

Software Architecture-based Regression Testing

Henry Muccini ^a, Marcio Dias ^b, Debra J. Richardson ^c

^a*Dipartimento di Informatica,
University of L'Aquila, L'Aquila, Italy, email: muccini@di.univaq.it*

^b*Dept. of Computer Science and e-Science Research Institute,
University of Durham, Durham, UK, email: marcio.dias@dur.ac.uk*

^c*Dept. of Informatics, Donald Bren School of Information and Computer Sciences,
University of California Irvine, USA, email: djr@ics.uci.edu*

Abstract

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, behaviors.

This work explores 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 “low-level” and “high-level” evolution, i.e., whether a slightly modified implementation conforms to the initial architecture, and whether the implementation continues to conform to 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.

Key words:

Software Architecture, Dependable Systems, Regression Testing, Architecture-based Analysis and Testing. ¹

¹ *Main acronyms: SA = Software Architecture, RT = Regression Testing, TS = Transition System, ATS = Abstract TS, ATC = Architecture-level Test Case, GP = program Graph.*

1 Introduction

A Software Architecture (SA) [16] 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. 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 [6]. Additionally, SA-based testing methods are available to check conformance of the implementation’s behavior with SA-level specifications of expected behavior [13] and to guide integration and conformance testing [7,28].

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” [34]. 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 rerun 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 the three authors 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 [28], 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 [13].

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 an 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 [28], 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.

1.1 Motivations and Goals

This section describes why SA-based RT can contribute to improve the overall system dependability (Section 1.1.1) and our ongoing project goals (Section 1.1.2).

1.1.1 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 [19]. Although SA-based RT may be used for both purposes, we here focus on the maintenance aspect, being confident this approach may be used during development as well.

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

Why may 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 [16]. Much research has investigated SA evolution, especially at runtime. In [32], for example, the authors analyze 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 [4] the authors describe an approach to specify architectures that permits the representation and analysis of dynamic architectures. In [24] the authors analyze the issues of dynamic changes to a software configuration, in terms of component creation and deletion, and connection and disconnection. In [27] the authors analyze such Architecture Description Languages which provide specific features for modeling dynamic changes.

Why SA-based Regression Testing?

Many functional and non-functional analysis techniques have been proposed to op-

erate at the SA-level [6]. However, the drawback is that (given that an 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 SA-level test cases and their execution at the code level is a long and expensive process (as described in [28] and summarized in Section 3.1). Reusing previous results as much as possible may strongly reduce testing effort while testing a modified architecture. Quoting [18], 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 can identify (functional) errors that are likely to be missed when applying traditional code-level testing, thus complementing traditional testing techniques. In line with research on specification-based RT (see Section 6), SA-based RT may valuably complement code-based RT, as discussed later in Section 5.

1.1.2 SARTE 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 software system, an architecture 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, an architecture 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 been also developed.
- Goal: Test the conformance of P'' with respect to S'' , while reusing previous

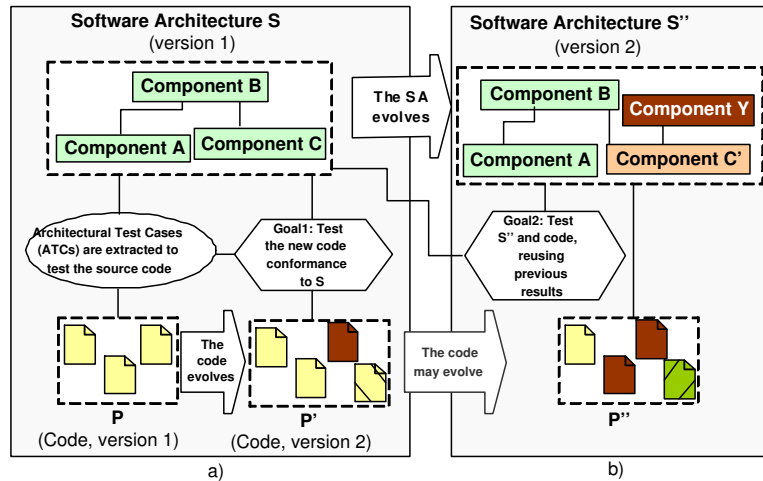


Fig. 1. Project goals: a) the implementation evolves; b) the software architecture evolves

test information for selective RT, 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 RT techniques with SA-based RT and by exploiting similarities and differences between SA 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. Ongoing research on this topic is presented in Section 7.

1.2 Paper Structure and Organization

The paper is organized as follows. Basic background on regression testing is provided in Section 2. A theoretical description of the approach is presented in Section 3. In Section 4 we illustrate the application of SARTE to two different applications: the Elevator example and the Cargo Router system. Section 5 provides a description of what we learned from our experience on SA-based regression testing together with some considerations. Related work are briefly discussed in Section 6. The paper concludes with a discussion on ongoing and future work (Section 7) and some concluding remarks (Section 8).

2 Regression Testing

In this section, we focus on the regression testing strategy to provide the background necessary to understand our approach for SA-based regression testing. We briefly introduce how regression testing works, describing, in broad terms, how to identify appropriate tests in a regression test selection context.

Regression testing, as quoted from [19], “attempts to validate modified software and ensure that no new errors are introduced into previously tested code”. 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' .

To explain how a regression testing technique works in general, let us assume that a program P has been tested with respect to a test set T . When a new version P' is released, regression testing techniques provide a certain 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 *selective regression testing*, T' is selected as a “relevant” subset of T , where $t \in T$ is relevant for P' if there is the potential that it could produce different results on P' that it did on P (following a *safe* definition).

In general terms and assuming that P is a program under test, T a test suite for P , P' a modified version of P and T' the new test suite for P' , regression testing techniques work according to the following steps: 1) select T' , subset of T and relevant for P' ; 2) test P' with respect to T' ; 3) if necessary, create T'' , to test new functionality/structure in P' ; 4) test P' with respect to T'' ; 5) create T''' , a new test suite and test history.

All of these steps are important for the success of a selective regression testing technique and each of them involves important problems [17]. However, step 1 (also called, regression test selection) characterizes a selective regression testing technique. For this reason, we focus on this step to propose an SA-based regression testing approach in the rest of this paper.

3 SARTE: SA-based Regression Testing

Our SA-based regression testing inherits the two-phased decomposition from traditional RT approaches, and comprises the following two phases:

SA-based conformance testing. We apply a SA-based conformance testing approach.

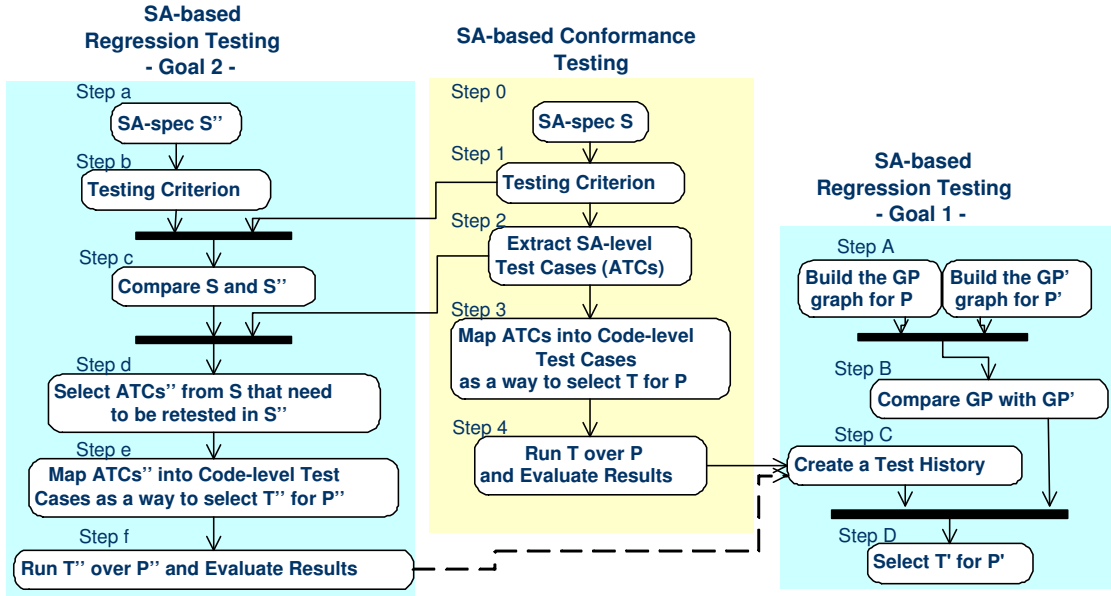


Fig. 2. Activity Diagram of our SA-based Regression Testing approach

SA-based regression test selection. This phase is decomposed to meet *Goal 1* and *Goal 2* in Section 1.1.2.

Figure 2 summarizes the activities required by SA-based conformance and regression testing. While SA-based conformance testing has been already analyzed in [28,29], goal of this paper is to focus on SA-based regression testing.

3.1 SA-based Conformance Testing

As mentioned before, this work builds upon the general framework for SA-based conformance testing set forth in [28], whose goal is to test the implementation conformance to a given software architecture. The software architecture specification is used as a reference model to generate test cases while its behavioral model, which describes the expected behavior, serves as a test oracle.

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

Step 0: SA specification. It begins with a topological and behavioral specification of the SA. The topology describes the SA structure in terms of components, connectors, and configuration. The behavioral model specifies how the SA is supposed to behave. Architecture description languages are employed for topological description, while transition systems (TS) are hereafter employed for SA behavior.

Step 1: Testing criterion. An observation function is introduced in order to implement a *testing criterion* that looks at the SA from a perspective that is deemed to

be relevant for testing purposes, while hiding away non-relevant actions from this perspective. The state machine-based model is abstracted, producing an Abstract TS (ATS), in order to show such high-level behaviors/components we want to test.

Step 2: Architecture-level test case. 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 ATS.

Step 3: Test cases. Naturally, such ATCs strongly differ from executable code-level test cases, due to the abstraction gap between software architecture and code (the *traceability problem* [14]). We deal with this problem through a “mapping” function which maps SA-level functional tests into code-level test cases.

Step 4: Test execution. 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, using the architectural behavioral model as a test oracle to identify when a test case fails or succeeds.

Experience on applying SA-based conformance testing has demonstrated its feasibility and suitability. However, repeating the entire testing process at any time the system evolves is undoubtedly too expensive, thus making SA-based testing less appealing and applicable. Here we propose an approach to deal with system evolution, which reuses previous test results to retest the modified architecture/implementation with reduced effort.

3.2 Goal 1: Test Conformance of a Modified Implementation 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), and assuming the same architectural decisions want to be implemented, 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: Generating P graph (GP). Most common approaches to code regression testing is to structurally represent P using a graph. Depending on the selective regression testing technique, this graph can be a control flow graph, a control depen-

dence graph, a program dependence graph and so on. After modification, P' is also represented as a graph.

Step B: Comparing GP with GP'. As in traditional code-based regression testing, the code graphs for P and P' are compared, to identify how code changes are reflected in the graph. Notice we are not restricted to any specific graph representation or graph comparison technique. As we will show later in Section 4, in our experimental studies, we used different technologies to this end.

Step C: Recording Coverage. P is executed over each $t \in T$. (We remind that a test suite T for P has been already identified during the SA-based conformance testing phase – Step 3 in Figure 2). During execution, information on how the graph is traversed is recorded, usually produced by light code instrumentation. For each test case t , the Test History keeps track of the nodes and arcs in GP traversed while executing P over t .

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 $t \in T$ might cover a node which is modified in P' , then t needs to be rerun on P' .

Once T' is selected, $t' \in T'$ is run over P' and results are collected and compared with an oracle to decide if the test failed or succeeded.

While those steps are those traditionally considered for regression testing, one of the main differences with respect to traditional code-based RT techniques is that *the oracle* in SA-based RT is *the software architecture specification itself* (as happens in specification-based regression testing). 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 (initial results on this topic will be presented in Section 7).

3.3 Goal 2: Test Conformance of an Evolved Software Architecture

Let us assume again that SA-based conformance testing has demonstrated that 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.

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 in terms of structural and behavioral models. A graph representing how the SA behaves is obtained by composing the TS models of component behavior, following the structural organization of those components in the architecture.

Step b: Testing Criterion. The testing criterion (previously identified in the conformance testing phase, step 1, and applied to S) is applied to S'' .

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

Topological changes are identified by comparing the two versions of the structural specification provided in some architectural language. Added or deleted components or modified configurations can be revealed by this analysis. However, structural comparison is certainly not enough, since the same topological description can lead to differently behaving architectures. Behavioral changes are identified based on differences in ATS nodes and edges.

An *SA diff* algorithm is used to compare the two behavioral specifications S and S'' and differences are identified.

Notice that the “comparison” step is executed over the ATSs (instead directly to the TS specifications) since we are interested to test the two specifications with respect to the same testing criterion.

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

Step e: Identify Code-Level Test Cases. Once we have identified the ATCs which need to be regression tested in S'' , we 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.

In order to gain some practical experience on the topic, in the following we specialize and refine some of the activities in Figure 2 to some examples.

4 Application of SARTE to Case Studies

In this section, we present our software architecture-based regression testing approach being applied into two different case studies: the Elevator example (Section 4.1) and the Cargo Router system (Section 4.2). For sake of brevity, we discuss only the application of Goal 1 to the Elevator case study and of Goal 2 to the Cargo Router system, and then we summarize the other results.

Since how to perform the SA-based conformance testing phase described in Section 3.1 has been already discussed elsewhere [28], we limit the discussion on how to identify SA-based test cases, while leaving more space for Goal 1 and Goal 2.

An evaluation of the overall results is presented in Section 4.3.

4.1 Elevator Example

Elevator systems have been widely used in testing and analysis literature because of two main reasons: *i*) everyone is familiar with elevator systems, and can easily understand the requirements for such application domain, and *ii*) these systems contain stateful components and timing requirements, which give them a level of complexity that is interesting for verification purposes.

In this section, we illustrate the application of our technique to an Elevator system whose architectural specification follows the C2 architectural style [2]². We have previously performed the SA-based conformance testing technique over this system, as reported in [29]. Now, going through this example, we illustrate how Goals 1 and 2 are accomplished in our SA-based regression testing technique.

We use two different configurations for the elevator system. Our first configuration (representing S) contains the building panel (which includes all the panels from different floors of the building) and one elevator car (Figure 3.b). Our second configuration (representing S'') contains the building panel, two elevator cars, and a scheduler component which assigns calls requested through the building to the closest elevator car (Figure 3.c). In detail, the components for these elevator systems are:

² The C2 style imposes some composition and behavioral rules enabling some level of independence between the components used to describe the SA. Communication may happen only through connectors. Communication is message based.

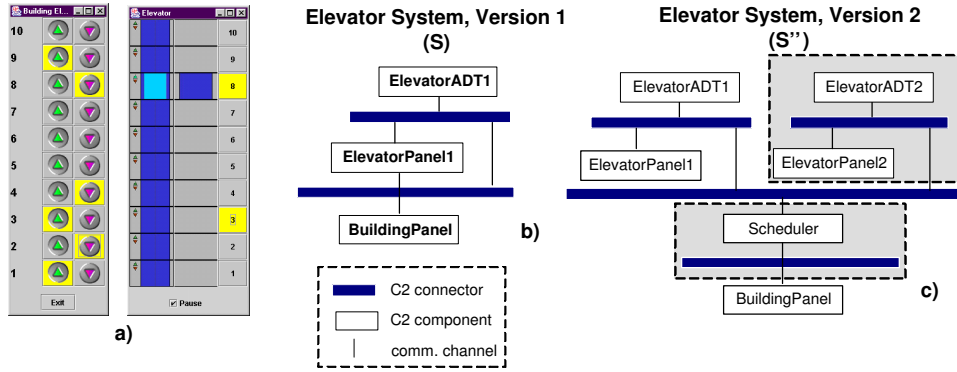


Fig. 3. The Elevator systems: a) GUI, b) architecture of the first elevator configuration (S), c) architecture of the second system configuration (S'')

- **BuildingPanel**: this component represents all the elevator call panels of the building. Through this component, users in different floors can request a call to the elevator system, indicating the desired direction.
- **Scheduler**: this component receives call requests from the BuildingPanel, and selects which elevator should attend such call. In our case study we are using a scheduling policy so that if a call is made at time “t”, it selects the elevator car that, at time “t”, could attend it with the lower waiting time required.
- **ElevatorPanel**: this component represents the internal panel of an elevator car. After entering the elevator, the passenger can request calls through it and see the current floor.
- **ElevatorADT**: this component maintains the information about the elevator car state, and a list of all the calls it needs to attend.

When applying SA-based conformance testing over the Elevator system, we modeled components transition systems using statecharts, with the semantics described in [12]. Each component has its behavior described with statecharts, and the architecture configuration combines those statecharts in order to produce a global architectural behavioral model [12]. From the global model, we applied a testing criterion in order to select as “relevant” all, and only, such messages necessary to send a call (the *addCall*) and to receive the *callAttended* message (which means that the call has been handled), while hiding the others. Twenty four ATCs have been selected corresponding to the testing criterion. Two of them are analyzed in this paper, and shown in Figure 4.

In ATC #5, the test scenario is given by the elevator (ADT1) attending a call placed in the building panel (BP). In other words, BP sends an *addCall* and ADT1 attends the call and replies with a *callAttended* message. In ATC #12, the test scenario is given by the elevator attending a call placed in its internal panel. From those two ATCs, 18 (code-level) test cases have been generated (shown in Figure 5) by applying the test selection technique in [29].

The code level test cases have been run on the system implementation and no errors

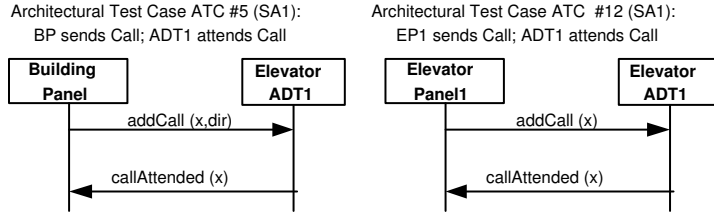


Fig. 4. Elevator system: Architectural Test Cases ATC #5 and ATC #12 for S

Path #1 – Test Cases

- TC1: BP.addCall (1,up); ADT1(1,up)
- TC2: BP.addCall (1,up); ADT1(2,up)
- TC3: BP.addCall (1,up); ADT1(2,down)
- TC4: BP.addCall (2,up); ADT1(1,up)
- TC5: BP.addCall (2,up); ADT1(2,down)
- TC6: BP.addCall (2,up); ADT1(3,down)
- TC7: BP.addCall (2,down); ADT1(1,up)
- TC8: BP.addCall (2,down); ADT1(2,up)
- TC9: BP.addCall (2,down); ADT1(3,up)
- TC10: BP.addCall (3,down); ADT1(4,down)
- TC11: BP.addCall (3,down); ADT1(3,down)
- TC12: BP.addCall (3,down); ADT1(2,down)

Path #2 – Test Cases

- TC1: EP1.addCall (1); ADT1(1,up)
- TC2: EP1.addCall (1); ADT1(2,up)
- TC3: EP1.addCall (1); ADT1(2,down)
- TC4: EP1.addCall (2); ADT1(1,up)
- TC5: EP1.addCall (2); ADT1(2,down)
- TC6: EP1.addCall (3); ADT1(2,down)

Fig. 5. Elevator system: Test cases derived from ATC #5 and ATC #12

have been revealed. More details on the Elevator architectural specification and test case selection can be found in [29] (even if there we used a different notation to specify the SA behavior).

4.1.1 Goal 1: Test Conformance of P' to the Elevator architecture, version 2

Taking into consideration the Elevator system architecture, version 2 (showed in Figure 3.c) and by submitting its implementation (version ElevatorArch5) to an informal, runtime analysis, we realized the algorithm used to assign calls to the elevators was not optimal. The initial algorithm assigns a call to elevator X, when at time “t” such elevator can attend the call with the lower waiting time. Unfortunately, if at time “t+1” other calls (internal or external) are made, we are not anymore guaranteed on the optimality. We thus decided to change the algorithm, and the code itself, in order to provide a more flexible (runtime) way to assign calls.

ElevatorArch5B version has been produced for this goal. Moreover, we produced a third version ElevatorArch5C, injecting a fault in the ElevatorArch5B version. In the following, we will refer to those versions as v5, v5B and v5C.

In order to regression test such implementations, we used the algorithm described in [33]: given a Java implementation of the system under test, the algorithm performs two phases, partitioning and selection. During the partitioning phase, a high-level graph representation of P and P' is provided and a quick analysis is performed. During selection, instead, the algorithm performs edge-level test selection, refining previous results. DejaVOO safely implements the regression test selection technique in [33], when some assumptions are verified, such as reflection is not applied and test can be run deterministically. Unfortunately, both assumptions do not hold in a C2-based software implementation. Thus, we had to make a partial use of De-

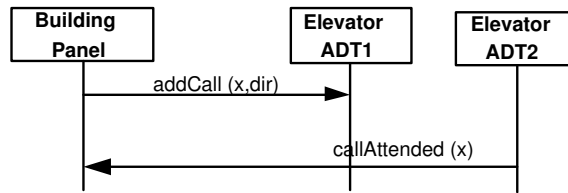


Fig. 6. Elevator system: Implementation scenario not conforming to the architectural specification

jaVOO: we used it to compare (v5, v5B) and (v5B, v5C), and to generate the JIGs³ for both versions of the code and the list of dangerous edges. Analyzing the dangerous edges document, we discovered that 8 edges have been changed going from v5 to v5B while 1 edge has been changed going from v5B to v5C. The document produced by DejaVOO clearly identified the code portions (classes, methods and lines of code) subject to change.

We thus manually instrumented those Elevator v5 methods subject to change in v5B and that Elevator v5B method changed in v5C. The instrumentation simply printed a message to mark the methods as traversed. When running the 18 test cases over v5, three dangerous edges (i.e., in the fireEntry, handle and notifCallAttended methods) are always traversed. This unfortunately means that all such test cases have to be rerun on P'. When running the 18 test cases over P' our pragmatic approach here has been to make a *deterministic* [11] analysis of the code to observe the desired sequence. The deterministic approach forces a program to execute a specified test sequence by instrumenting it with synchronization constructs that deterministically reproduce the desired sequence. This determinist analysis is performed through monitoring and debugging capabilities provided by Argus-I [13], by setting break-points during code execution. We force the system to be in a state described by the architectural test case, then we try to deterministically force the execution of one of the ATS paths compliant with the test frame.

When running the 18 test cases on P', the deterministic execution of three different test cases over P' returned an architectural fault: while the elevator call is handled by the ADT1 component, the CallAttended message is generated by the ADT2 component (see Figure 6). This kind of behavior is never modeled as part of the SA specification, and results into an unexpected behavior. Thus, the changes made from version ElevatorArch5 to ElevatorArch5B have affected the implementation conformance the initial architecture.

When running the 18 test cases over v5B, instead, only 9 test cases traversed dangerous edges (thus, had to be re-executed). Moreover, the injected error has been detected by three test cases.

³ A JIG is the Java Interclass Graph, used to extend a traditional control flow graph, in which nodes represent program statements and edges represent the flow of control between statements.

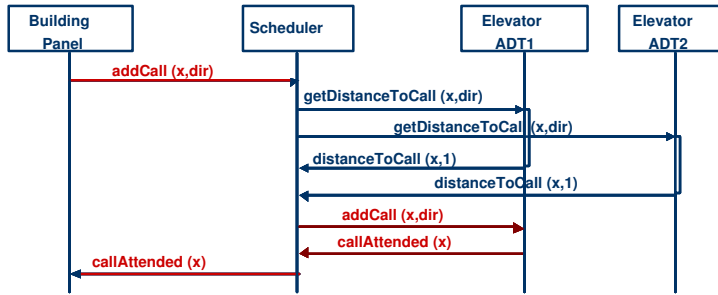


Fig. 7. Elevator system: Path covered when executing ATC #5 on the Elevator system version 2

4.1.2 Goal 2: Test Conformance of Evolved Elevator Architecture (overview of results)

When comparing the two architectural specifications, the following differences have been identified: *i*) an additional layer has been added between BuildingPanel and the main elevator connector, including a new connector and the Scheduler component, *ii*) besides the Scheduler, elevator components (ElevatorADT2, Elevator-Panel2) were added. The *SADiff* algorithm has been used to compare the behavioral models of both architectures to return differences between the two statecharts. This algorithm compares the XML representation of both graphs, and returns differences among them. From this analysis we discovered that path traversed by ATC #5 (see Figure 4) has been affected by changes (although both BP and ADT1 components have not been subject to changes) while ATC #12 has not. As illustrated in Figure 7, when submitting the Elevator version 2 architecture to the *addCall(x,dir)* input, a completely different path is traversed in the Elevator version 2 transition system.

When mapping ATC #5 into code-level test cases for S'' , we now obtain 36 test cases for ATC #5 in S'' . The execution of such test cases does not reveal any conformance fault.

4.2 Cargo Router System

The *Cargo Router* system [32] 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 8 shows two different architectural specifications of the Cargo Router sys-

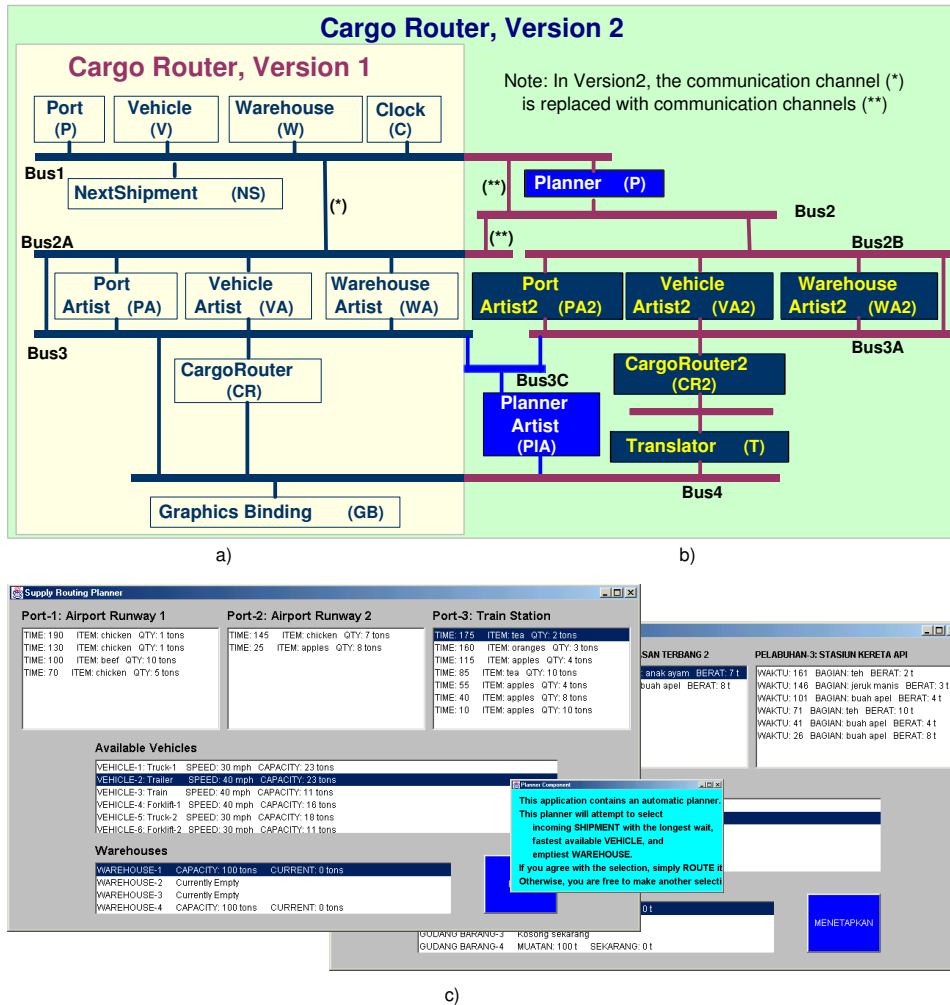


Fig. 8. The Cargo Router system: a) SA version 1 (S); b) SA version 2 (S''); c) GUI

tem. In the remainder of this case study, we assume that the architectural specification is written in accordance with the C2 style rules.

Figure 8.a 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 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 8.a+b shows an evolution of the initial architecture (Cargo Router, ver-

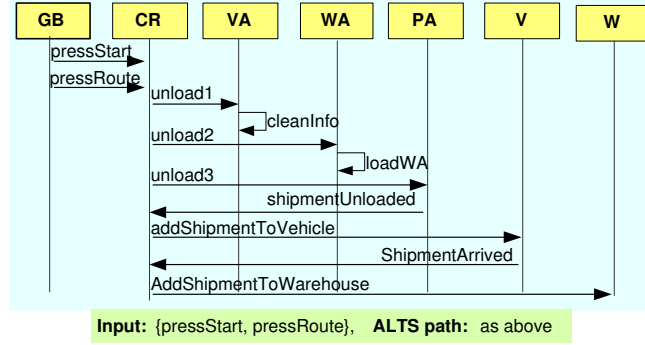


Fig. 9. Cargo Router system: Architectural Test Case ATC #42

sion 2); 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 (PlA) components), which automatically selects the incoming shipment with the longest wait, fastest available vehicle and emptiest warehouse.

Figure 8.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.

When applying the SA-based conformance testing to the Cargo Router system, we produced an SA specification of the system in terms of C2-style architecture modeled by using the Argus-I tool [13] and behavioral models in the FSP algebra. A testing criterion has been identified to focus on “*all those behaviors generated by a routing event*” (hereafter called, *Routing Criterion*). An Abstract Transition System (ATS) has been produced and 164 architecture-level test cases have been produced using the McCabe’s path coverage criterion [36]. Figure 9 shows one of the ATCs previously identified. The ATC in Figure 9, has been mapped to six different code-level test cases. We finally used the Argus-I tool monitoring and debugging capabilities [13] to make a *deterministic* analysis [11] 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 [28,29].

4.2.1 Goal 1: Test Conformance of P' to the initial Cargo Router SA (overview of results)

We submitted the Cargo Router implementation P to two different modification. P_1' modifies the use of the “random function” in class *Vehicle.java* to select (at startup) vehicles available for shipments, and P_2' includes a new feature that supports vi-

sualizing “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 [5], where the *CalcDiff* algorithm for object oriented programs is presented. The tool JDIFF implements the *CalcDiff* algorithm: given two Java programs, it initially matches classes, interfaces and methods in the two versions, then, it produces enhanced control-flow graphs for matched methods, and performs the comparison between methods in the original version and the corresponding method in the modified version.

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.

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.

4.2.2 Goal 2: Test Conformance of an Evolved Architecture of the Cargo Router

Here, we follow and describe the steps in Figure 2, Goal 2.

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

The Cargo Router version 2 architectural specification consists of 305 lines of FSP statements and the resulting global Transition System TS'' is composed by 360,445 states and 869,567 transitions. By focusing on the *Routing* testing criterion (previously identified), we produced an ATS composed by 8,448 states and 55,200 transitions.

Step c: Comparing S with S''

In [32] 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 8, 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 TSs. 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 the LTSA “Animator” feature to make a comparison among the two ATCs. Notice that the comparison is made at the abstract level (ATS, instead of TS) since ATCs are extracted from ATCs.

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 ATS which has been modified in the S'' ATS. We report some interesting results by considering a few of the ATCs generated in Section 4.2.

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'' , and since the path was not affected by other components' changes, we are guaranteed that ATC #12 in the ATS 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'' , thus we should expect ATC #26 needs to be retested. However, when applying the architectural diff among ATS and ATS'' , 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 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 ATS, ATC #42 covers nodes which have been modified in ATS'' . Thus, ATC #42 needs to be retested on S'' .

To check the differences between ATS and ATS'' , we used the LTSA "Animator" feature which allows paths simulation in an ATS 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 9). 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 merchandize).

4.3 Comparison and Discussion

Figure 10 describes similarities and differences among the two conducted experiments:

Specification Language: a C2 style architectural specification has been used to

	Specification Language	ATS generation process	Code-level Diff algorithm	SA-level Diff algorithm	Tool support and Technologies
Elevator	C2 and statecharts	FC2 Tools	DejaVOO	SADiff	Argus -I, statecharts, FC2 Tools, DejaVOO, SADiff
Cargo Router	C2 and FSP	New FSP specification	JDIFF	LTSA Animator	Argus -I, LTSA, McCabe's, JDiff,

Fig. 10. Similarities and Differences

model the topology of both the Elevator and the Cargo Router examples. This style has been chosen for two different reasons: it is supported by the C2 framework, which helps to make rigorous the mapping between SA test cases and code test cases and simplifies the test case execution thanks to the Argus-I tool support. While it is important to note that the research here proposed is not tied to C2, its coding process and monitoring tools support heavily simplified some of the aforementioned discussed steps.

Regarding the behavioral specification, while the Elevator example architecture has been modeled via statecharts, the behavior of Cargo Router's components has been modeled using the FSP algebra, in order to produce a TS.

ATS generation process: in the Elevator example the concept of *obs-function* has been implemented by using the FC2Tools [1] suite of graph transformation tools, while in the Cargo Router example a new FSP specification has been realized to model the abstract TS.

Code-level Diff algorithms: two different algorithms have been used for this purpose: DejaVOO and JDIFF. While DejaVOO implements entirely a regression test selection technique and has been only partially used in our experiments, JDIFF has provided more appropriate tool support for our purposes.

SADiff algorithms: the SADiff algorithm has been implemented in two different ways: a home made, prototypal, SADiff algorithm (which compares the XML representation of both graphs) and the LTSA Animator feature, which allows for a step-by-step animation of the transition systems.

Tool support: Argus-I has been used to specify the SA topology of both the Elevator and the Cargo Router systems. Then, two different directions have been taken.

The Elevator behavioral specification has been provided in terms of statecharts, with the semantics described in [12]. The FC2Tools have been used to abstract the global statechart model based on the defined observation function. Argus-I features have been used to run the deterministic testing of the system. DejaVOO has been utilized to compare the graph representations of the different versions of the code. The SADiff algorithm has been utilized to compare the SA specifications.

The Cargo Router behavioral specification, instead, has been provided in the FSP algebra, thus generating a TS via the LTSA tool. The ATS has been generated by

modifying the FSP specification according to the observation function. The McCabe’s coverage criteria has been used to generate the ATCs, through an ATS graph coverage. JDIFF has been used to identify how the implementation evolved. The LTSA tool simulation features have been used to compare the SA specifications.

	(a) SA version	(b) # components/ connect ors	(c) FSP: lines of spec.	(d) # of TS states	(e) # of TS transitio ns	(f) # of ATS state s	(g) # of ATS transit.	(h) # of ATCs	(i) # of ATCs re-run	(l) Code Version	(m) # of TCs re-rerun	(n) # of faults found
Elevator	v1	3/2		356	2,204	9	14		1/2	-	-	-
		6/4								v5		
	v2			4,852	32,504	28	35	24		v5B	18/18 *	3
									v5C	9/18	3+injected	
CargoRouter	v1	10/4	190	21,144	133,644	80	244	164	6/42	P		
										P1'	0/18	None
										P2'	10/18	1
	v2	17/8	305	360,445	869,567	8,448	55,200	--		-	-	-

* 18 test cases for two ATCs

Fig. 11. Results comparison: *a*) versions of the SA specification, *b*) number of components and connectors in the SA, *c*) number of lines of FSP specification, *d*) number of states in the SA transition system, *e*) number of transitions in the SA transition system, *f*) number of states in the SA abstract transition system, *g*) number of transitions in the SA abstract transition system, *h*) number or architectural test cases (ATC), *i*) number of ATCs to be rerun, *l*) versions of the SA implementation, *m*) number of test cases to be rerun, *n*) number of faults identified

Based on a comparison of the results obtained from the application of SARTE to the two case studies (see Figure 11), we can conclude that the approach has been applied so far to small-medium size systems (from 3 to 17 components) with a behavioral model complexity which ranges between 356 to 360,445 states. Regarding Goal 1, the best results have been obtained with the Cargo Router system, where none of the 18 test cases need to be rerun when moving from P to P'. Regarding Goal 2, the best results have been obtained with the Cargo Router system: when moving from SA version 1 to version 2, the 85% of architectural test cases were not needed to be retested. More experiments, based on industrial-size architectures, are needed to provide a definite evaluation on the savings permitted by the proposed approach.

5 Lessons Learned and Considerations

Based on experiments performed over the Cargo Router and the Elevator systems, we identified the following lessons learned:

Comparison with traditional RT: 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

the new code version. 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 a new code version. *ii*) When an ATC is discarded (e.g., ATC #12 and ATC #26, Cargo Router system), the retest of all such TCs related to ATC are avoided, thus reducing retesting effort.

Testing Criterion (goal 1): Code-level test cases are selected according to a testing criterion, previously identified at the architecture level. The test cases execution, thus, cover only a portion of the system implementation (according to the testing criterion). When the system implementation changes, only some of the system features and code portions are affected by the change. If the modifications affects code portions not covered by the testing criterion, test cases might not traverse such changes, avoiding test case (re)selection.

Architectural test case to be retested (goal 2): 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'', we are not guaranteed it does not need to be retested. If the architectural path is affected by other components' changes, we might still be required to rerun it (as in the Elevator system example in Section 4.1.2). *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 $SA_{diff}(ATS,ATS'')$ does not identify any real change (with respect to what we are interested in testing), thus avoiding re-selection of the ATC. Since one ATC can be mapped into many code-level test cases, this represents a significative cost reduction not only in the test execution (for reducing the number of test cases to rerun), but also in the regression test case selection process (for reducing the number of test cases to check the need for rerun).

Architectural Diff (goal 2): In previous experiments we have performed, 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: 1) checking ATC'' against ATC may be a bottleneck when the number and sizes of paths become large, and 2) there may be paths in ATC'' that do not 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 have tried to make a structural-only comparison between S and S'' (without considering the behavioral specification). We found 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.

C2 style and Model Driven Architecture: 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. Darwin, ArchJava, Fujaba, MetaH), which support a rigorous SA-based coding process [27].

The idea of driving the code realization from system specifications is certainly not new, and it has been recently re-discovered in the software architecture field, also thanks to the Model Driven Architecture (MDA) [31] trends. In MDA, in fact, the focus in software development is shifted from coding to modeling, while separating the application logic from the underlying platform technology. Models are primary artifacts retained as first class entities. They can be manipulated by means of automated model transformations, and utilized to produce the system implementation (while moving from abstract and functional Platform Independent Models (PIMs) to more technical and refined technologies, when moving to Platform Specific Models (PSMs)). Then, assuming to have a precise mapping between architectural elements and implementation is not too restrictive, and future work on integrating SARTE and MDA technologies will be described in Section 7.

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. Section 7 describes ongoing work on this direction.

Evaluating SARTE: Five are the main categories discussed in [35] which should be taken into account when evaluating selective retest strategies. The five categories are: *inclusiveness* (which measures the extent to which a selective retest strategy chooses “modification revealing” tests⁴), *precision* (which measures the ability of a RT technique to omit tests that are non-modification-revealing), *efficiency* (computational cost and automation), *generality* (which measures the ability of the RT technique to handle realistic changes in realistic contexts), and *accountability* (which evaluates the approach ability to promote the use of coverage criteria). We briefly describe here how each of them can be applied to SARTE. Since SARTE’s Goal 1 is implemented through traditional regression testing techniques (then, its evaluation depends on the specific approach applied), we mainly focus on Goal 2.

Regarding inclusiveness, SARTE can be considered a “safe” regression testing technique (100% inclusiveness), since it selects all the changed nodes/arcs in the Transition System model. However, SARTE differently from traditional “safe” techniques, identifies at a high level those aspects that are expected to change and gives guidance for testing those. Regarding precision, our current experience let us believe that SARTE is precise, since it never selected a non-modification-revealing

⁴ tests which will produce different outputs in P and P' .

test. Further analysis on real systems will validate/confutate such belief. When dealing with efficiency, two main operations are required by SARTE: to compare the two architectural specifications (both structure and behavior) and to select those test cases in the architecture S , which may cover different paths in the different version S'' . SARTE is more general than other approaches, since it covers both SA- and code-level test case reselection, and it is not tied to a specific architecture description language. To conclude, in the context of accountability, the architectural specification is initially “pruned” according to a selected observation function, then, a path coverage criteria is applied to select architectural test cases. SARTE reiterates such criteria when moving from testing to regression testing.

6 Related Work

For a description of existing SA-based testing techniques we direct interested readers to [28]. Here, we outline some relevant research on specification-based regression testing, and how our work is related to them.

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 [25,37,8] extract the system specification from the code through a reverse engineering approach. In [25], an Object-Oriented Software Testing and Maintenance Environment (OOTM) is extracted from the source code in order to generate test cases. The Class Firewall tool analyzes changed classed and the OOTM model in order to select reusable test cases. In [37], the authors compare the control flow graph of the original code with the one extracted from the modified program, building an Affected Function Dependence Graph (AFDG), to identify affected variables/functions. In [8] both control- and data-flow information is gained from the code, specified through different diagrams and used to identify reusable test cases.

These approaches differ from our in two major ways. First, 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 reverse engineered specification.

Some approaches to regression testing at the component level have been presented [20,38]. In [20] 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 [38], the test model the authors present is realized using the Component Interaction Graph, which depicts the interactions and dependence relationships among components. In our approach, we are interested in identifying architectural level errors and, in this context, we also depict the interactions and dependencies among components, but these were derived from the architectural specification, and not component metadata. The advantage of our approach here is that we can identify problems not only in the context of individual components (or sets of component), but also of the whole architecture - the architectural specification provides a global picture of the system that is not provided by simply composing components and their metadata.

Orso et al. [33] present a scalable, safe and precise regression testing selection technique for object-oriented programs based on a two-phased algorithm (partitioning and selection). While the partitioning phase builds a high-level graph representation for P and P' , in order to identify the parts of P and P' to be further analyzed, the selection phase builds a more detailed graph representation for those identified parts, and conventionally selects the set of test cases in T that transverse the changes. One of the similarities between this technique and our approach is that both tackles the issue of effective and efficient regression testing selection by combining the use of two different abstractions: a higher and a lower level. However, the high-level representation in our approach is based on the architectural description (and not a high-level graph extracted from the code), and our goals include the identification of architectural drifts as well.

To the best of our knowledge, [18] is the only research work presenting ideas on using software architecture for effective regression testing previously to ours, and it explores the idea of applying similar code-based regression testing techniques over graphs derived directly from CHEM architectural descriptions, and using those graph similarly to a program control flow graph in a traditional RT technique. That paper misses further work in applying SA-based regression testing to actually test program code. In this work, we propose a SA-based RT technique that can actually test program code, based on the SA description, and performed some case studies that gave us further confidence about the effectiveness of our approach.

7 Ongoing and Future Work

This Section introduces ongoing work on how we are planning to use SARTE to handle the architectural drift, and future ideas on how to extend/improve SARTE in novel research domains.

7.1 Ongoing work: Handling the Architectural Drift

The architectural drift [34] has been always a serious problem, caused by a frequent evolution of the implementation, without the architectural design adaptation to new low-level decisions. It is not uncommon, in fact, that during evolution 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. When the architecture “drifts” out of conformance with the implementation, the two artifacts are not anymore aligned and become two unrelated documents.

By means of software architecture regression testing, we can test the modified implementation and adjust the architectural model according to the updated implementation. Thus, the goal is to reconstruct the SA according to the modified implementation.

Goal 3: architecture reconstruction.

- Context: Given a software architecture specification S , its initial implementation P and a modified version P' . The test case t has been run over P without detecting any architecture-related fault. Test case $t' = t$ has been selected to be rerun over P' and its execution has revealed an architectural fault.
- Goal: To reconstruct the actual architecture when the first goal determines that the code no longer conforms to the initial architecture.

In particular, going back to the SARTE project, the architectural drift can be handled by implementing the following steps:

Step i: since t' reveals an architectural fault, its execution over P' identifies an unwanted architectural behavior, expressed in terms of an architecture level sequence diagram SC (like for example the scenario in Figure 6). SC identifies which components are involved in the fault (`BuildingPanel`, `ElevatorADT1` and `ElevatorADT2` in the example) and how the SA behavior needs to be adjusted in order to comply to the implementation;

Step ii: the SA conforms to the implementation if SC is a feasible path in the SA behavioral model. In order to enforce SC into the SA specification, the transition system model associated with the components involved in SC needs to be adjusted in order to make SC a valid scenario. A synthesis process from scenarios to state machines can be performed in order to generate the new SA specification. This activity is driven by SC and wants to reconstruct the SA transition system according to the modifications made in SC to the original SA specification. For each component C in SC , the synthesis process generates (a portion of) the new transition system TS' for C , used to guide the modification to the original TS for C .

A further theoretical discussion on this topic is certainly a relevant issue, but out of

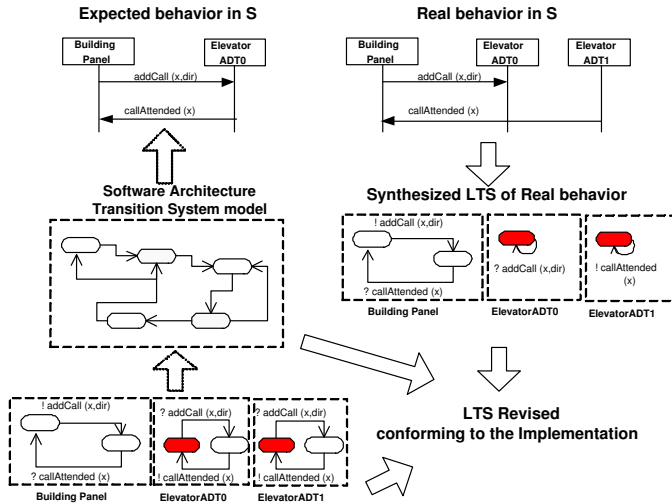


Fig. 12. How to handle architectural drift

the scope of this paper and left to further work on the topic. However, we applied this sketchy approach to the Elevator example, in order to gain some initial practical experience.

Regression testing applied to the Elevator example (Section 4.1.1) identified a t' which revealed the following fault: “while the elevator call is handled by the ADT1 component, the CallAttended message is generated by the ADT2 component”. Assuming we want to analyze how much this change affects the Elevator system architecture, we have to adapt the architectural specification according to this scenario SC. We have then to understand how to propagate the SC behavior back to the TS model in order to adjust the initial SA specification. By applying a synthesis techniques to the **Building Panel**, **ElevatorADT1**, and **ElevatorADT2** components, we obtain an excerpt of the modified transition systems. Figure 12 summarizes our ongoing work on this direction.

7.2 Future work: SARTE and MDA

The introduction of UML as a de-facto standard in software systems modeling has facilitated the introduction of UML-based notations for specifying SAs. Many proposals have been presented so far to adapt UML 1.x (see [26] for a survey on the topic) and more recently UML 2.0 (e.g., [21]) to model software architectures.

In parallel, the Model Driven Architecture (MDA) [31] philosophy of shifting the focus of software development from coding to modeling and building systems via models’ transformations, has strongly impacted the way software architecture specifications (in the UML) could be used in the near future for code generation purposes (see, for example, current work on the Fujaba project [3]).

In the context of SARTE, future work will investigate how to realize a model-based specification, testing and regression testing of software architectures. In this scenario, UML-based notations will be utilized for the specification of software architectures, and code generation techniques from models will be used to guarantee the alignment among SA models and the executable implementation.

7.3 *Future work: SARTE and Product Line Architecture*

When dealing with the software architecture of a product family, the concept of Product Line Architecture (PLA) (e.g., [23]) has been introduced as an emerging evolution to SA modeling for product family. Whereas an SA defines the structure and behavior of a single product, a PLA precisely captures, in a single specification, the overall architecture of a suite of closely-related products [9]. A PLA specification focusses on modeling *mandatory elements* (which are present in the architecture of each and every product), *optional elements* (which may or may not be present in a product), *variant elements* (which must be present in a product architecture but can be chosen to be one of a number of different alternatives) [22] and explicitly models connection and communication constraints.

Specific *product architectures* can be selected from the PLA thanks to a decision model which permits to make product-specific decisions. Many product architectures can be generated from a PLA.

In the context of PLA, assuming a product architecture PA1 has been selected and tested, whenever a new product architecture PA2 is selected, SARTE can be used for testing PA2 according to previous results obtained when testing PA1.

7.4 *Future work: SARTE for Testing Dynamic and Service Oriented Software Architectures*

A dynamic software architecture represents a system which can evolve at compile time or runtime, by adding, updating and removing components or connections [32]. When dealing with architectures evolving at runtime, the architecture does not simply consist of a fixed set of static elements, but may become an open system, ready to evolve according to some pre-defined rules or self-adapting to the external requests.

The concept of open system becomes even more accentuated when dealing with service oriented architectures (e.g., [15]), where, components available on the network are automatically discovered and integrated at runtime, while implementing the “find, bind, and execute” paradigm.

SARTE as discussed so far, permits to handle compile-time evolution and can be considered simply as the first step in the direction of a more ambitious project which wants to test dynamically evolving, adaptable, software architectures. Indeed, major modifications are needed for extending SARTE to runtime evolving architectures: runtime mechanisms to discover how the architecture evolves are needed (a sort of monitoring over the architectural components), the assumption that the architectural model of the system exists needs to be relaxed (we can only assume to know the behavior of single components, while the configuration and integration varies over time), the retesting time must be reduced as much as possible (otherwise, we might test an architectural configuration which is not anymore running).

8 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. 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 cases where the code evolved relative to unaffected software architecture; and we also explored cases where the architecture evolved as well, and not only the implementation. In both situations, the proposed approach was applied to the Elevator system and 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 different architectural diff utilities.

A more precise experimental results (based on industrial-size architectures) are needed to provide a definite evaluation on the savings permitted by the proposed approach. Moreover, the SARTE application to real and complex architectures will permit to evaluate how much the approach scales to real systems. From a preliminary analysis, we may conclude that bigger architectures concerning real systems may require a bigger computational time to apply the observation function, and a bigger number of architectural test cases can be produced according to the testing criterion. In order to empirically validate our initial analysis, we are planning to expose the Siemens C.N.X. and Terma GmbH architectures model-checked and tested in [10,30] to SARTE.

Even if this project represents the first step in the direction of more ambitious projects (described in Section 7), we hope this proposal may attract interest on the issue of software architecture-based regression testing.

Acknowledgments

We are indebted to Leila Naslavsky, Lihua Xu, Yuewei Zhou, and Tim Standish of the ROSATEA research group, for their valuable insights, and we thank Nenad Medvidovic, who provided the Cargo Router system, and Alessandro Orso, who provided some guidance on DejaVOO.

References

- [1] FC2Tools. <http://www-sop.inria.fr/meije/verification/quick-guide.html>.
- [2] The C2 Architectural Style. On-line at: <http://www.ics.uci.edu/pub/arch/c2.html>.
- [3] Fujaba Project. <http://wwwcs.uni-paderborn.de/cs/fujaba/publications/index.html>, 2005. University of Paderborn, Software Engineering Group.
- [4] R. J. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of the 1998 Conf. on Fundamental Approaches to Software Engineering (FASE '98)*, March 1998.
- [5] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, Linz, Austria, september 2004.
- [6] M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods*. SFM-03:SA Lectures, LNCS 2804, 2003.
- [7] A. Bertolino, P. Inverardi, and H. Muccini. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods*, chapter Formal Methods in Testing Software Architectures, pages 122–147. SFM-03:SA Lectures, LNCS 2804, 2003.
- [8] S. Beydeda and V. Gruhn. Integrating White- and Black-Box Techniques for Class-Level Regression Testing. In *Proc. COMPSAC 2001*, pages 357–362, 2001.
- [9] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000.
- [10] A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In *Proc. International Testing Methodology Workshop*, Lecture Notes in Computer Science, LNCS, vol. 3236, pp. 351 - 365 (2004), October 2004.
- [11] R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Trans. on Software Engineering*, 24(6):471–490, June 1998.

- [12] M. Dias and M. Vieira. Software Architecture Analysis based on Statechart Semantics. In *Proc. of the 10th Int. Workshop on Software Specification and Design (IWSSD-10)*, pages 133–137, San Diego, USA, November 2000.
- [13] M. Dias, M. Vieira, and D. J. Richardson. Analyzing Software Architecture with Argus-I. In *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 758–761, Limerick, Ireland, June 2000.
- [14] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *J.C.P. Woodcock and P.G. Larsen (Eds.), FME'93: Industrial-Strength Formal Methods*, pages 268–284. LNCS 670, Springer Verlag, 1993.
- [15] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.
- [16] D. Garlan. Software Architecture: a Roadmap. In *ACM ICSE 2000, The Future of Software Engineering*, pages 91–101. A. Finkelstein, 2000.
- [17] 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)*, pages 188–197, Japan, April 1998.
- [18] 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*, pages 73–77, July 1998.
- [19] M. J. Harrold. Testing: A Roadmap. In A. Finkelstein, editor, *ACM ICSE 2000, The Future of Software Engineering*, pages 61–72, 2000.
- [20] 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*, November 2001.
- [21] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Carnegie Mellon University, Software Engineering Institute, 2004.
- [22] M. Jaring and J. Bosch. Representing Variability in Software Product Lines: A Case Study. In *SPLC*, pages 15–36, 2002.
- [23] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [24] J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A case study. In *Proc. 4th IEEE Int. Conf. on Configurable Distributed Systems*, May 1998.
- [25] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim, , and Y. Song. Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of the ACM*, October 1995.

- [26] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1), January 2002.
- [27] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.
- [28] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.
- [29] H. Muccini, M. Dias, and D. Richardson. Systematic testing of software architectures in the c2 style. In *Proc. Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2004), ETAPS2004*, volume Lecture Notes in Computer Science, LNCS 2984, Barcelona, Spain, 2004.
- [30] N. D. Nisio, M. Cardone, and H. Muccini. SA-based Model Checking and Testing vs. traditional approaches: an experience on the NIS. In *ESA/ESOC Ground Operation Software System (EGOS) Workshop*, Darmstadt, Germany, November 2005.
- [31] Object Management Group (OMG). *OMG/Model Driven Architecture - A Technical Perspective*, 2001. OMG Document: ormsc/01-07-01.
- [32] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-Based Runtime Software Evolution. In *Proc. Int. Conf. on Software Engineering, ICSE 98*, April 1998. Kyoto, Japan.
- [33] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, Newport Beach, CA, USA, November 2004.
- [34] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. 17(4):40–52, October 1992.
- [35] G. Rothermel and M. J. Harrold. A Framework for Evaluating Regression Test Selection Techniques. In *In Proc. 16th Int. Conference on Software Engineering, ICSE 1994*, pages 201–210, Sorrento, Italy, May 1994.
- [36] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication 500-235*, August 1996.
- [37] Y. Wu, M. H. Chen, and H. M. Kao. Regression Testing of Object-Oriented Programs. In *Proc. Int. Symposium on Software Reliability*, 1999.
- [38] Y. Wu, D. Pan, and M. H. Chen. Techniques of Maintaining Evolving Component-Based Software. In *IEEE Proc. Int. Conf. on Software Maintenance*, 2000.