Towards Software Architecture-based Regression Testing

Henry Muccini [1]
muccini@di.univaq.it

Marcio S. Dias [2]
marcio.dias@dur.ac.uk

Debra J. Richardson [3]
djr@ics.uci.edu

[1] - Dipartimento di Informatica, University of L'Aquila, Italy
[2] - Department of Computer Science and e-Science Research Institute, University of Durham, UK
[3] – Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California Irvine, USA

ABSTRACT

When architecting dependable systems, in addition to improving system dependability by means of construction (fault-tolerant and redundant mechanisms, for instance), it is also important to evaluate, and thereby confirm, system dependability. There are many different approaches to evaluating system dependability, and testing always has been an important one.

Previous work on software architecture testing has shown it is possible to apply conformance-testing techniques to yield confidence that the behavior of an implemented system conforms to the expected behavior of the software architecture, specified with Architecture Description Languages (ADLs).

In this work, we 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. We consider assessing both "top-down" and "bottom-up" evolution, i.e., whether a slightly modified implementation conforms to the initial architecture, and whether the (modified) implementation conform an evolved architecture. A better understanding on how regression testing can be applied at the software architecture level will help us to assess and identify architecture with higher dependability.

Keywords: Software Architecture, Dependable Systems, Regression Testing, Architecture-based Testing and Analysis, Architecture-level Test Cases.

1. INTRODUCTION

A Software Architecture (SA) [6] specification captures system structure (i.e., the architectural topology), by identifying architectural components and connectors, and required system behavior, designed to meet the system requirements, by specifying how components and connectors are intended to interact. Software architectures can serve as useful high-level "blueprints" to guide the production of lower-level system designs and implementations, and later on for guidance in maintenance and reuse activities. Moreover, SA-based analysis methods provide several value added benefits, such as system deadlock detection, performance analysis, component validation and much more [5]. Additionally, SA-based testing methods are available to check conformance of the implementation's behavior with SA-level specifications of expected behavior [4] and to guide integration testing [5, 14].

Reaping these architectural benefits, however, does not come for free. To the contrary, experience indicates that dealing with software architectures is often expensive perhaps even too expensive, in some cases, to justify the benefits obtained. For example, consider the phenomenon of "architectural drift" [18]. It is not uncommon during evolution that only the low-level design and implementation are changed to meet tight deadlines, and the architecture is not updated to track the changes being made to the implementation. Once the architecture "drifts" out of conformance with the implementation, many of the aforementioned benefits are lost: previous analysis results cannot be extended or reused, and the effort spent on the previous architecture is wasted. Moreover, even when implementation and architecture are kept aligned, SA-based analysis methods often need to be re-run completely from the beginning, at considerable cost, whenever the system architecture or its implementation change.

SARTE (Software Architecture-based Regression TEsting) is a collaborative project among three universities focused on providing a framework and approach for SA-based testing in the context of evolution, when both architecture and implementation are subject to change. The topic of architecture-based testing has been extensively analyzed by one of the authors in [14], where a general framework for software architecture-based conformance testing has been proposed. A software architecture-based testing and analysis toolset (Argus-I) was developed by two of the authors, as described in [4]. SARTE builds upon the research and development in both previous projects.

In this context, this paper shows how SA-based regression testing provides a key solution to the problem of retesting a SA after its evolution. In particular, after identifying SA-level behavioral test cases and testing conformance of the code with respect to the expected architectural behaviors [14], we show what should be tested when the code and/or architecture is modified and how testing information previously collected may be reused to test the conformance of the revised implementation with respect to either the initial or revised architecture. We describe, in general terms, i) how implementation-level test cases may be reused to test the conformance of modified code with respect to the architectural specification, and ii) how to reuse architecture-level test cases when the architecture evolves. Our approach relies on reusing and modifying existing code-level regression testing (RT) techniques

2. MOTIVATIONS AND GOALS

Regression testing may be used during development, to test families of similar products, or during maintenance, to test new or modified configurations [9]. Although architecture-based regression testing may be used for both purposes, we focus here on the maintenance aspect, being confident this approach may be used during development as well.

Software architectures may change over time due to the need to provide more dependable systems, the need to remove identified deficiencies, or the need to handle evolving collections of components at runtime [6]. Much research has investigated architecture evolution, especially at runtime. In [17], for example, the authors analyze how software architecture may change at runtime (in terms of component addition, component removal, component replacement, and runtime reconfiguration) and how

tool suites may be used to cope with such evolution. In [1] the authors describe an approach to architectural specification that permits the representation and analysis of dynamic architectures. In [11] the authors analyze the issues of dynamic changes to a software configuration, in terms of component creation/deletion and connection/disconnection. In [13] the authors analyze such Architecture Description Languages (ADLs), which provide specific features for modeling dynamic changes.

Many functional and non-functional analysis techniques have been proposed to operate at the software architecture level. The drawback is, however, that given that the architecture may evolve, current techniques require that SA-based analysis be completely re-run from scratch for a modified SA version, thereby increasing analysis costs and reducing benefits. To mitigate this drawback, we propose here to apply regression testing at the SA level in order to lower the cost and greatly improve the cost-benefit properties of SA-based testing.

The benefits we expect are manifold: i) the selection of architecture level test cases and their execution at the code level is a long and expensive process (as described in [14] and summarized in Section 4.1). Reusing previous results as much as possible may strongly reduce testing effort while testing a modified architecture. Quoting [8], in fact, "regression testing can account for as much as one-third of the total cost of a software system … the use of software architecture for regression testing activities has the potential for a bigger impact on the cost of software"; ii) SA-based testing may identify (functional) errors that are likely to be missed when applying traditional code-level testing, thus complementing traditional testing techniques. Similarly, SA-based regression testing may valuably complement code-based regression testing. Moreover, many specification-based regression testing approaches have been proposed so far to complement traditional regression testing approaches.

SARTE's intermediate project goals are depicted in Figure 1, where the left side embodies our first goal and the right side embodies the second goal:

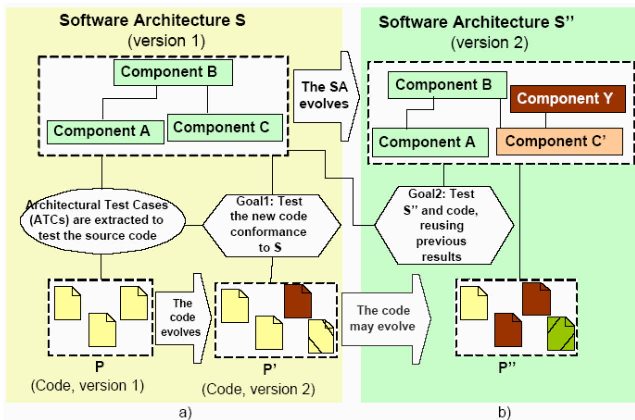

Figure 1. Project goals: a) the implementation evolves; b) the software architecture evolves.

Goal 1: *Test Conformance of a Modified Implementation P. to the initial architecture.*

***Context*:** Given a software system, a software architecture specification for this system S, and an implementation P, we first gain confidence that P correctly implements S. During maintenance, a modified version of the code (P) is implemented - where some components from P' remain, some components are modified, and/or some new components are introduced.

***Goal*:** Test the conformance of P'. with respect to S, while reusing previous test information for selective regression testing, thereby reducing the test cases that must be retested.

Goal 2: *Test Conformance of an Evolved Software Architecture.*

Context: Given a software system, a software architecture specification for this system S, and an implementation P, we have already gained confidence that P correctly implements S. Suppose evolution requires a modified version of the architecture (S") – where some architecture-level components are kept, others are modified, and/or new ones are introduced and consequently a modified implementation P" may have also been developed.

Goal: Test the conformance of P" with respect to S", while reusing previous test information for selective regression testing, thereby reducing the test cases that must be retested.

In the rest of this paper we address both goals, by proposing an approach to integrate existing code-level regression test techniques with SA-based regression test and by exploiting similarities and differences between software architecture versions.

A different goal is to reconstruct the actual architecture when the first goal determines that the code no longer conforms to the initial architecture. This is a sort of reverse-engineering activity that could mitigate the architectural drift problem in a more general context. This goal will be analyzed in future work.

3. REGRESSION TESTING

Here, we briefly introduce how traditional code-based regression test selection techniques work, in order to identify the basis for an SA-based regression test selection approach. We focus on selective regression test techniques and then reuse the same logical steps to propose a software architecture-based regression testing approach in Section 4.

Regression testing "attempts to validate modified software and ensure that no new errors are introduced into previously tested code" [9]. The traditional approach is decomposed into two key phases: i) testing the program P with respect to a specified test suite T, and ii) when a new version P' is released, regression testing of the modified version P' to provide confidence that P' is correct with respect to a test set T'.

In the simplest regression test technique, called *retest all*, T' contains all the test cases in T, and P' is run on T'. In regression test selection techniques, T' is selected as a "relevant" subset of T, i.e., assuming that t ε T, t is included in T' if there is the potential that it could produce different results on P' than it did on P (following a safe definition). An empirical study and analysis on different regression test selection techniques has been proposed in [7], together with the identification of the different activities required. Specification-based regression test techniques, which are also guiding our research, will be outlined in Section 7.

In this paper, we focus on how to select a subset of relevant test cases for P', known as the regression-test-selection problem, and characterizing a regression test selection technique, but at the SA-level rather than at the code-level. In other words, instead of selecting code level test cases, our approach selects architectural level test cases [16].

4. SA-BASED REGRESSION TESTING

Our SA-based regression testing inherits the two-phased decomposition from traditional RT approaches, therefore

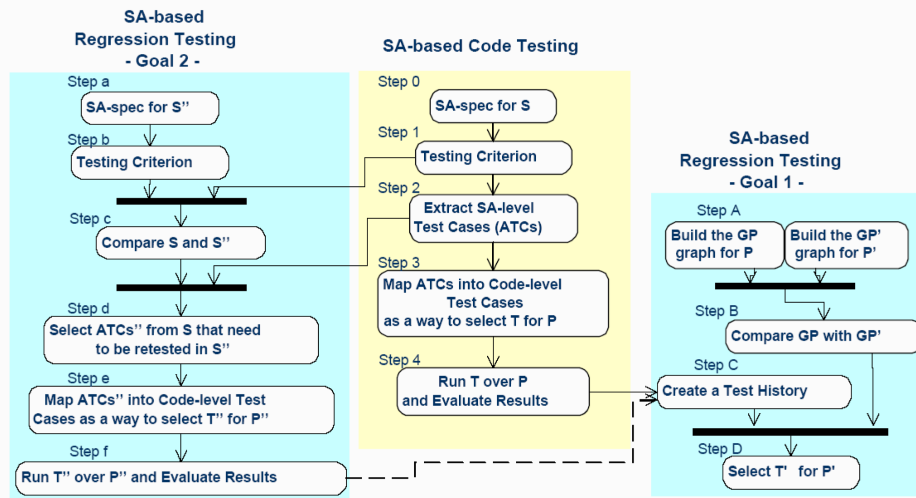

Figure 2. Activity diagram for our software architecture-based regression testing approach

using the architectural model as a test oracle to identify when a test case fails or succeeds.

This approach has been demonstrated to work well for SA, but repeating the entire testing activity whenever the system evolves is undoubtedly too expensive, thus making SA-based testing less appealing. Here we propose an approach to deal with system evolution, which reuses previous test results to retest the modified architecture and/or implementation with reduced effort.

comprising the following two phases: i) *SA-based testing*. In particular, we apply a SA-based conformance testing approach, ii) *SA-based regression test selection*. This phase is decomposed to meet Goal 1 and Goal 2 (discussed in Section 2.)

Figure 2 summarizes the activities required by SA-based conformance and regression testing. While SA-based conformance testing has been already analyzed in [14, 16], goal of this paper is to focus on SA-based regression testing.

4.1. SA-based Conformance Testing

As mentioned previously, this work builds upon the general framework for SA-based conformance testing set forth in [14], whose goal is to test implementation conformance to a given software architecture.

The framework encompasses five different steps, as shown in the middle section of Figure 2.

Step 0) It begins with a topological and behavioral specification of the SA, where behavior is modeled through a state machine-based formalism; in the following, we use Labeled Transition Systems (LTS) to model component behavior.

Step 1) An observation function is introduced in order to implement a testing criterion that looks at the SA from a perspective deemed to be relevant for testing purposes, while hiding away non-relevant actions from this perspective. The LTS model is abstracted, producing an Abstract LTS (ALTS), in order to show only such high-level behaviors/components we want to test.

Step 2) An architecture-level test case (ATC) is defined as an ordered sequence of architectural events we expect to observe when a certain initiating event is performed. This definition encompasses two different keywords: the sequence of actions, which represents expected behaviors, and the initiating event, that is, the architectural input which should allow the sequence to happen. Deriving an adequate set of ATCs entails deriving a set of complete paths that appropriately cover the ALTS.

Step 3) Naturally, such ATCs strongly differ from executable code-level test cases, due to the abstraction gap between SA and coding (the traceability problem). We deal with this problem through a “mapping” function which maps SA-level functional tests into code-level test cases.

Step 4) Finally, the code is run over the identified test cases. The execution traces are analyzed to determine whether the system implementation works correctly for the selected architectural tests,

4.2. Goal 1: Test Conformance of P' to the initial SA

Let us assume SA-based conformance testing has provided confidence that the implementation P conforms to a given SA. After evolving P to P' (Figure 1.a), how can we test the conformance of P' to the same architecture?

The approach we take here is based on the idea of integrating SA-based testing techniques with existing code-based RT techniques. Figure 2 (right side) summarizes, through an activity diagram, how code-level RT may be integrated with SA-level conformance testing to select a new test suite T':

Step A: Generate P graph (GP). Most common approaches to code regression testing are to structurally represent P using a graph. After modification, P' is also represented as a graph. In the software architecture, the graph is obtained by composing the LTS models of component behavior, following the structural organization of those components in the architecture.

Step B: Compare GP with GP'. As in traditional code based testing, the code graphs for P and P' are compared, to identify how code changes are reflected in the graph. In the software architecture, the changes are identified based on differences in LTS nodes and edges (including symbols).

Step C: Record Coverage. A test history report is built during test case execution. It records a trace of which nodes/arcs have been covered by the test case t during its execution over P.

Step D: Test Case Selection for P'. The information gathered from the test history and the graph comparison is used to identify test cases in T to be rerun on P'. If execution of P on t ε T covers a node which is modified in P', then t needs to be rerun on P'.

Once T' is selected, t' ε T' is run over P' and results are collected and compared with an oracle to decide if the test failed or succeeded. One of the main differences with respect to traditional code-based techniques is that the oracle in SA-based RT is the software architecture specification itself (as happens in specification-based RT). In fact, when t' is run on P', the test fails if its execution does not allow the expected behavior to be reproduced (i.e., the architecture-level test case associated to t'). Moreover, code-level test cases are always driven by well formalized functional and structural architectural requirements.

The advantages we expect from this approach are twofold: i) as in traditional RT, we reduce the size of the test suite for P', eliminating all those tests which do not need to be reapplied to P', and ii) when conformance faults are detected, we can gather information on how to adjust the initial architecture.

4.3. Goal 2: Test Conformance of Evolved Architecture

Let us assume again that SA-based conformance testing has demonstrated that the implementation P conforms to a given SA. After evolving S into S" (Figure 1.b) how can we test the implementation's conformance to the new architecture?

The approach we take here is based on the idea of comparing the two architectural specifications to identify changed/unchanged portions of the SA. Both structural and behavioral changes are taken into account. In particular, the LTSs for S and S" are compared, and differences are identified in the two graphs (using a sort of "diff" algorithm). In a fashion similar to traditional code-based RT, whenever an ATC traverses a path in the SA LTS modified in the S" LTS, then it needs to be retested in S". Figure 2 (left most side) summarizes how Goal 2 may be realized through different activities:

Step a: New SA specification. The architectural specification of the evolved system S" is provided.

Step b: Testing Criterion. The testing criterion (previously applied to S) is applied to S".

Step c: Comparison. The architectural specifications are compared to identify topological changes (i.e., added/deleted components or modified configuration) and behavioral changes (i.e., modified components).

Step d: Select Architectural Test Cases for S". Those ATCs from SA affected by the architectural changes are selected to be retested in P", the implementation for S". Notice that any ATC discarded in this step may represent many code-level test cases that were eliminated, thus strongly reducing retesting effort.

Step e: Identify Code-Level Test Cases. Once we have identified the ATCs which need to be regression tested for S", we need to map these architectural test cases to code-level test cases for S", in order to select T" for P". This step is analogous to Step 3 in SA-based testing.

Step f: Test Execution. After T" has been selected for S", we need to run T" over P" and evaluate the result of the performed SA-based regression testing. This step is analogous to Step 4 in SA-based testing.

5. LESSONS LEARNED

Based on some experiments performed over Cargo Router and Elevator systems [15], we were able to identify the following lessons learned:

a) Goal 1: Testing Criterion: Code-level test cases cover only portions of the code significant for the selected testing criterion - that is, they cover only features and components selected by the testing criterion. When code changes affect features and code portions that have been hidden by the selected testing criterion, test cases might not traverse such changes, avoiding test case (re)selection.

b) Goal 2: Architectural test case to be retested: The application of Goal 2 in our experiments has highlighted the following results: i) if an architecture test case (ATC) covers only components that have not been changed when moving from S to S", then the ATC does not need to be retested (in other words, code level test cases generated from and associated with that ATC need not be retested). ii) If an ATC covers a component C modified in S", this ATC may need to be retested for S". However, if C has been hidden by the testing criterion, it may happen that diff(ALTS,ALTS") does not identify any real change (with respect to what we are interested in testing), thus avoiding selection of the ATC.

c) Goal 2: Architectural Diff: In previous experiments [20], the Diff technique was based on deriving and then comparing the ATCs derived from the two architectures to one another, instead of comparing the SAs themselves. That solution had two drawbacks: checking ATC" against ATC may be a bottleneck when the number and sizes of paths become large, and there may be paths in ATC" that don't exactly match any paths in ATC, even though they test behavior already tested in ATC. More recently, in a preliminary version of the current SA-based diff idea, we tried to make a structural-only comparison between S and S" (without considering the behavioral specification). We find out that this comparison is too coarse-grained. In fact, assuming a component C becomes C' is not enough to conclude that any ATC covering C needs to be re-executed in S", exactly like assuming a class X is modified into X' is not enough to conclude that a test case covering X needs to be re-executed in traditional RT techniques.

d) Semantic and Syntactic Check: It is interesting to notice that while Goal 1 drives a functional test by a syntactic/structural diff comparison between P and P' (i.e., if P' differs from P, we need to retest the functional conformance of P' to SA), in Goal 2 the functional diff between S and S" drives a structural check at the code level (i.e., if the behavior of S" differs from S, then some test cases need to be rerun on P").

e) C2 style framework: It is important to note that the research proposed here is not tied to C2 and its implementation framework. In fact, C2 is used in our work as representative of all such frameworks and architectural languages (e.g., Aesop, Darwin, Rapide, MetaH), which support a rigorous SA-based coding process [13]. The assumption of having a precise mapping between architectural elements and implementation is not too restrictive, and already investigated by some researchers (e.g., [6]).

f) Identifying new test cases: When dealing with RT techniques, one typical activity consists in identifying new test cases to test new features or statements. This activity should be handled in a SA-based RT too. Preliminary experiments show that when moving from S to S", previously defined testing criteria may be applied to the revised SA to identify new ATCs, and new testing criteria could be added to test newly introduced features or components.

6. RELATED WORK

For a description of existing SA-based testing techniques we direct interested readers to [14]. Useful references on regression testing approaches have been provided in Section 3. Here, for sake of space, we outline some relevant research on specification-based regression testing.

Existing techniques for specification-based regression testing either extract the specification from the source code or assume source code availability. Papers on this topic work at i) the source code level or at ii) the component level. The approaches proposed in [12, 19, 2] extract the system specification from the code through a reverse engineering approach. These approaches differ from our in two major ways. Our approach to test generation is driven by an independently developed specification of the software architecture, which also serves as an oracle to describe the expected behavior. As a result, the originating abstraction level of the specification is different (implementation-level rather than the architecture-level specification in our approach). Second, a shortcoming of these approaches is that any defect in the implementation is also reflected in the re-verse engineered specification.

Some approaches to regression testing at the component level have been presented [10, 20]. In [10] the authors reuse the idea of category partition testing and develop test frames for each component. Metadata packaged within components provides the appropriate information when the component code is unavailable. In [20], the test model the authors present is realized using the Component Interaction Graph (CIG), which depicts the interactions and dependence relationships among components.

To the best of our knowledge, [8] is the only previous paper presenting ideas on using software architecture for effective regression testing.

7. FUTURE WORK

This research paper has proposed a first attempt to handle the retesting of a software system during evolution of both its architecture and implementation, while reducing the testing effort.
We initially analyzed the different activities to be performed in a software architecture-based RT process. This high-level analysis has been refined for C2 style architectures. We examined the case where the code evolved relative to unaffected software architecture and we explored the case where the architecture evolved and not only the implementation. In both situations, the proposed approach was applied to the Cargo Router case study and some results have been collected.

Our approach is tool supported by the LTSA tool, the C2 framework, the Argus-I environment and the UNIX diff utility.
In future work, we are planning to consider the following extensions/improvements. In the immediate future, we want to complete the regression testing process by considering how to reconstruct the actual architecture when the first goal determines that the code no longer conforms to the initial SA. We are also planning to improve and implement the architectural “diff” algorithm.

In the long term, we plan to scale our approach from C2 style architectures to generic architectures implemented through component-based and middleware technology. In particular, we would like to expose the Siemens C.N.X. architecture model-checked and tested in [3] to SA-based RT.

REFERENCES

1. R. J. Allen, R. Douence, D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In Proc. of the 1998 Conf. on Fundamental Approaches to Software Engineering (FASE ’98), March 1998.
2. S. Beydeda, and V. Gruhn. Integrating White-and Black-Box Techniques for Class-Level Regression Testing. In Proc. COMPSAC 2001, pp. 357-362.
3. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In Proc. Int. Workshop on Integration of Testing Methodologies, ITM ’04. October 2004.
4. Dias, M., Vieira, M., Richardson, D., Analyzing Software Architecture with Argus-I, In Proc of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000, pp. 758-761.
5. Formal Methods for Software Architectures. Tutorial book on Software Architectures and Formal Methods. In SFM-03:SA Lectures, Eds. M. Bernardo and P. Inverardi, LNCS 2804, 2003.
6. D. Garlan. Software Architecture: a Roadmap. In A. Finkelstein (Ed.), ACM ICSE 2000, The Future of Software Engineering, pp. 91-101, 2000.
7. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. In Proc. of the 20th Int. Conf. on Software Engineering (ICSE’98), pp. 188-197, Japan, April 1998.
8. M. J. Harrold. Architecture-Based Regression Testing of Evolving Systems. In Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis (ROSATEA), CNR-NSF, pp. 73-77, July 1998.
9. M. J. Harrold. Testing: A Roadmap. In A. Finkelstein (Ed.), ACM ICSE 2000, The Future of Software Engineering, pp. 61-72, 2000.
10. M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using Component Metadata to Support the Regression Testing of Component-Based Software. In Proc. Int. Conf. on Software Maintenance, Nov. 2001.
11. J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A case study. In Proc. 4th IEEE Int. Conf. on Configurable Distributed Systems, Annapolis, May 1998.
12. D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim, and Y. Song Developing an Object-Oriented Software Testing and Maintenance Environment. In Comm. of the ACM, Oct 1995.
13. N. Medvidovic, and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Trans. on SE, vol. 26, no. 1 (Jan 2000).
14. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. In IEEE Trans. on SE. Vol. 30, Issue N. 3, March 2004, pp. 160-171.
15. H. Muccini, M. Dias, and D. Richardson. The SARTE Project: Software Architecture-based Conformance & Regression Testing. - <http://www.henrymuccini.com/Projects.htm>.
16. H. Muccini, M. Dias, D. Richardson. Systematic Testing of Software Architectures in the C2 style. In Proc. Fundamental Approaches to Software Engineering (FASE ’04), ETAPS 2004, March 2004.
17. P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-Based Runtime Software Evolution. In Proc. Int. Conf. on Software Engineering, ICSE 98, Kyoto, Japan, April 1998.
18. D. E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. In SIGSOFT Software Engineering Notes 17,4 (October 1992): 40-52.
19. Y. Wu, M. H. Chen, and H. M. Kao. Regression Testing of Object-Oriented Programs. In Proc. Int. Symposium on Software Reliability, 1999.
20. Y. Wu, D. Pan, and M. H. Chen. Techniques of Maintaining Evolving Component-Based Software. In IEEE Proc. Int. Conf. on Software Maintenance, 2000.