Systematic Testing of Software Architectures in the C2 style

Henry Muccini[1], Marcio Dias[2] and D. J. Richardson[2]

[1] Universita' degli Studi dell'Aquila
Dipartimento di Informatica,
Via Vetoio, 1 – L'Aquila - Italy
muccini@di.univaq.it
[2] School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA
[mdias,djr]@ics.uci.edu

Abstract. The topic of software architecture (SA) based testing has recently raised some interest. Recent work on the topic has used the SA as a reference model for code conformance testing, to check if an implementation fulfills (conforms to) its specification at the SA level. In this context, on previous papers, we have analyzed: i) how suitable test cases can be "selected" from the SA specification and ii) how they may be "refined" into concrete tests executable at the code level. While the selection stage has been done systematically, the refinement step has been left to be done manually, based on the software engineer knowledge on how to map "abstract values of the specification to the concrete values of the implementation". In this paper, we extend previous approaches, by providing a systematic way to perform the refinement step. We show how choosing a specific architectural style, which supports implementation and facilitates the mapping among SA-based and code-based test cases, a completely systematic SA-based testing approach can be delivered.

1 Introduction

Software testing consists of the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified behavior" [5]. Traditional approaches to software testing select test cases based on the source code of the program to be tested [16].

With the advent and use of software specifications, source code no longer has to be the single source for selecting test cases: formal, informal and model-based specifications can be used for this purpose [5, 16]. The importance of the use of formal methods in software specification and design does not need to be stressed here. Several authors have highlighted the advantages of formal methods in testing, and different techniques have been proposed to select tests from semi-formal specifications [24], algebraic specifications [3], model-based specifications [12] and Software Architecture-based specifications [4, 25].

Particular interest has been devoted to *specification-based conformance testing* [8, 14, 30]. Conformance testing checks that an implementation fulfills its specification. Several authors [8, 14, 30] have dealt with the problem of *automatically generating test suites to test the conformance of an implementation under test (IUT) to its specification*, where both specifications and IUT are expressed in the form of Labeled Transition Systems (LTS), Finite State Machines (FSM) or Input/Output LTS (depending on the approach).

Some interest has been devoted to *Software Architecture-based conformance testing* [6, 7]. Given a software architecture (SA) description, conformance testing has been used to detect conformance errors between the SA specification and its implementation. The SA specification has been used as a reference model to which the source code should conform.

One of the most challenging problems of SA-based conformance testing is the necessity of a common model that makes it possible to compare the expected behavior of a SA with its real implementation. A common model would bridge or map the elements from these two different abstractions, addressing the so called "mapping problem" or "traceability problem". Traceability concerns "relating the abstract values of the specification to the concrete values of the implementation" (as quoted from [12]). Several researchers have recognized the importance and difficulty of this step [12, 32, 27], which has been deeply analyzed in [6].

The main goal of this paper is to review and extend our previous work on SA-based conformance testing, to provide a ***systematic way*** to use an SA for code testing. This research is driven by a previous analysis we performed in [7], where we identified the factors making the distance between SA and code high. As a result, the development process, the relationships among architectural components and the source code, and the SA-level of abstraction strongly influence that distance. If "explicit mapping rules (could) drive the source-code implementation from architectural components, connectors, and messages" [7], then the mapping problem could be easily managed.

In this paper, we propose a specialization and refinement of our general approach for SA-based conformance testing, in order to obtain a systematic approach for performing code level conformance testing based on SA specifications. Here, we deal with this problem in a specific SA style, the C2 style [29, 10]. We show how the SA to code mapping rules imposed by the C2 framework helps to limit the mapping problem, and allows a systematic testing approach.

This research is part of a project called SARTE (SA-based regression testing) [22], which aims to provide an approach and framework to test software architectures and code, when both are subject to changes.

The rest of the paper is structured as follows: Section 2 provides an overview on Software Architecture-based Testing. Section 3 describes the research outline and our proposal. Section 4 describes the C2 architectural style and presents the case study used in this paper. Section 5 explains how the general approach mentioned in Section 2 can be specialized to C2 style architectures. Some results from our experiment are presented in Section 6. We discuss some related work in Section 7, and our conclusions and future work in Section 8.

2 Software Architecture-based Testing: an Overview

The topic of *architectural testing* has recently raised some interest [25, 4, 16, 27, 7].

In [25], the authors define six architectural-based testing criteria, adapting specification-based approaches; in [4] the authors analyze the advantages of using SA-level testing for reuse of tests and to test extra-functional properties. In [16] the author presents a discussion on the use of software architecture for testing. In [27], the authors present an architecture-based integration testing approach that takes into consideration architecture testability, simulation, and slicing.

The approach proposed in [7] is, to the best of our knowledge, the first effort to tackle the whole cycle of SA-based testing with a comprehensive and concrete approach. It spans the spectrum from test derivation based on architecture dynamics down to test execution over system implementation.

The general approach is composed by some logical steps which can be outlined with the help of Figure 1.

In *Step 0*, a topological and behavioral specification of the SA is required.

In *Step 1*, a software architect, by looking at the software architecture dynamics from different viewpoints, defines various testing criteria [5]. Each criterion highlights a specific perspective of interest for a test session and is realize through an *obs-function.*

Step 2 derives, through the selected obs-function, an Abstract LTS (ALTS), which still expresses all high-level behaviors we want to test, but hides any other irrelevant behavior.

Step 3 uses the ALTS in order to select an architecture-level test suite. Each "architectural test" is a sequence of architecture-level actions meaningful with respect to the testing criterion.

Finally, *Step 4* uses the architectural test cases to actually test whether the source code conforms to the architectural description. This step has to identify how SA-level abstract test cases can be related to concrete values of the implementation (i.e., *traceability/mapping* among SA and code) and how the code may be run over the identified test cases. The traceability problem has been handled by using an informal "mapping" function while the execution traces are analyzed to check whether the system implementation works correctly with respect to the architectural tests.

The goal of this paper is to improve and refine that work, in order to handle, in a systematic way, all the testing approach, as outlined in Section 3.

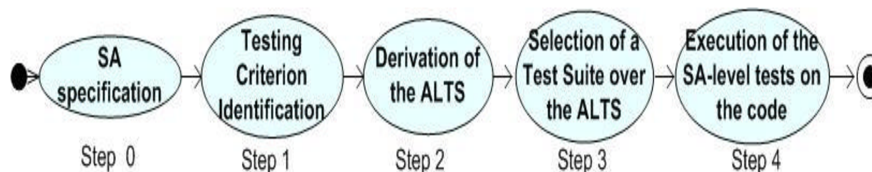

Figure 1. An Activity Diagram of a SA-based Testing

3 Research Outline

The SA-based testing process proposed in [7] is largely mechanical, but some important human interventions are required.

The *test selection stage* (steps 0 to 3) is systematic: the SA is formally specified using an ADL of a model-based specification, the LTS is automatically generated from the specification, the ALTS can be automatically generated using existing tools, the SA test selection process is implemented using existing tools to cover the ALTS.

The *test execution stage* (step 4), instead, is informal and left to the software engineer ability. In particular, a software engineer has to manually deal with the traceability problem, i.e., "relating the abstract values of the specification to the concrete values of the implementation" [12]. This important problem has been already recognized by other researchers [32, 27] but never formally handled.

In previous papers [6, 7], we managed such problem in a very general context, supposing that a well-formalized architecture-based development process was ***not*** in place (as happens in real world) and the SA specification and the low-level design have been intermixed without any formal mapping. One advised solution was to use some development support which explicitly adopts a formal mapping between architectural and implementation elements.

In this paper we specialize and refine some of the activities presented in Figure 1. We choose a specific architectural style, the C2 style, which supports implementation and facilitates the mapping among SA-based and code-based test cases. We enrich the C2 structural specification with a behavioral one, to accommodate a behavioral model of the system. We reuse and adapt previous experience to identify and select relevant architectural test cases. In particular, we completely revise steps 3 and 4.

We use existing tools (namely, Argus-I [2]) in order to run deterministic testing. We thus apply this technology to the Elevator case study, described in Section 4.2.

4 Preliminary information

In this section we provide information which will be useful in the following. We outline the C2 style for describing software architectures and we present the Elevator case study, used in Section 5 to put in practice the proposed approach.

4.1 C2 style Software Architectures

C2 [10, 29] is an architectural style introduced in 1995 by researchers from the University of California, Irvine. This style imposes some compositional and behavioral rules [29] enabling some level of independence (called "substrate independence") between the components used to describe the SA.

Components have visibility only on components up to them but they do not need any information on components beneath them. Moreover, communication may happen only through the explicit use of connectors. Each component and each connector

exposes exactly two interfaces, to send “requests” and receive “notifications”; a request consists in requiring a service to other components while a notification identifies the output of a request. “Links” are used to configure a C2 style architecture, by relating component and connector interfaces. C2 components, connectors, interfaces and links can be visualized and analyzed using, for example, the Argus-I [2] tool.

The *C2 framework* [10] helps software engineers to produce the actual implementation of the architecture. In the following, we assume our SA complies to the C2 style and the implementation is realized through the C2 framework.

4.2 The Elevator case study and its Software Architecture

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

In the configuration for our case study, the elevator system contains the *building panel* (which includes all the panels from different floors of the building), two *elevator cars,* and a *scheduler* algorithm to assign calls requested through the building to the closest elevator car.

The components of this elevator system are:

- ***ElevatorADT***: this component maintains the information about the elevator car state, such as: motion {moving, stopped-closed, stopped-opened} and direction {up, down}. In addition to state information, the ElevatorADT keeps a list of all the calls it needs to attend.
- ***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.
- ***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 “runtime” scheduling policy so that if a call is made at time “t”, it initially selects the elevator car (EC) that, at time “t”, could attend it with the lower waiting time required. At time “t+i”, i = {1, …, n}, the Scheduler checks if EC is still the best option, and, if it is not anymore, can switch to another elevator. This process is repeated until the call is served by one elevator.
- ***Synchronizer***: this component synchronizes the elevator movements, so that all of them makes a move at the same time.

5 Systematic Testing of C2 style architectures

In this section we describe how the generic SA-based conformance code testing proposed in [7] and outlined in Section 2 can be instantiated to C2 style architectures in order to become systematic. The theory is thus applied to the Elevator example described in Section 4.2.

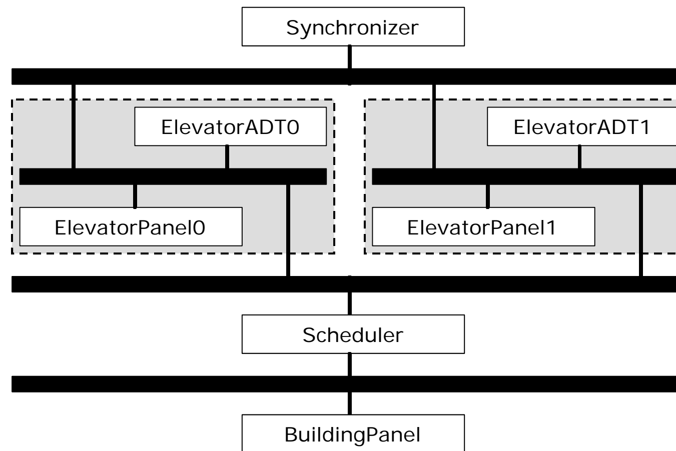

Figure 2. The Elevator C2 style architecture

5.0 Step 0: SA specification

The system software architecture is modeled following the C2 style. Figure 2 shows the Elevator architecture, as visualized by Argus-I [2]. Many requests and notifications are sent and received by components, through the five connectors shown in Figure 2 (thick horizontal lines). Such requests are used to realize different services, like *AddCall*, *CallAttended*, *GetId*, *Suspend Clock*, *Resume Clock*. (The description of the C2 architecture, complete of requests and notifications, may be found in [23], Appendix A).

It is to be noticed that the C2 specification is just structural, and a behavioral description of components and connectors interaction is missing. Some work has been done in the past on this direction, formalizing a portion of C2 in the Z specification language [19] or by using pre- and post-conditions, but they are still really preliminary. We thus decided to complement the C2 style specification with a behavioral one. We use the Finite State Process (FSP) [18] algebra. In this notation, each component and each connector is represented through a process. The behavior of each process can be automatically represented through a Labeled Transition System (LTS) and those LTSs can be combined together (following the FSP semantics of composition) in order to produce a global LTS, describing how components and connectors work together. We choose this language, since it is tool supported (LTSA tool [17]) and we had previous experience with it. We modeled the behavior of each

single Elevator component in FSP. The FSP specification for the Elevator architecture can be found in [23], Appendix B.

By using the LTSA tool over the FSP specification, an LTS has been generated for each component. Such LTSs have been combined together generating a global LTS composed by 4288 states and 26336 transitions (before minimization), representing how the system evolves when different services are requested or notified.

This model will be used in the following to extract SA-level test cases.

5.1 Step1: Definition of an observation

In principle, the global LTS could directly be used as the reference model for deriving the test scenarios, assuming that an architectural test is essentially a *sequence of system actions meaningful at the architectural level*. Unfortunately, by considering this global graph, it is very difficult to realize the testing *selection stage* [7], i.e., to "suitably" select a set of test cases. In fact, the LTS provides a tremendous amount of information flattened into a graph (i.e., many messages, services, parallelism, interleaving, many components interacting and so on). By identifying some *testing criteria* [5], we can select only such behaviors suitable for the criterion itself, abstracting away uninteresting behaviors. We can thus focus on a subset of relevant behaviors.

An *obs-function* (as defined in [7]) which partitions LTS actions into "relevant interactions" *R* (i.e., those we want to observe by testing) and "non-relevant interactions" *NR* (i.e., those we are not interested in at this time) can realize such a testing criterion.

In the context of a C2-FSP specification, we extend the concept of *obs-function* into *structural* and *functional* observations. A structural observation focuses on the SA topological description provided by C2. The software tester, looking at the C2 specification, identifies components/messages/connectors she is more interested to test. A functional observation, instead, tries to capture relevant information from the behavioral model. With a functional observation, the tester can identify system functionalities she is more interested to test.

In this paper, due to space limitations, we describe one functional observation. The functional observation of interest is "*all those behaviors involving the **AddCall** and **CallAttended** services*". In other words, we consider as "relevant" all and only, those interactions necessary to realize the *AddCall* and *CallAttended* services, while hiding the others. In the following, we refer to this testing criterion as the "AddCall+CallAttended" *obs-function.*

5.2 Step 2: Derivation of the Abstract LTS.

An *obs-function* allows to derive an Abstract LTS (ALTS) which satisfies the criterion itself and still expresses all high-level behaviors we want to test in the initial LTS.

The AddCall+CallAttended observation has been produced by modifying the Elevator FSP specification, hiding all such events not relevant for our testing criterion.

The new FSP specification produces an ALTS composed by 41 states and 51 transitions. This ALTS reduces the initial LTS by describing those requests and notifications related to the *AddCall* and *CallAttended* services only.

It is to be noticed that the hiding operator does not delete the components from the specification. It just makes invisible the messages sent and received by the hidden components, still guaranteeing the model correctness.

5.3 Step 3: Selection of a test suite over the observed behavior

In code-based testing, a test case is usually defined as the input value provided to a program P, with the corresponding expected output. At the architecture level, a test case can be defined as:

Definition: *SA Test case*
An SA test case is an ordered sequence of architectural events observed when a certain initiating event is performed.

This definition encompasses two different keywords: the sequence of actions, which represent expected behaviors, and the initiating event, that is, the architectural input which should allow the sequence to happen.

The expected sequence of actions, for a certain testing criterion, can be extracted by applying a path coverage over the ALTS. Each ALTS path represents a sequence of expected architectural events. Many coverage criteria can be applied. A *complete path coverage* criteria can be applied when the ALTS dimension is reasonable, and when we are interested in a thorough coverage. In [7] we proposed to use *McCabe's path coverage* criteria [31] in order to provide a less thorough coverage by identifying only independent paths.

In this paper, we adapt the ***category partition method*** [24]. Given a functional unit of interest, the category partition method requires to identify functional "*parameters*" and "*environment conditions*". A parameter is an input to the functional unit while the environment condition is a characteristic of the system's state at the execution time. Following the category partition method, mutually exclusive "*choices*" are identified for each parameter and condition (i.e., parameters and conditions values), "*constraints*" are identified for each choice (i.e., when a choice may occur) and "*test frames*" are identified by computing the cross-product of the different choices (i.e., choices are combined together).

Given the ALTS for the *AddCall+CallAttended* obs-function previously defined, we applied the category partition method in order to select ALTS paths of interest. The *AddCall* service represents the functional unit of interest. The only input this service receives is the "add call" request from the Building Panel. There are no choices or constraints related to this input. However, there are many environmental conditions to be considered, as reported in Table 1 and described in the following:

- The *Scheduler* ***selects*** which elevator should attend the call, based on the lower waiting time required. ADT0 or ADT1 can be selected to handle the call (condition #1, Table 1);
- The *Scheduler* can ***check again*** which is the best elevator for the call, depending on a periodical check (condition #2, Table 1);
- The *Scheduler* can ***reselect*** the elevator, if the check again condition is true. If the initial choice is still the best, the initial elevator attends the call, otherwise, there is a switch to another elevator (condition #3, Table 1).

Table 1 describes the environmental conditions, with the possible choices and constraints. Table 2, instead, reports the test frames we are interested to test. Test frame #1, for example, means that we are interested to test the following behavior: the elevator ADT0 is selected initially and a reselection process does not happen.

#	Environment Condition	Choice	Constraint
1	*Elevator selection (ES)*	ADT0 or ADT1	Lower waiting time
2	*Check again (CA)*	YES or NOT	Periodical check
3	*Elevator reselection (ER)*	ADT0 or ADT1	Lower waiting time

Table 1: Environmental Conditions, choices and constraints

Since the conditions, their choices and constraints were selected for the AddCall and CallAttented services, a mapping among the ALTS paths and the test frames is always possible. In our example, 144 complete paths were extracted from the selected ALTS and partitioned into the six test frames. Table 2, column three, says that 8 different paths satisfied test frame #1 and #4, while 32 paths satisfied the others (for a total of 144 different paths). Reusing the idea proposed in the category partition method, we can select one ALTS path for each test frame, as representative of the all set. The six representative paths are listed in [23], Appendix C.

#	Test Frames	# of ALTS paths in the partition
1	ES = ADT0, CA = NOT	8
2	ES = ADT0, CA = YES, ER = ADT0	32
3	ES = ADT0, CA = YES, ER = ADT1	32
4	ES = ADT1, CA = NOT	8
5	ES = ADT1, CA = YES, ES = ADT1	32
6	ES = ADT1, CA = YES, ES = ADT0	32

Table 2: Test Frames and ALTS paths

5.4 Step 4: Tests execution over the source code

This step describes how i) SA-level abstract test cases can be related to concrete values of the implementation (i.e., *traceability/mapping* among SA and code) and how ii) the code may be run over the identified test cases. We analyze those two distinct topics in the following subsections.

Refinement of the architectural tests into code-level tests

One of the reasons we decided to start this research using the C2 style, instead of a generic SA, is that C2 is supported by the C2 framework [10], which dictates how C2 style architectures have to be implemented. The C2 framework can be considered as a set of predefined abstract classes and interfaces that have to be implemented following certain constraints when developing a C2 style architecture. The framework allows the software engineer to implement a C2 style architecture in a straightforward manner: each architectural component is implemented by a Java component. Auxiliary classes can be introduced in order to implement specific aspects. SA events have exactly the same signature in the code and in the FSP architectural specification, the mapping is one-to-one based on signature matching.

Thanks to the strong relationship among a C2 specification and its implementation, the mapping between architectural test cases and code-level test cases may be performed systematically. In order to test the code conformance to a selected SA test case, we could run the code, make an elevator call and check if one of the architectural test cases is traversed. However, depending on the system status (e.g., elevators floor and direction, call made in a specific floor to go up or down), all the 144 ALTS paths, in the six test frames, could be executed. This means that we need to refine the parameters and environment conditions previously identified in order to use an SA test case as an oracle.

Let's see how the refinement process may work, by using the ALTS path in Figure 3 (path #1 in [23], Appendix C). This path is representative of the test frame #1 in Table 2. We are interested to test if this specific behavior can be executed at the code level, when the constraints identified in test frame #1 are verified, that is, *assuming that ADT0 has the lower waiting time* (ES = ADT0) *and the periodical check does not apply* (CA = NOT). In order to refine path #1 (as any other path), we identified execution parameters which allow *ADT0 to have the lower waiting time* and *the periodical check not to apply*. We found out that:

- the waiting time constraint depends on the *direction* and current *floor* of the two elevators, on the floor the call is made, and on the direction the user wants to go;
- the periodical check does not apply only when the building panel and the elevator ADT0 are at the same floor and the call has the same direction of ADT0.

It means that path #1 has to be executed when BP(x,y), ADT0(x,y), ADT1(*,*) holds, that is, both BP and ADT0 are at the same floor "x", BP makes a call to go *up/down*

when ADT0 is going in the same direction "y" and ADT1 may be in a generic floor with a generic direction.

Table 3 shows seven code-level test cases. Test case 7, for example, states that when the AddCall is sent, if both BP and ADT0 are at the third floor going up, ADT0 should be selected to get the call and the ALTS path #1 should happen.

Summarizing, the idea is to reproduce the initial condition so that the architectural test case should happen. This is not in general an easy step, but as shown in the next paragraphs, it is made mechanic thanks to the use of Argus-I.

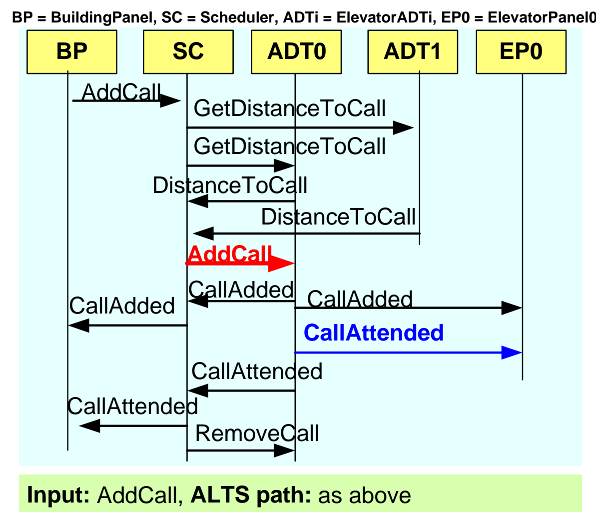

Figure 3: Architectural Test Case for the AddCall+CallAttended observation

Test Path	Test Case
Path 1	1) BP(1,up) ADT0(1,up) ADT1(1,up)
	2) BP(5,down) ADT0(5,down) ADT1(5,down)
	3) BP(5,down) ADT0(5,down) ADT1(5,up)
	4) BP(10,down) ADT0(10,down) ADT1(5,down)
	5) BP(1,up) ADT0(1,up) ADT1(2,up)
	6) BP(3,up) ADT0(3,up) ADT1(2,up)
	7) BP (3,up) ADT0(3,up) ADT1(4,down)

Table 3: Mapping the SA path 1 to low level test cases

Tests execution

Our pragmatic approach here is to make a *deterministic* [9] analysis of the code execution 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 [2], by setting breakpoints during code execution.

We force the system to be in a state described by the test case (as in Table 3), then we try to deterministically force the execution of one of the ALTS paths compliant with the test frame.

When it is not possible to reproduce one of the expected architectural behaviors, the system implementation is not behaving as expected, i.e., it is not conform to the architecture specification. In such cases, an architectural error is revealed.

6 Results and Considerations

We used the C2 framework in order to produce a Java implementation of the Elevator system. Moreover, we produced a faulty version of the same system, injecting some faults. We thus tested the two systems for conformance to the six architecture level test cases selected in Section 5.3 (also listed in [23], Appendix C).

Table 4 summarizes relevant results. We used 42 (code level) test cases over the two versions of the code (V1 is the faulty implementation while V2 is the initial one). SA level test cases #3 and #6 (related to the test frames #3 and #6 in Table 2) detected many faults in the code faulty version V1. These errors are due to the fact that, when a call request R is rescheduled from one elevator (A) to the other (B), while elevator B receives the AddCall(R) event, both elevators receive the RemoveCall(R) event. Therefore, since elevator B received a RemoveCall(R) just after the AddCall(R), it does not attend call R, which is left unattended by the elevator system.

More importantly, SA level test case #4 (related to the test frame #4 in Table 2) detected a “real” error in both the original and faulty version. Basically what happens is that although elevator ADT1 was supposed to receive the AddCall event, elevator ADT0 is the one actually receiving it.

SA test case #	**# of Code level test cases**	**V1 Faulty Implementation**	**V2 Original Implementation**
1	7	No faults detected	No faults detected
2	8	No faults detected	No faults detected
3	6	**5 faults detected**	No faults detected
4	7	**1 fault detected**	**1 fault detected**
5	8	No faults detected	No faults detected
6	6	**5 faults detected**	No faults detected

Table 4: Results

In order to produce unbiased results, we performed this evaluation separately: one of the authors produced the SA specification of the system and the SA-level test cases while the other implemented the system, refined the SA test cases into code level test cases and run the test cases.

The first consideration to be done is that C2 is used in this paper as representative of all such frameworks which support a code generation process. The interest of this research, in fact, is to analyze how a generic framework, supporting the code generation, may help to make systematic the testing process. Moreover, the assumption that the code generation process is driven by a superimposed framework is not to be considered too restrictive. Recent research is investigating how ADLs can support the generation of executable code [20, 25]. In particular, in [25] the authors analyze how ADLs are evolving in order to bridge the gap between a software architecture specification and its implementation. Both Monroe [21] and Garlan [15] point out how skeletal code automatic generation may reduce implementation time. Moreover, Aesop, C2, and Darwin generate skeletal code in C/C++ and Rapide executes the design code internally. Furthermore, MetaH is supported at the implementation level by Ada, and ArchJava. For more details, please refer to [25].

It may be of interest, for future research, to analyze how this experience may be reused in contexts different from C2. We expect that when the code generation process is systematic, the testing process may be performed systematically too.

Some problems we initially found were how to run the code and deterministically analyze the code execution and how to identify parameters. Thanks to the Argus-I tool we overcame the first problem. Following the category partition method, we easily learned how parameters may be identified.

7 Related Work

In this section we briefly present important research areas related to our approach.

The topic of ***specification-based conformance testing*** has been extensively analyzed by many authors [8, 30, 14] , as already pointed out in the Introduction. Comparing our approach with their, we can notice that we also use the SA-derived LTS as a reference model to derive test cases. However, all such approaches produce a model of the implementation under test and define some implementation relations (conf, ioconf, ioco, etc.) between code and specification. In our case, we do not assume to be able to produce an LTS model of the implementation thus we compare architecture level sequence of events with lower level execution paths.

The topic of ***SA-based testing*** has been already discussed in Section 2. Again, the main difference between our and other approaches is the challenge to consider the whole cycle of SA-based testing, from architecture specification to test execution over system implementation.

The difficulty of ***tracing*** information is not new, as already recognized in [12, 32, 27]. Some relevant papers have been written on the topic. We can here mention reference [13] which shows a way to detect traceability between software systems and their model and proposes a list of interesting references on traceability techniques. Some work has been done in bridging the gap between requirements and software architectures (e.g., [28]), and much other work addresses requirements traceability. The problem of mapping abstract tests into the System Under Test is under study in the ongoing AGEDIS project [1].

8 Conclusions and Future Work

In this paper we refine our previous experience [6, 7] on SA-based testing. While our previous papers were dealing with a generic architecture in a generic software development process (without assuming any relationship between SA and code), we here make a stricter assumption on the software development process, in order to make more formal the full testing process.

In particular, by adopting the C2 style architecture and the related C2 framework, we are able to systematically implement the *test execution stage* described in step 4, handling both traceability/mapping among SA and code execution over the identified test cases problems. By using C2, in fact, traceability is explicitly maintained between the architectural events and the code-level sequences while code execution is allowed through the Argus-I tool.

In future work we want to investigate how a similar approach can be applied to those other ADLs which support code generation. Moreover, we want to apply this approach for SA-based regression testing, in the SARTE project [22].

In the long term, we plan to use the experience gained in this paper in order to relief some constraints. Our desire is to be able to specify and test architectures in a generic ADL (assuming that a behavioral model can still be produced). In particular, we would like to take a generic architecture described using the XADL ADL [11], providing a behavioral description in the form of state-based machine model, to implement this architecture using a component-based technology through a middleware (Java/RMI, COM+ of CORBA) and test the system implementation.

Acknowledgments

The authors gratefully acknowledge the Italian MIUR national project SAHARA and the National Science Foundation, both of which partially supported this work.

References

1. AGEDIS Project. Automated Generation and Execution of Test Suites for Distributed Component-based Software. On-line at: *http://www.agedis.de/index.shtml*.
2. The Argus-I project. University of California, Irvine. Information on-line at *http://www.ics.uci.edu/~mdias/ research/ArgusI*.
3. G. Bernot, M. C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. Software Engineering Journal, Vol. 6, N. 6, pp. 387-405, 1991.
4. A. Bertolino and P. Inverardi. Architecture-based software testing. In *Proc. ISAW96*, October 1996.
5. A. Bertolino. Knowledge Area Description of Software Testing. In *SWEBOK*: The Guide to the Software Engineering Body of Knowledge, Joint IEEE-ACM Soft. Eng.
6. A. Bertolino, P. Inverardi, and H. Muccini. An Explorative Journey from Architectural Tests Definition downto Code Tets Execution. In IEEE Proc. Int. Conf. on Software Engineering, ICSE2001, pp. 211-220, May 2001.

7. A. Bertolino, P. Inverardi, H. Muccini. Formal Methods in Testing Software Architectures. Chapter in Formal Methods for Software Architectures, SFM-03: SA Lectures, Eds. M. Bernardo, P. Inverardi, LNCS 2804, 2003, p. 124-149.
8. G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In ACM Proc. Int. Symposium on Software Testing and Analysis, ISSTA '94, pp. 109-124, 1994.
9. 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.
10. The C2 Architectural Style. On-line at: *http://www.ics.uci.edu/pub/arch/c2.html*.
11. E. Dashofy, A. van der Hoek, and R. N. Taylor An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In Proceedings of the ICSE 2002 International Conference on Software Engineering (ICSE 2002), Orlando, Florida, May 2002.
12. 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-Strenght Formal Methods*, pp. 268-284. *LNCS 670*, 1993.
13. A. Egyed. A Scenario-Driven Approach to Traceability. IEEE Proc. Int. Conf. on Software Engineering (ICSE2001), pp. 123-132, May 2001.
14. J.-C. Fernandez, C. Jard, T. Jeron, L. Nedelka, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In *Proc. of the Eighth Int. Conf. on Computer Aided Verification* (CAV'96), USA, pp. 348-359, 1996. *LNCS 1102*, Springer, 1996.
15. D. Garlan. "Software Architecture", *Encyclopedia of Software Engineering*, John Wiley & Sons Inc., 2001.
16. M. J. Harrold. Testing: A Roadmap. In A. Finkelstein (Ed.), *ACM ICSE 2000, The Future of Software Engineering*, pp. 61-72, 2000.
17. Labelled Transition System Analyzer (LTSA). On-line at: *<http://www-dse.doc.ic.ac.uk/~jnm/book/>*.
18. J. Magee, and J. Kramer. *Concurrency: State models & java programs.* Wiley publisher, April 1999.
19. N. Medvidovic. Formal definition of the Chiron-2 software architectural style. Technical Report UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, August 1995.
20. N. Medvidovic, and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering, vol. 26, no. 1 (January 2000).
21. R. T. Monroe, Rapid Development of Custom Software Architecture Design *Environments*, Ph.D. Thesis, Carnegie Mellon University, 1999.
22. H. Muccini, M. Dias, and D. Richardson. Software Architecture-based Conformance and Regression Testing - documents. On-line at: *http://www.HenryMuccini.com/Research/ ICSE04_Submitted.htm*.
23. H. Muccini, M. Dias, and D. Richardson. Systematic Testing of Software Architectures in the C2 style. Extended version of the ETAPS 2004 publication. On-line at: *http://www.HenryMuccini.com/Research/ETAPS04.htm*.
24. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, Vol. 31, N. 6, pp. 676-686, June 1988.
25. O. Papapetrou, A. Stavrou and G. A. Papadopoulos. From Software Architectures to Coordination Programming: Bridging the Gap Between Architecture Specification and System Implementation. Submitted for publication.

26. 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.
27. J. Richardson, J. Stafford, and A. L. Wolf. A Formal Approach to Architecture-based Software Testing. Technical Report, University of California, Irvine, 1998.
28. Straw 2001. First Int. Workshop “From Software Requirements to Architectures” (STRAW'01), May 14, 2001, Toronto, Canada.
29. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. In IEEE Transactions on Software Engineering, June 1996.
30. J. Tretmans. Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, Vol.29, pp. 49-79, 1996.
31. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* NIST Special Publication 500-235, August 1996.
32. M. Young. Testing Complex Architectural Conformance Relations. In *Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis* (ROSATEA), CNR-NSF, pp. 42-45, July 1998.