

# Reasoning about Software Architecture-based Regression Testing through a Case Study

Henry Muccini  
Dipartimento di Informatica  
University of L'Aquila  
L'Aquila, Italy  
muccini@di.univaq.it

Marcio Dias  
Dept. of Computer Science and  
e-Science Research Institute  
University of Durham  
Durham, UK  
marcio.dias@dur.ac.uk

Debra J. Richardson  
Donald Bren School of Inf.  
and Computer Sciences  
University of California Irvine  
Irvine, CA, USA  
djr@ics.uci.edu

## Abstract

*Two main issues need to be covered when dealing with the dependability of component-based systems: quality assurance of reusable software components and quality assurance of the assembled component-based system. By focussing on the assembly, a software architecture specification of a component-based system allows to explicitly model the structure and required system behavior by specifying how components and connectors are intended to interact. Software architecture-based conformance testing techniques can yield confidence on the implementation conformance to expected structural and behavioral properties as specified in the architectural models.*

*In this paper we explore software architecture-based regression testing methods that enable reuse of earlier saved results to test if a different assembly of components conforms to the evolved software architecture. The approach is presented through a running example.*

## 1. Introduction

Roughly speaking, a component-based software system is an assembly of reusable components, designed to meet the quality attributes identified during the architecting phase [7]. 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.

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 the verification stage to be properly performed required the as-

sembly 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 [9] 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. In a component-based context, SA provides an high-level blueprint on how components are supposed to behave when integrated in a certain system.

Some techniques have been proposed so far to extract (architecture-level) test cases from architectural specifications [8, 19, 3, 11, 13, 15]. In particular, in previous papers we have shown how 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 testing [8, 15, 16]. By applying such techniques to component-based system, we may then acquire a certain confidence on the assembly fulfillment of architectural properties.

The main drawback of such SA-based testing techniques is that whenever the SA model or the system implementation change, the entire test selection and execution process needs to be re-executed from scratch. This strongly limits the applicability of architectural testing techniques in practice, since it requires to repeat the entire testing process at any time a new component-based system is produced out of a set of components. Goal of this paper is to introduce SARTE (Software Architecture-based Regression TESTING), a project which focuses on providing a framework and an approach for SA-based testing. We show how SA-based regression testing provides a key solution to the problem of retesting an SA when moving from a system assembly to

another.

In particular, after identifying SA-level behavioral test cases and testing conformance of the code with respect to the expected architectural behaviors [15], 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 the approach through a case study where the SA is specified following the C2 architectural style [6].

In Section 2, we identify the project’s primary goals. Section 3 describes a case study which will be used as a running example in the following of the paper. Basic background on regression testing is provided in Section 4. A description of the approach on the running example is provided in Section 5. Section 6 concludes the paper, presenting directions for future work.

## 2. Project Goals

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:

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

- Context: Given a component-based 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 component-based system ( $P'$ ) is implemented - where some components from  $P$  remain, and some components are modified (for example, by adding/removing internal objects or interfaces).
- 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 component-based implementation  $P''$  may have been also developed.
- Goal: Test the conformance of  $P''$  with respect to  $S''$ , while reusing previous test information for selective

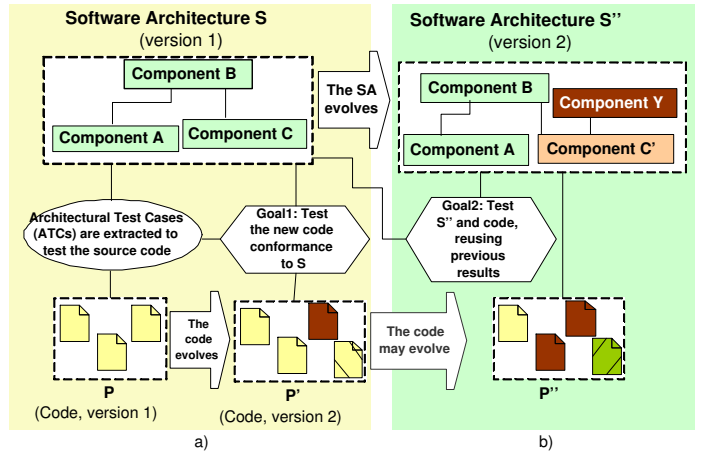


Figure 1. Project goals: a) the component-based system implementation evolves; b) the software architecture evolves

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 testing techniques with SA-based regression testing and by exploiting similarities and differences between SA versions.

## 3. The Cargo Router system example

The *Cargo Router* system [18] 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 2 shows two different architectural specifications of the Cargo Router system. In the remainder of this case study, we assume that the architectural specification is written in accordance with the C2 style rules [6]<sup>1</sup>.

Figure 2.a realizes the above mentioned features through the following components: Port ( $P$ ), Vehicle ( $V$ ),

<sup>1</sup> C2 communication rules require that all communication be achieved via message passing through connectors. A component within a given architecture is only aware of services provided by components “above” it in the hierarchy, and is completely unaware of services provided by components “beneath” it.

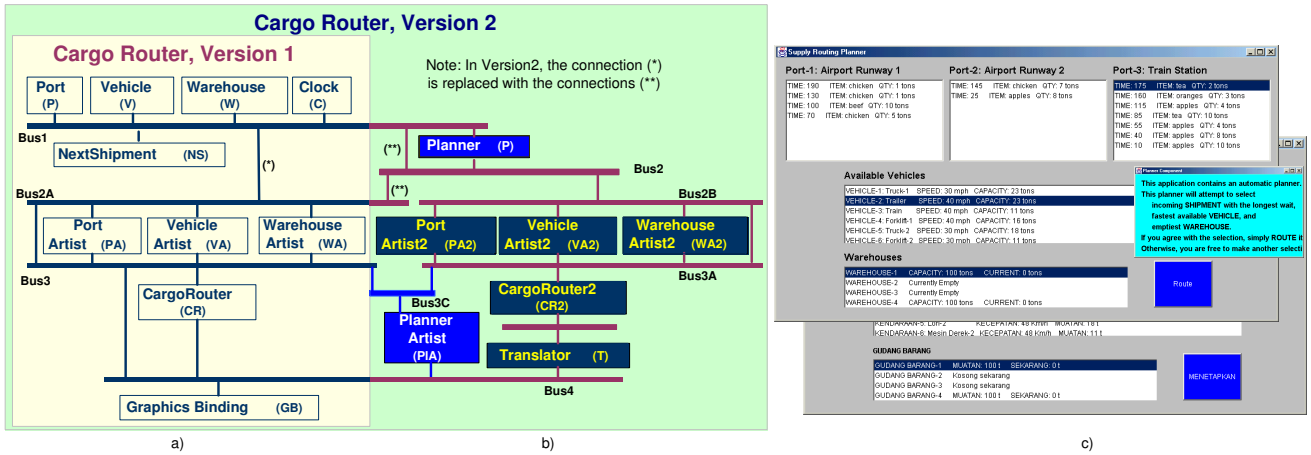


Figure 2. The Cargo Router system. a) SA version 1 (S); b) SA version 2 (S''); c) GUI

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 CargoRouter (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 NextShipment (NS) component regulates the incoming of new cargo on a selected port. The Clock (C) sends ticks to the system.

Figure 2.a+b shows an evolution of the initial architecture (Cargo Router, v2); it realizes a bilingual 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 (PIA) components), which automatically selects the incoming shipment with the longest wait, fastest available vehicle and emptiest warehouse.

Figure 2.c illustrates the bilingual 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 window informs an automatic planner is in place.

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

toring software architectures.

#### 4. Background on Regression Testing

Here, we briefly introduce how traditional code-based regression testing techniques work, in order to identify the basis for an SA-based regression testing approach. We focus on selective regression testing techniques and then reuse the same logical steps to propose a SA-based regression testing approach in Section 5.

Regression testing, as quoted from [12], “attempts to validate modified software and ensure that no new errors are introduced into previously tested code”.

The traditional regression testing 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 testing technique, called *retest all*, T' contains all the test cases in T, and P' is run on T'. In **regression-test-selection** (RTS) techniques, T' is selected as a “relevant” subset of T, i.e., assuming that  $t1 \in T$ , t1 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” RTS definition). An empirical study and analysis on different RTS techniques has been proposed in [10], together with the identification of the different activities required. Specification-based regression testing techniques, which are also guiding our research, have been discussed in [17].

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 an RTS technique, but at the SA-level rather than at the code-level. How new test cases

should be selected for new functionality will be addressed in future work.

## 5. SA-based Regression Testing

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

*SA-based testing.* In particular, we apply a SA-based conformance testing approach.

*SA-based regression test selection.* This phase is split into two sub-phases, in order to meet *Goal 1* and *Goal 2* identified in Section 2.

Figure 3 summarizes the activities required by SA-based conformance and regression testing: while in [17] we provided a step-by-step (theoretical) description of our approach, we here describe it through its application to the Cargo Router running example.

Section 5.1 briefly describes how the SA-based testing phase has been implemented. Sections 5.2 and 5.3 describe how to retest a modified implementation of the initial SA (Goal 1) and a modified SA (Goal 2), respectively.

### 5.1. SA-based Testing applied to the Case Study

Following the five steps depicted in Figure 3.a, in *Step 0*) we specified the Cargo Router topology using the C2 style architecture through the Argus-I tool [1]. The system behavior has been modeled by Labeled Transition Systems (LTSs) (one for each component), specified through the Finite State Process algebra (FSP) [14] and drawn by the LTSA tool [14]. The Cargo Router v1 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*) we have defined a testing criterion to focus on “all those behaviors generated by routing events” (hereafter called, *Routing Criterion*). By focusing on the *Routing Criterion*, we identified a more selective/abstract LTS (called ALTS), composed by 80 states and 244 transitions. From this ALTS, we identified 164 architecture-level test cases (ATCs) using McCabe’s path coverage criterion [20]. Figure 4 shows one of the ATCs previously identified. To map SA-level ATCs to code-level test cases we made use of the C2 framework which dictates how architectural components are implemented by Java components. The mapping between architectural test cases and code-level test cases is systematic and tool supported, as analyzed in [16].

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

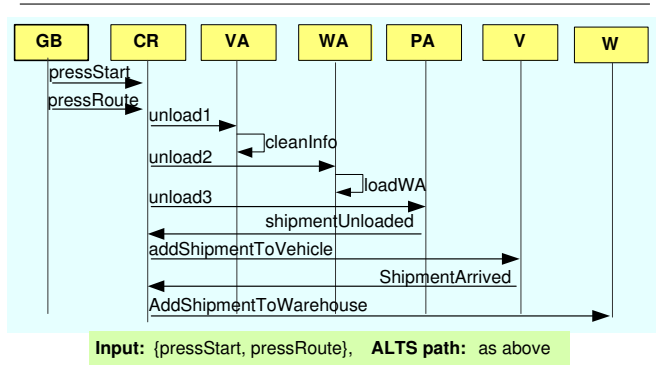


Figure 4. Architectural Test Case ATC #42

code-level test cases [16]. The ATC in Figure 4, for example, has been mapped to six different code-level test cases.

We finally used the Argus-I tool monitoring and debugging capabilities [1] to make a *deterministic* analysis [5] of the code and observe the desired sequence. At the end of this analysis, we identified no architectural errors at the code level. Further details on this phase are out of the scope of this paper and may be found in our previous work [15, 16].

### 5.2. Goal 1: Test Conformance of a Modified Implementation $P'$ to the initial SA

In the previous phase, the SA-based conformance 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.a), we want to test the conformance of the new implementation  $P'$  to the initial architecture.

Following the four steps depicted in Figure 3.b, we took into consideration two different implementations of the Cargo Router system:  $P_1'$  which modifies the use of the “random function” in class *Vehicle.java* to select (at startup) vehicles available for shipments, and  $P_2'$  which 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  $P_2'$ .

In order to regression test such implementations, we applied the concepts reported in the JDiff [2] algorithm by hand<sup>2</sup>: we built a graph representation of  $P$ ,  $P_1'$ , and  $P_2'$  (*Step A*) and we compared two pairs of implementations:  $(P, P_1')$  and  $(P, P_2')$  (*Step B*). We were able to discover four lines changed between  $(P, P_1')$ , all local to a single method, while we discovered 25 changes in moving from  $P$  to  $P_2'$ , changes that involved two different components and four different methods.

<sup>2</sup> at the time of this paper, the JDiff tool was not yet available

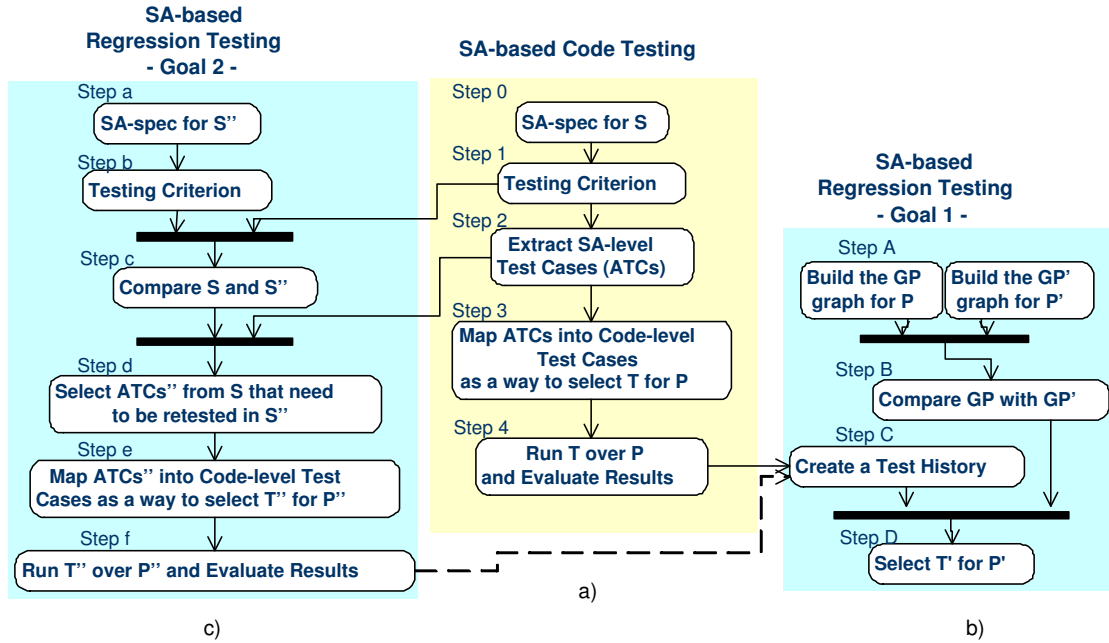


Figure 3. Activity Diagram of our SA-based Regression Testing approach

We then manually instrumented those  $P$ 's methods subject to change in  $P_1'$  and  $P_2'$ . The instrumentation simply prints a message to mark the changed method/lines as traversed (*Step C*). We ran  $P$  over a subset of the code test cases  $T$  previously selected.

When  $P$  is run over  $T$ , we discovered that the changed method in  $P_1'$  is never traversed. This means that all such test cases do not have to be rerun on  $P_1'$ . Even if not necessary, we re-ran some of them, without identifying any conformance errors. We also discovered that eight of the 18 code-level test cases we ran did not cover any changed method in  $P_2'$ . We thus retested only ten of the 18 test cases. When retesting such test cases, we identified all of the injected faults. To conclude the experiment, we also tried to retest the eight discarded test case. None of them revealed any architectural error.

### 5.3. Goal 2: Test Conformance of an Evolved Software Architecture

In Section 5.1 the SA-based conformance testing approach has demonstrated that  $P$  conforms to its SA. After evolving the architecture  $S$  into  $S''$  (Figure 1.b), we want to check the component-based system implementation's conformance to the new architecture.

The approach we take 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 3.c, we here describe the differ-

ent steps in Goal 2.

#### Steps a-b: $S''$ specification and Testing Criterion

The Cargo Router v2 specification consists of 305 lines of FSP statements and the resulting global LTS'' is composed by 360,445 states and 869,567 transitions. By focusing on the *Routing* testing criterion (identified in Section 5.1), we produced an ALTS composed by 8,448 states and 55,200 transitions.

#### Step c: Comparing $S$ with $S''$

In [18] the authors state that a software architecture changes when a new component/connector is added, removed, replaced or the architecture is reconfigured. In our context, both C2 structural and FSP behavioral specifications are used to compare architectures. When moving from  $S$  to  $S''$  in Figure 2, we have the following differences:

**Architecture Reconfiguration:** another instance of the artists components (PA2, VA2, WA2) and of the cargo router (CR2) have been added to produce the bilingual graphical user interface (GUI);

**Added components:** the Translator component has been added to translate contents from English to Indonesian. The Planner and Planner Artist components have been added to allow the automatic routing feature;

**Added connectors:** connectors Bus2, Bus2B, Bus3A,

Bus3C 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, we compared the component LTSs. The modified components are listed below:

- PortArtist: ports selected by the planner components need to be highlighted in the PortArtist’s GUI;
- 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;
- VehicleArtist: vehicles selected by the planner components need to be highlighted in the VehicleArtist’s GUI;
- 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;
- WarehouseArtist: warehouses selected by the planner components need to be highlighted in the WarehouseArtist’s GUI.

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

Since here we are investigating the regression test selection problem (i.e., how to select  $ATC''$ , a subset of  $ATC$  relevant for testing  $S''$ ), we focus on how components in  $S$  changed when moving to  $S''$ . We utilize a sort of “diff” algorithm which compares the behavioral models of both architectures and returns differences between the two LTSs.

#### Step d: Select ATCs 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  $S''$ . *atc1*  $\in$   $ATC$  is included in  $ATC''$  if it traverses a path in the  $S$  ALTS which has been modified in the  $S''$  ALTS.

Here, we report some interesting results by considering a few of the  $ATCs$  identified in Section 5.1.

$ATC$  #12 covers two different components (GB and CR) by exchanging three different messages (pressStart, Route, nothingSelected). Since both components were not modified in  $S''$ , we are guaranteed that  $ATC$  #12 in the ALTS traverses only unchanged nodes in  $ALTS''$ . 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  $S''$ , thus we should expect  $ATC$  #26 needs to be retested. However, when applying  $\text{diff}(ALTS, ALTS'')$  we discover  $ATC$  #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 differ-

ences 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 ALTS and  $ALTS''$ , we 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  $TCs''$ . We here report just one of them, that is  $ATC$  #42 (Figure 4). Six  $TCs''$  have been produced out of  $ATC$  #42. When retesting  $ATC$  #42 in the Cargo Router bilingual system, in fact, we 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  $2 \times n$  tons of the routed merchandise).

When comparing SA-based and traditional regression testing results, we may draw two important considerations: *i*) our technique 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 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. Our 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 could properly test code version two. *ii*) When an  $ATC$  is discarded (e.g.,  $ATC$  #12 and  $ATC$  #26), the retest of all such  $TCs$  related to  $ATC$  are avoided, thus reducing retesting effort.

## 6. 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 regression testing process. This high-level analysis has been refined for C2 style architectures. We examined the case where the code evolved relative to an 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 provide a more rigorous empirical *evaluation* of the proposed approach, to address how *new test cases* should be selected when new functionalities are added into the system, and 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 [4] to SA-based regression testing.

## Acknowledgments

We are indebted to Nenad Medvidovic, who provided the Cargo Router system, Alessandro Orso, who helped us running JDiff on the system, and to Tim Standish, Leila Naslavsky, Lihua Xu, and Yuewei Zhou, of the ROSATEA research group, for their valuable insights.

## References

- [1] The Argus-I project. University of California, Irvine. Information on-line at <<http://www.ics.uci.edu/~mdias/research/ArgusI>>.
- [2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In IEEE Proc. 19th Automated Software Engineering Conference, ASE 2004.
- [3] A. Bertolino and P. Inverardi. Architecture-based Software Testing. In *Proc. ISAW96*, October 1996.
- [4] 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.
- [5] R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Trans. on Software Engineering*, Vol. 24, N. 6, pp. 471-490, June 1998.
- [6] The C2 Architectural Style On-line at: <<http://www.ics.uci.edu/pub/arch/c2.html>>.
- [7] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-based Software Systems*. Artech House, July 2002.
- [8] 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.
- [9] D. Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In *Formal Methods for Software Architectures*, pages 1–24. Lecture Note in Computer Science, 2804, 2003.
- [10] 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.
- [11] 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.
- [12] M. J. Harrold. Testing: A Roadmap. In A. Finkelstein (Ed.), *ACM ICSE 2000, The Future of Software Engineering*, pp. 61-72, 2000.
- [13] Z. Jin and J. Offutt. Deriving Tests From Software Architectures. In the *Twelfth IEEE Int. Symp. On Software Reliability Engineering*, (ISSRE'01). Hong Kong, November 2001.
- [14] J. Magee, and J. Kramer. *Concurrency: State models & java programs*. Wiley publisher, April 1999.
- [15] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. In *IEEE Transactions on Software Engineering*. Vol. 30, Issue N. 3, March 2004, pp. 160-171.
- [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] H. Muccini, M. Dias, D. Richardson. Towards Software Architecture-based Regression Testing. Internal Report, University of L'Aquila. Year 2005. Available at [www.HenryMuccini.com/research/2005/SARTE\\_TR.htm](http://www.HenryMuccini.com/research/2005/SARTE_TR.htm).
- [18] 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.
- [19] D. J. Richardson and A. L. Wolf. Software Testing at the Architectural Level. *ISAW-2* in Joint Proc. of the *ACM SIGSOFT '96 Workshops*, pp. 68-71, 1996.
- [20] A. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. NIST Special Publication 500-235, August 1996.