ISSN No. 0976-5697

Volume 3, No. 5, Sept-Oct 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Applying ReTesting Analysis Techniques:A Case Study For A Modified Systems

Dr. G. Mahadevan*
Dept of Computer Applications
AMC Engineering College, Bangalore, India
g_mahadevan@yahoo.com

Badri H.S
Dept of Computer Applications
Presidency College, Bangalore, India
badriiyengar@yahoo.com

Abstract: Software architectures are becoming very important to the development of quality systems. When developing dependable systems, it is very important to evaluate and confirm system dependability. Testing is one of the main approach for evaluating system dependability.
Previous work on software architecture –based testing has shown it is possible to apply conformance testing techniques to yield, some confidence on the implemented system conformance to expected, architecture-level, behaviours.
This work explores how regression testing can be applied systematically at the architecture level in order to reduce the cost of retesting modified systems , and also assess the regression testability of the evolved system, assessing both slightly modified implementation conforms to the initial architecture, and whether the implementation continues to conform to an evolved architecture.

Keywords: Software architecture, System Dependability, Regression Testing, Reliability, Architecture –based testing

I. INTRODUCTION

A component-based software system is an assembly of reuse components, designed to meet the quality attributes identified during the architecture phase. Components are specified, designed and implemented with the intention to be reused, and are assembled in various contexts in order to produce a multitude of systems.

Component-based software development (CBSD) or component-based software engineering (CBSE) is concerned with the assembly of pre-existing software components into larger pieces of software. Underlying this process is the notion that software components are written in such a way that they provide functions common to many different systems. Borrowing ideas from hardware components, the goal of CBSD is to allow parts (components) of a software system to be replaced by never, functionally equivalent, components.

Component-based software development encompasses two processes:

a. Assembling software systems from software components and
b. Developing reusable components.

The activity of developing systems as assemblies of components may be broadly classed in terms of four activities;

a. component qualification
b. component adaptation
c. component assembly
d. system evolution and maintenance

The quality of a component-based system strongly depends on both the quality of the assembled components, and on the quality of the assembly and its subsumed architecture. While the quality of a single component can be analyzed in isolation, the quality of the assembly can be verified only after components integration. While in the past verification stage to be properly performed required the assembly of already developed components, with the advent of model-driven development, the models themselves may be analyzed before components are developed or bought. In particular, a software architecture (SA) specification of a component-based system plays a major role in validating the quality of the assembly.

A Software Architecture [1] specification captures system structure (i.e., the architectural topology), by identifying architectural components and connectors, and required system behavior, designed to meet system requirements, by specifying how components and connectors are intended to interact. In a component-based context, SA provides an high-level blueprint on how components are supposed to be have when integrated in a certain system. Moreover, SA-based analysis methods provide several value added benefits, such as system deadlock detection, performance analysis, component validation and much more [2]. Additionally, SA-based testing methods are available to check conformance of the implementation's behavior with SA-level specifications of expected behavior and to guide integration and conformance testing.

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". 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 mentioned 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 rerun completely from the beginning, at considerable cost, whenever the system architecture or its implementation change.

Software architecture asserts that architecture is not just a phase or an activity in the software development life cycle, but a discipline pervading all phases of development. The architecture can be defined as the set of principal design decisions about a system; The study believes that integrating the discipline of architecture into the development process has the potential to increase the quality of software produced while reducing both the costs of development and the time to market.

A. Motivation and goals:

This section describes why Software Architecture Based Regression Testing [SARTE] can contribute to improve the overall system dependability.

a. SARTE Motivations:

Regression testing permits to test modified software to provide confidence that no new errors are introduced into previously tested code. It may be used during development, to test families of similar products, or during maintenance, to test new or modified configurations. Although SA-based RT may be used for both purposes, the focus is on the maintenance aspect, being confident that this approach may be used during development as well.

In this section analysis is i) why a software architecture may change due to maintenance or evolution, and ii) why regression testing at the architecture level is a relevant discussion.

a) Software Architectures change: Software architectures may change over time, due to the need to provide a more dependable system, the need to remove identified deficiencies, or the need to handle dynamically evolving collections of components at runtime. Much research has investigated SA evolution, especially at runtime. In, for example, the authors [3] [4] [5] [8] analyzed how an 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 [3] [6] [7] the authors describe an approach to specify architectures that permits the representation and analysis of dynamic architectures. In the authors analyzed the issues of dynamic changes to a software configuration, in terms of component creation and deletion, and connection and disconnection. In the authors analyzed such Architecture Description Languages which provide specific features for modeling dynamic changes.

b) Reason For SA-based Regression Testing: Many functional and non-functional analysis techniques have been proposed to operate at the SA-level [2]. However, the drawback is that (given that an architecture may evolve) current techniques require that SA-based analysis be completely rerun from scratch for a modified SA version, thereby increasing analysis costs and reducing benefits**.** To mitigate this drawback, the proposal here is 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.

SARTE's intermediate project goals are depicted in figure 1.1, where the left side embodies the study first goal and the right side embodies the second goal of the study.

b. SARTE Goals (Software Architecture-based Regression Testing)**:**

Goal 1: Test Conformance of a Modified Implementation P0 to the initial SA:

a) Context: Given a component-based software system, a software architecture specification S, and an implementation P, the confidence that P correctly implements S is gained. During maintenance, first a modified version of the code (P') is implemented where some components from P remain, some components are modified (for example, by adding/removing internal objects or interfaces).

b) 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.

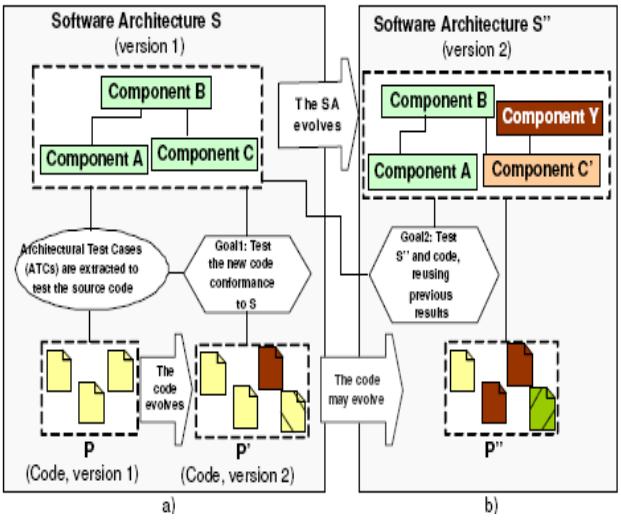


Figure 1.1 Project goals:

a) the component based system implementation evolves

b) the software architecture evolves

Goal 2: Test Conformance of an Evolved Software Architecture:

a) Context: Given a software system, a software architecture specification for this system S, and an implementation P, and 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 component-based implementation P**"** may have been also developed.

b) ***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.

II. THE CARGO ROUTER SYSTEM EXAMPLE

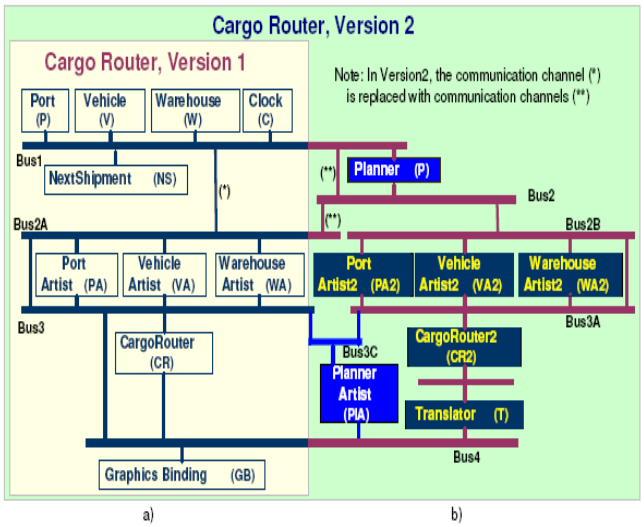
The Cargo Router system is a logistic system which distributes incoming cargo from a set of delivery ports to a list of warehouses. The cargo is transported through different vehicles, selected from a list of available ones and depending on some parameters (e.g., shipment content, weight, delivery time).

When a cargo arrives at an incoming port, an item is added to the port's item list, with information on cargo content, product name, weight and time elapsed since arrival. End- users, looking at warehouses and vehicles status, route cargo by selecting an item from a delivery port, an available vehicle, and a destination warehouse.

Figure 1.2 shows two different architectural specifications of the Cargo Router system. In the remainder of this case study, it is assumed that the architectural specification is written in accordance with the C2 style rules [2].

Figure 1.2a realizes the above mentioned features through the following components:

Port (P), Vehicle (V), and Warehouse (W) components are ADTs keeping track of the state of ports, the transportation vehicles, and the warehouses, respectively. The Port Artist (PA), Vehicle Artist (VA), and Warehouse Artist (WA) components are responsible for graphically depicting the state of their respective ADTs to the end-user. The Cargo Router (CR) component determines when cargo arrives at a port and keeps track of available transport vehicles at each port. The Graphics Binding (GB) component renders the



drawing requests using the Java AWT graphics package. The Next Shipment (NS) component regulates the incoming of new cargo on a selected port. The Clock (C) sends ticks to the system.

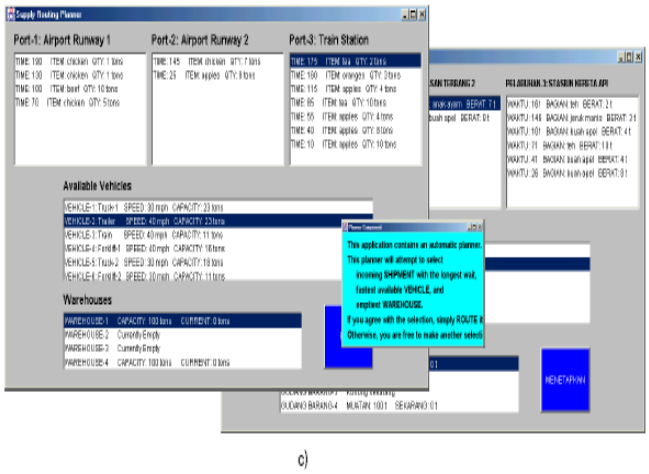


Figure 1.2 The Cargo Router system:
a) SA version 1 (S); *b*) SA version 2 (S**”**); c) GUI

Figure 1.2a+b shows an evolution of the initial architecture (Cargo Router, version 2); it realizes a graphical interface, through the duplication of the artists and cargo router components, and the introduction of the Translator (T) component, which supports translating the contents in the original windows to a different language. Moreover, this new architecture contains an automatic Planner feature (implemented through the Planner (P) and Planner Artist (PlA) components), which automatically selects the incoming shipment with the longest wait, fastest available vehicle and emptiest warehouse.

Figure 1.2c illustrates the graphical user interface. The top pane identifies the incoming ports, the mid pane lists the available vehicles, while the bottom pane shows the destination warehouses. The right most windows informs an automatic planner is in place.

It is important to note that the research proposed here is not tied to C2. However, the approach is to instantiated to this context since C2 supports a rigorous SA-based coding process and provides tool support for analyzing and monitoring software architectures.

III. SA-BASED REGRESSION TESTING

Software architectures are becoming centric to the development of quality software systems, being the first concrete model of the software system and the base to guide the implementation of software systems. 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 for evaluating system dependability, and testing has been always an important one, being fault removal one of the means to achieve dependable systems.

Previous work on software architecture-based testing has shown it is possible to apply conformance testing techniques to yield some confidence on the implemented system conformance to expected, architecture-level, and behaviors. However the proposed SA-based regression testing inherits

the two-phased decomposition from traditional regression testing approaches, therefore comprising the following two phases:

A. SA-based conformance testing:

In particular, a SA-based conformance testing approach is applied whose goal is to test the implementation conformance to a given software architecture.

B. SA-based regression test selection:

This phase is decomposed to meet Goal 1 and Goal 2 identified.

Figure 1.4 summarizes the activities required by SA-based conformance and regression conformance and regression testing. A step-by-step (theoretical) description of the approach as stated is provided, and also described it through its application to the Cargo Router running example as stated below.

Method 1 briefly describes how the SA-based testing has been implemented. Method 2 describes how to retest a modified implementation of the initial SA (Goal 1) and a modified SA (Goal 2) respectively.

C. Method 1:

a. SA -Based Testing applied to the Case Study:

Following the five steps depicted in figure 1.3, in step (0), specified the Cargo Router topology using the C2 style architecture through the Argus-I tool. The system behavior has been modeled by Labeled Transition System (LTSs) (one for each component), specified through Finite State Process algebra (FSP) and drawn by the LTSA tool. The Cargo Router v 1 specification consists of 190 lines of FSP statements and the resulting global LTS is composed by 21,144 states and 133,644 transitions. Following Step(1) defined a testing criterion to focus on "all those behaviors generated by routing events" (hereafter called, Routing Criterion). By focusing on the Routing Criterion, identified a more selective/abstract LTS (called ALTS), composed by 80 states and 244 transitions. From this ALTS, further identified 164 architecture-level test cases (ATCs) using McCabe's path coverage criterion.

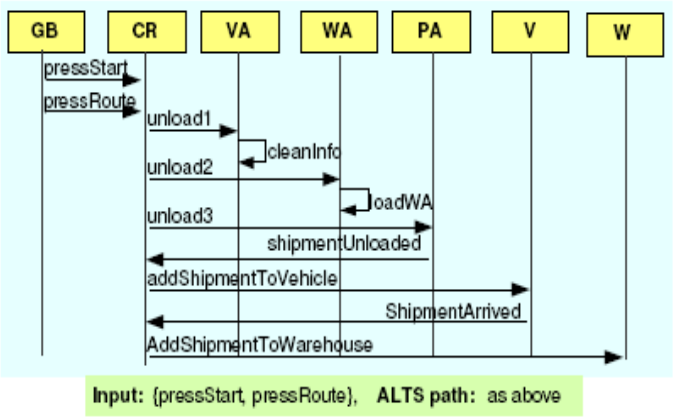


Figure 1.3 Architectural Test Case ATC #42
Figure 1.3 shows one of the ATCs previously identified.

To map SA-level ATCs to code-level test cases the C2 framework which dictates how architectural components are implemented by java components are used. The mapping between architectural test cases and code-level test cases is systematic and tool supported, as analyzed in.

It is important to note that executing the system with certain inputs may require more information than just the architecture-level inputs. This is why parameters and environmental conditions must be used when mapping ATCs to code-level test cases. The ATC in figure 1.2, for example, has been mapped to six different code-level test cases.

In the study the final used is the Argus-I tool monitoring and debugging capabilities to make a deterministic analysis of the code and observe the desired sequence. At the end of analysis, the study identified no architectural errors at the code level

D. Method 2:

Goal 1: Test Conformance of a Modified Implementation P' to the initial SA:

In the previous phase, the SA- based conformers testing has provided confidence that the implementation P of a component based system conforms to a given SA. After modifying the system implementation P into P' (figure 1.1.a), it is needed to test the conformance of the new implementation P'to the initial architecture.

Following the four steps depicted in the figure 1.4 b, taken into consideration to different implementations of the Cargo Router system: P1**'** which modifies the use of the "random function" in class Vehicle java to select (at startup) vehicles available for shipments, and P2**'** includes a new feature that supports visualizing "Shipments in Progress" – that is, vehicles, warehouses and shipments in use at a given time. Some faults have been also injected into P2**'.**

In order to regression test such implementations, the applications here is based on the concepts reported in the JDiff algorithm by hands. By building a graph representation of P, P1**'** and P2**'** (step A) and comparing two pairs of implementations: (P, P1') and (P, P2') (Step B), able to discover four lines changed between (P, P1'), all local to a single method. The study further discovered 25 changes in moving from P to P2**'**, changes that involved two different components and four different methods.

Further then manually instrumented those P's methods subject to change in P1**'** and P2**'**. The instrumentation simply prints a message to mark the changed method/lines as traversed (Step C). Finally ran P over a subset of the code test cases T previously selected. When P is run over T, the study discovered that the changed method in P1**'** is never traversed. This means that all such test cases do not have to be rerun on P1**'**.

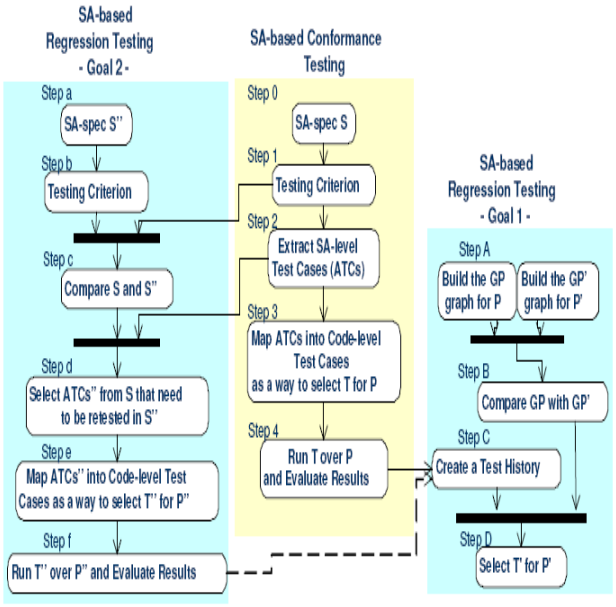


Figure 1.4 Activity Diagram of SA-based Regression Testing approach

Even if not necessary, some of them are re-runned, without identifying any conformance errors. Further the study also discovered that eight of the 18 code-level test cases runned did not cover any changed method in P2**'** and thus retested only ten of the 18 test cases. When retesting such test cases, all of the injected faults are also identified. To conclude the experiment, further retested the eight discarded test case. None of them revealed any architectural error.

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

In this section, the SA- based conformance testing approach has demonstrated that P confirms its SA. After evolving the architecture S into S**"** (figure 1.1b), the study is, to check the component-based implementation's conformance to the new architecture.

The approach taken here is based on the idea of comparing the two architectural specifications (both structural and behavioral) to identify changed/ unchanged portions of the SA.

Following figure 1.4 c, described here the different steps in Goal 2:

Steps a-b: S" specification and Testing Criterion:

The Cargo Router version2specification consists of 305 lines of FSP statements and the resulting global Transition System (LTS**")** is composed by 360,445 states and 869,567 transitions. By focusing on the Routing testing criterion .The study produced an ALTS composed by 8,448 states and 55,200 transitions.

Step c: Comparing S with S":

In this state the study considers that a software architecture changes when a new component/connector is added, removed, replaced or the architecture is reconfigured. In present context, both C2 structural and FSP behavioral specifications are used to compare architectures. When moving from S to S**"** in figure 1.4, the following differences are indicated.

a. Architecture Reconfiguration: Another instance of the artists components (PA2, VA2, WA2) and of the cargo router (CR2) have been added to produce the graphical user interface (GUI).

b. Added components: The Translator component has been added to translate contents. The Planner and Planner Artist components have been added to allow the automatic routing feature.

Added connectors**:** connectors Bus2, Bus2B, Bus3A, Bus 3C have been added.

Modified components**:** In order to move from S to S**"**, many existent components have been changed. In order to identify behavioral differences, the study compared the component TCs. The modified components are listed below:

a) Port Artist: Ports selected by the planner components need to be highlighted in the Port Artist's GUI.

b) Vehicle: This component is queried by the Planner component to get information on available vehicles and it informs both vehicle artists components about any changes.

c) Vehicle Artist: Vehicles selected by the planner components need to be highlighted in the Vehicle Artist's GUI.

d) Warehouse: This component is queried by the Planner component to get information on warehouses capacity and it informs both vehicle artists components about any change.

e) Warehouse Artist: Warehouses selected by the planner components need to be highlighted in the Ware house Artist's GUI.

Modified connections: The connection between Bus2A and Bus1 has been replaced by the connections between Bus2A-Bus2 and Bus2-Bus1.

Since here the study investigation is the regression test selection problem (i.e., how to select ATC**"**, a subset of ATC relevant for testing S**"**), the focus on how components in S changed when moving to S**"**. The study utilizes a sort of "diff" algorithm which compares the behavioral models of both architectures and returns different between the two LTSs.

Step d: Select ATCs(Architecture- Level Test Case) from S that need to be retested in S":

Assuming S is the architecture under test, ATC is an architectural test suite for S regarding a testing criterion TC, S**"** a modified version of S, and ATC**"**is the new test suite for S**"**. ATC is included in ATC**"** if it traverses a path in the S ALTS which has been modified in the S**"** ALTS.

Here, the study report some interesting results by considering a few of the ATCs identified.

ATC #12 covers two different components (GB and CR) by exchanging three different messages (pressStart, Route, nothing Selected). Since both components were not modified in S**",** and since the path was not affected by other components' changes, the study guarantees that ATC #12 in

the ALTS traverses only unchanged nodes in ATS**"**. Thus, ATC #12 does not need to be reconsidered in S**"**.

ATC #26 covers six different components (GB, CR, VA, WA, PA, and V). Components VA, WA, PA and V have been modified when moving from S to S**"**, and thus should expect ATC #26 needs to be retested. However, when applying the architectural diff (ALTS and ALTS**"**), the study discovers ATC #26 traverses a non modified path. This happens since, even if some traversed components have been changed, the application of the Routing testing criterion to S**"** abstracts away differences between S and S**"**. Thus, ATC #26 does not need to be retested.

ATC #42 covers seven components (GB, CR, W, VA, WA, PA, V), the last five of which were modified when moving to S**"**. Although this case seems quite similar to ATC #26, when simulated in ALTS, ATC #42 covers nodes which have been modified in ALTS**"**. Thus, ATC #42 needs to be retested on S**"**.

To check the differences between ATS and ALTS**"**, the study used the LTSA "Animator" feature which allows paths simulation in an ALTS graph.

Steps e-f: Mapping ATCs" into code-level test cases TCs", and TCs" execution:

Five of the ATCs to be retested have been mapped into code-level test cases TCs**"**. The study reports just one of them, that is ATC #42 (Figure 1.3). Six TCs have been produced out of ATC #42. When retesting ATC #42 in the Cargo Router system, in fact, the study identified the following (genuine) code-level failure. When the process of routing an incoming cargo of n tons to a selected warehouse is concluded, the warehouse artist shows twice the quantity expected (i.e., it contains 2*n tons of the routed merchandize).

When comparing SA-based and traditional regression testing results, the present study helped to draw two important considerations:

i. The technique considered in the study shows something quite different from the safe regression test selection techniques in the literature. Although regression test selection technique shows that some test cases would need them to be retested, it happens that the differences between the two versions could make it infeasible to use the initial set of test cases to properly test code version two. The study approach, instead, while recognizing the need for retesting some ATCs, provides guidance for testing changed aspects by mapping ATCs into code-level test cases that properly version two.

ii. When an ATC is discarded (e.g., ATC#12 and ATC #26), the retest of all TCs related to ATC are avoided, thus reducing retesting effort.

IV. RESULTS AND DISCUSSION

A C2 style architectural specification has been used to model the topology of Cargo Router examples. This style has been chosen since it supports the C2 framework, which helps to make rigorous the mapping between SA test cases and code test case and simplifies test case execution. The results comparison help to conclude that the approach of SARTE can be applied to small-medium systems only. Further when moving from SA version 1 to version 2, the 86% of architectural test cases were not needed to be retested.

V. CONCLUSIONS

This research work has proposed an approach to handle the retesting of a software system during evolution of both its architecture and implementation, while reducing the testing effort. The case where the code evolved relative to unaffected software architecture and the case where the architecture evolved were applied to the case study Cargo Router Systems and results were collected. From a preliminary analysis, it can be concluded that bigger architectures concerning real systems may require a bigger computational time to apply the observational function, and a bigger number of architectural test cases can be produced according to the testing criterion.

VI. ACKNOWLEDGMENTS

I am indebted to Dr. G. Mahadevan (M.E., Ph.D) for his valuable insights and guidance.

VII. REFERENCES

[1]. 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).

[2]. I.Crnkovic and M. larsson, editors. Building Reliable Component-based Software Systems. Artech House, July 2002

[3]. 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 98.

[4]. Formal methods for Software Architectures. Tutorial book on Software and Formal Methods. In SFM-03:SA Lectures, Eds. M Bernardo and P. Inverardi, LNCS 2804, 2003

[5]. J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press/Addison-Wesley Publishing Co., 2000.

[6]. M. Dias and M. Vieira. Software Architecture Analysis based on Statechart Semantics.

[7]. FC2Tools.http://www-sop.inria.fr/meije/verification/quick-guide.html

[8]. The C2 Architectural Style. On-line at: http://www.ics.uci.edu/pub/arch/c2.html