster analysis, which can save effort.

**Graph Comparison.** In order to identify critical edges and understand how the code changed, the graphs are compared. A good knowledge in control flow graphs analysis is required during this step, since the Test Architect will see more easily how the code behaves.

When performing progressive regression testing, the last two steps are replaced by specification comparison. This step aims to compare the original specification with the modified one, identifying added, deleted or changed components, classes, and features.

2) **Test Design and Selection:** After the graphs, specification or textual (code) comparison, the critical edges and paths are analyzed aiming to find some test that exercise a modified portion of the architecture. In this step, the Test Architect analyses the paths trying to classify the previous designed tests (integration tests) from the repository according to the classes established in Section II-B. It will aid in the suite composition task, since all relevant test cases will be identified. A good knowledge about the SPL architecture is required from the Test Architect to perform this step, since he needs to understand the change and how it impacts the whole application code.

When the correction or evolution involves structural or specification changes, some test cases (and/or scripts) need to be designed to reflect the new architecture constructs. Not
only new test cases need to be designed, but also some of them need to be redesigned (updated) to cover a specific modified portion of the architecture. An important aspect when dealing with test case update and design is how to keep track (mapping) the test case with the architecture code portion. A simple modification may impact in a large number of tests, making update tasks expensive. The more suitable is this mapping, the less escaped defects and easier number of tests making update tasks expensive. The more code portion, simple modification may impact in a large keep track of mapping the test case with the architecture when dealing with test case update and design is how to modified portion of the architecture important aspect them need to be redesigned or updated to cover a specific only new test cases need to be designed but also some of classified test cases. False positive happens when the verification activities inform that the asset is correct when it, in fact, is not; it can occur due to a wrong test case or an incomplete test set. This can be a dangerous mistake in critical systems, since we cannot double check every positive result [23]. False negative happens when verification activities indicate that the asset is not correct when it indeed is. It is safer than false positive but rather more expensive. Resources are misused to attempt to fix what is not broken [23]. Moreover, incorrect test cases may generate false negatives.

Test Suite Composition. After the test case selection and design, a test suite is composed using them. The Test Designer can create test suites grouping tests based on different information, for example, a test suite responsible for exercising a determined feature, component or even a specific functionality. This test suite will be used to build the regression test cycle to be further executed.

Test Case Prioritization. This prioritization aims to order the test cases and scripts from the test suite, executing tests with higher priority, based on criteria such as criticality or complexity of implementation, earlier than lower priority test cases. Prioritization techniques might be used, some of them taking advantage of some information about previous executed test cases to order the test suite [24]. Testers might wish to schedule test cases in a sequence that cover all the critical variabilities implementation first, exercises features from a specific product, or tests which cover a specific architectural quality attribute. Rothermel et al. in [24] analyze prioritization techniques and show that improvements can be achieved even with the least expensive of those techniques. It is truly important to apply a technique.

3) Execution: During the test execution phase, the test suites are executed against the modified version in a regression testing cycle. The Test Engineer exercises the architecture, executing the test cases. If some inconsistence is observed, he should search in the repository for a Change Request (CR) entry, previously reported, that matches the problem found. In case where no CRs are found in the repository, a new one should be created and associated to the problem found. The execution results and the new and associated CRs are reported and an investigation starts in order to precisely identify which components, modules, versions and modifications caused the failure.

Depending on the failure, the regression test approach will forward the damaged portion to unit test or integration test, for the purpose of creating a test case to cover that path.

4) Reporting: All these information will be gathered to further compose the Test Report during reporting phase. This report is extremely important for the Test Manager since he will use this information for component, architecture or product schedules and also to build other test plans.

IV. EXPERIMENTAL STUDY

The experimental study was defined using a structured process [25], which is composed by four phases: definition, planning, operation and analysis and interpretation.

The definition was based on the Goal/Question/Metric mechanism [26], [27] which was important to define and evaluate a set of operational goals using measurement. The goal of the analysis was to verify the approach applicability. In addition, metrics were collected with the purpose to improve the approach understandability, completeness, applicability and effectiveness, and minimize the risks of applying it in a real and critical scenario. To achieve the goal previously defined, some quantitative and qualitative questions were defined. They are evaluating (i) the effort to apply each approach step, (ii) difficulties to understand and apply it, (iii) some missing activities, roles or artifacts and (iv) the number of defects found and the number of correct classified test cases.

The experiment definition can be summarized as being: Analysis of the regression testing approach for the purpose of evaluation with respect to understandability, usability, completeness, applicability and effectiveness from the point of view of SPL researchers and test engineers in the context of a SPLs project.

The experiment was performed at UFPE - Federal University of Pernambuco -, Brazil, from February 2010 to March 2010, and the set of subjects was composed of eight post-graduate students in software testing area.

In order to help this evaluation, two system versions for the bank domain were developed in order to simulate a SPL architecture. The first version (V1) was developed with eighteen (18) classes and one interface, and fifty-eight (58) integration test cases used to test the conformance of the system against its specification. A second version was developed with new functionalities (simulating the evolution) and a set of seven seeded faults. This new version (V2) is composed by (24) classes and (3) interfaces. All of these
changes aimed to evaluate the regression testing approach in both scenarios during the evolution and correction.

It is important to highlight that these injected faults were inserted based on three sources: (i) McGregoer's SPL fault model [28] where he summarizes the most common faults found in SPL projects; (ii) based on the mapping study previously performed; (iii) the experimenter knowledge in the application domain.

The experiment was performed using a set of eight subjects, where each one applied the approach in both scenarios. Firstly, both code versions were provided with a set of change requests (three), as well as a set of previously designed integration test cases, for the subjects to validate the approach considering the corrective scenario. The subjects needed to apply the approach aiming to find the seeded faults, as well as to classify the integration test cases. It is important to reinforce that the steps related to graph generation and graph comparison are optional in the approach, but the subjects were asked to use them at least one time. After reporting this first result, the class diagrams (from both versions) were provided in order to characterize the evolution scenario. In this context, the subjects had to evaluate the specification changes, correctly classify the existing integration tests and create new test cases. The test cases should be designed obeying the same coverage criteria used in the previous designed integration test cases. Figure 7 summarize all experiment scenarios and provided artifacts.

![Experiment Scenarios](image)

Figure 7. Experiment Scenarios.

The results of the experiment were collected using measurement instruments. Thus, it was prepared timesheets to collect the time spent in each activity and step.

V. EXPERIMENTAL STUDY RESULTS

This section analyzes the results of the points in which the approach was evaluated.

A. Effort to Apply the Approach (EAA)

This aspect was evaluated in two scenarios: corrective and progressive. The corrective scenario spent 94.78 hours to be performed, while the progressive scenario was executed in 56.06 hours. These numbers concern the total number of worked hours of the members in each step.

Corrective Scenario. During the data collection, five outliers were identified through a box plot graphic [29], since it is recommended to visualize the dispersion and skewness of samples. However, we chose to keep these outliers and consider their times in the effort analysis due to the limited number of subjects (eight). The effort to apply the approach is shown in Tables 8. The table in the left shows the effort to apply each step, whereas the second shows only the steps that were completely and correctly reported, since some subjects did not report correctly the results or no report was provided. The EAA metric is composed dividing the time spent for each step by the total time (sum of all steps).

![Effort to apply the corrective scenario](image)

Figure 8. Effort to apply the corrective scenario.

The time spent during the planning step can be justified by the fact that no subjects had performed it previously. Since it was the first time they were working in such a project, they needed some time to understand the test plan and to collect all information in order to fill it. Besides gathering information they had to plan the test cycle considering the constraints and information provided by instrumenter.

Regarding the textual comparison step, the subjects needed to compare both code versions, and understand how the change would impact the domain application rules. They should identify portions of the code in order to discover critical paths, that would further exercised by the created and selected test cases.

Since the graph generation and comparison steps were considered as optional in our approach, only some subjects executed these steps. Some of them realized the importance of the need of such a tool in order to identify and catch language behaviors. It is important to highlight that most of the subjects complained of boredom when executing this task.

Progressive Scenario. As in the previous scenario analysis, we decided to keep all subjects pointed as outliers due to the limited number of subjects. Tables 9 show the effort of every step. The table in the left shows the effort to apply all steps, and the second shows the effort to apply the progressive scenario considering only completely and correctly reported steps.

![Effort to apply the progressive scenario](image)

Figure 9. Effort to apply the progressive scenario.

We observed that the time spent to perform the progressive scenario was lesser than the corrective one. It can be explained because all subjects firstly applied the corrective
scenario and then performed the progressive scenario. It means that, when performing the second run, the subjects had already pursued a minimum required expertise in the domain and application code, impacting the overall effort.

By analyzing the datasets, we noticed that the time spent to perform the Test Design and Selection step in progressive scenario was lesser than in corrective scenario, due to the acquired experience, or even, as reported by some subjects, the fact that during the progressive scenario the amount of retestable test cases were smaller than in the corrective scenario.

B. Approach Understanding and Application Difficulties

Analyzing subject’s answers regarding difficulties they faced during the approach execution, it was identified that 62.5% subjects had some difficulties to understand the approach. Because of the understanding problem, all of them faced problems when applying the approach. The difficulties are summarized in Table 1.

Table 1: DIFFICULTIES TO USE THE APPROACH.

<table>
<thead>
<tr>
<th>Difficulty</th>
<th>Number of subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large number of steps</td>
<td>2</td>
</tr>
<tr>
<td>Test Classification</td>
<td>3</td>
</tr>
<tr>
<td>Input/Output Identification for each step</td>
<td>2</td>
</tr>
<tr>
<td>Relation between Role and Task</td>
<td>1</td>
</tr>
</tbody>
</table>

Two subjects (ID 1, 5) claimed that the main problem to understand the approach was the number of steps and tasks. They said that the approach comprises too many steps and tasks they are very complex, which requires a large knowledge in development and testing areas. Another three subjects (ID 3,4,5) reported that one of the understandability issues was the lack of examples during test design and selection, more specifically to aid in test classification task. Other subjects (ID 3,6) stated that the input and output of each step were not clearly presented in the approach. One subject (ID 1) reported difficulties in understanding the relation between role and tasks, i.e. which tasks each role should perform.

C. Activities, Roles and Artifacts Missing

The goal with this point was to collect more information about the regression testing approach missing steps. In this sense, the subjects were asked if there was any missing step, activities, roles and artifacts. Analyzing the data, we could notice that none of the subjects identified any of them.

D. Number of Defects

By analyzing the faults found during the approach application, a dataset was structured. As it can be seen in Figure 10, all injected faults were identified at least by one subject. During the analysis, we noticed that the subjects did not report only the root cause of the issue, but they report faults in different architecture layers. For example, a fault was injected in a lower layer and it was propagated to the layers above, although the subjects reported the faults in all layers. In addition, no purposely injected faults and indentation errors were identified. These aspects should be considered in further experiments and something should be done to avoid it.

It is important to highlight that only the root cause was considered to evaluate this aspect, as well as no purposely injected faults and indentation faults were considered in this evaluation. Faults wrongly reported in the questionnaire were also not considered. All of them served as lessons learned to avoid in future experiments. Figure 10 shows the number of subjects per fault found.

By observing the subject with the best results in this aspect, we could see that all of them have more than 2 years of experience in software development. It can indicate that a high experience in development is required to apply the approach. Analyzing the subject with the best results in fault detection, the unique factor that was observed and could justify its success is the fact that, as being the first to deliver the experiment results, he executed the experiment without interruption.

Regarding the number of subjects that found a specific fault (see Figure 10), we can notice that the faults (1,2 and 7) were the most found during this experiment. It could be explained by the fact that the CRs provided by the experimenter described these faults. It also indicates that the CRs help the approach execution. In additional, no correlation was found regarding to the type of fault.

The approach has been proven efficient in fault detection, but it can not be completely generalized, since more experiment with real SPL architectures, faults and tests, and more subjects must to be performed.

E. Number of Tests Correctly Classified (NTCC)

The goal of this aspect was to evaluate how the approach is aiding the subjects during the test classification step. In this direction, the subjects were asked to classify the test cases in five categories: obsolete, reusable, retestable, new-structural and new-specification. Unfortunately, some subjects (ID 1, 3) were excluded of this evaluation since they did not report anything or wrongly report the results.

Table II summarizes the number of test correctly classified by the subjects. Not reported (NR) means that the subjects report some test cases but not correctly, and none indicates that no test cases were reported.

A set of 58 test cases were provided to the subjects, among these tests, 3 obsolete, 14 retestable, 41 reusable, 2 new-
structural and 14 new-specification. Most of the subjects complained about the absence of examples regarding each type of test. It can explain the bad results in this aspect.

We noticed that subjects with high experience in software development had better results. It can indicate that experience in development can help the process of test classification.

F. Threats to Validity

Envisioning a possible replication of this study and the generalization of the results, we have identified the following aspects: Experiment Side, Instrumentation, Subject gained experience and Java development experience.

More information about this experimental study can be seen in the dissertation WebSite.

VI. RELATED WORK

Several researchers highlighted the importance of regression testing in the context of SPL, but not systematically. McGregor in [16] reports that during products derivation, the assets are often modified to fit the products needs. The modified portion of those assets are tested using regression testing. According to Kolb [30], the major problems in testing product lines are the large number of variations, redundant work, the interplay between generic components and product specific components, and regression testing.

In [31], the authors emphasize that with the advent and use of software specifications, source code no longer has to be the single source for selecting test cases. Their particular interest has been devoted to specification-based conformance testing. The main goal of their work is to review and extend their previous work on Software Architecture (SA)-based conformance testing, to provide a systematic way to use an SA for code testing. They present a conformance testing approach, establishing a set of steps in order to test a C2 style architecture. This work also presents a case study, where the approach is applied in the elevator systems architecture.

In [3], the authors explore how regression testing can be systematically applied at the software architecture level in order to reduce the cost of retesting modified systems, and also to assess the regression testability of the evolved system. This approach addresses two goals: (i) Test conformance of a modified implementation P to the initial SA and (ii) test conformance of an evolved software architecture. To achieve these goals, a set of steps and tools were used.

In our work, we did not implement or were restricted to any test selection technique to select test cases. Instead, we studied some approaches and their characteristics in order to figure out how systematically perform regression testing in SPL architectures taking advantage of their similarities. The main difference between our work and Henry’s work [31] is that we are considering conformance testing in SPL context, considering the existing variability and not being restricted to any architectural style. Regarding the second study [3], besides it is not considering SPL context, researchers did not treat test prioritization on their work. In additional, our approach defines a systematic way to perform regression testing describing some artifacts, roles, activities and steps. We also performed an experimental study in order to evaluate our approach.

VII. CONCLUSION

In this work, we presented a systematic SPL regression testing approach and its activities, steps, roles and artifacts. An experimental study was performed using two versions of a bank system to evaluate different points of the approach, such as the existence/lack of any artifact, role or step, that should be removed and/or added, how the approach deals with progressive and corrective scenarios, its efficiency in fault detection, as well as during the test classification step.

According to our analysis, we could identify that the effort to apply the corrective scenario is greater than the progressive scenario, since the subjects reported that the specification comparison is less costly than code (textual) comparison and graph generation and analysis, we believe that this difference will be more visible in real scenarios.

After executing both methods, none activity, artifact or role were identified as missing by the subjects. Another important aspect was that all seeded faults were found at least by one subject, which indicates the efficiency of the approach in fault detection. Regarding the test classification topic, the approach showed its efficiency, but some subjects suggest improvements, such as providing examples of each test class and providing guidelines to aid during this step.

Despite of not having been experimented in the real SPL context, with real architectures, the approach showed its efficiency during its application. For future work, new experiments will be executed considering real components, modules and SPL architectures, in industrial projects. It would be better if the architectures could be suffering real evolutive and corrective modifications. In addition, guidelines to help the test classification step and tool support to aid the approach execution, are also identified as future work. Furthermore, the lessons learned in this experiment will provide important information to execute future evaluations.

ACKNOWLEDGMENT

This work was partially supported by the National Institute of Science and Technology for Software Engi-
neering (INES\textsuperscript{2}), funded by CNPq and FACEPE, grants
573964/2008-4 and APQ-1037-1.03/08

REFERENCES


