International Journal of Software Engineering
and Knowledge Engineering
Vol. 30, No. 2 (2020) 191–216
© World Scientific Publishing Company
DOI: 10.1142/S0218194020500096

Case Study Investigation of the Fault Detection and Error Locating Effects of Architecture-based Software Testing

Jihyun Lee
Department of Software Engineering
Jeonbuk National University
567 Baekje-daero, Deokjin-gu
Jeonju-si, Korea
jihyun30@jbnu.ac.kr

Sungwon Kang*
School of Computing, KAIST
291 Daehak-ro, Yuseong-gu, Daejeon, Korea
sungwon.kang@kaist.ac.kr

Received 11 January 2019
Revised 27 February 2019
Accepted 10 July 2019

For software testing, it is well known that the architecture of a software system can be utilized to enhance testability, fault detection and error locating. However, how much and what effects architecture-based software testing has on software testing have been rarely studied. Thus, this paper undertakes case study investigation of the effects of architecture-based software testing specifically with respect to fault detection and error locating. Through comparing the outcomes with the conventional testing approaches that are not based on test architectures, we confirm the effectiveness of architecture-based software testing with respect to fault detection and error locating. The case studies show that using test architecture can improve fault detection rate by 44.1%–88.5% and reduce error locating time by 3%–65.2%, compared to the conventional testing that does not rely on test architecture. With regard to error locating, the scope of relevant components or statements was narrowed by leveraging test architecture for approximately 77% of the detected faults. We also show that architecture-based testing could provide a means of defining an exact oracle or oracles with range values. This study shows by way of case studies the extent to which architecture-based software testing can facilitate detecting certain types of faults and locating the errors that cause such faults. In addition, we discuss the contributing factors of architecture-based software testing which enable such enhancement in fault detection and error locating.

Keywords: Software product line testing; architecture-based software testing; points of observation; points of control and observation.

*Corresponding author.

1. Introduction

Currently, the typical IT/R&D project budget assigned to testing ranges from 11% to 40% of the total budget [1, 2]. The World Quality Report noted that the average expenditure for quality assurance and testing was 26% of the total IT amount spent in 2018 [3]. Testing needs to be more efficient and requires research for better management of the scale and complexity of software systems.

Software architecture refers to a set of important early design decisions for the development of software systems. It provides guidelines and determines the important constraints on development to ensure a high level of software quality, thereby improving productivity, reusability, and maintainability during the software development process [4, 5]. As software architecture represents a principled approach to cope with the large-scale structures of software systems [6], software testing also requires an approach to deal with the scale and complexity of software systems. As software architecture can have advantages for those involved in software testing because the code is developed on the basis of software architecture, we expect to exploit software architecture for the testing and design specialized structures, i.e. test architecture capable of addressing various testing challenges. This expectation has been shared by many researchers [7, 8]. As stated in Shaw *et al.* [7], "Rethinking our approach to software testing on the basis of software architecture is an important research direction in software testing." This appraisal has been reemphasized in other work [6].

Architecture-based software testing is "a testing approach that relies on test architecture by employing architecture views such as logical views, module views and execution views in order to control and observe the interaction points at which the interactions between components or modules occur" [12]. In contrast with the testing approaches that are not architecture-based software testing uses application architecture to design test architecture by placing control and observation points at various architecturally important locations as defined by the application architecture and utilize them for detection faults at the locations close to where the errors causing them reside so that the effort for error locating is minimized.

Test architecture concepts were defined and used in distributed systems for conformance and interoperability testing of the systems [14, 30, 31]. Controllable distributed testing [14] has a similar purpose as architecture-based software testing has but is different from it in that it considers only one particular architecture view, i.e. the distribution architecture, and utilizes for observation and control the interaction points between distributed components, disregarding the possibility of observing and controlling the interaction points of components internal to distributed systems and utilizing various other architecture views such as logical architecture, module architecture, execution architecture, etc. Keum *et al.* [13] apply such concepts to the testing of service-oriented architecture (SOA) software, which is noted for complex interactions among services in a distributed systems setting. Although they used architecture-based software testing, they differ in that they failed to provide a framework of architecture-based software testing by considering only one particular architecture view, i.e. the distribution architecture, and limiting observation and control to the interactions between distributed components. This contrasts with our previous work in [12], which generalizes architecture-based software testing to observe and control the interactions of components that are internal to distributed systems and to utilize various other architecture views such as logical architecture, module architecture, and execution architecture. There are studies [8, 9, 17] that use the Testing and Test Control Notation (TTCN-3) for software system testing based on test architecture concepts. Such studies indicate that test architectures can be an important cornerstone of software testing and can offer significant benefits, but they discuss the potential possibilities of test architecture at the conceptual level.

Test architecture concepts for software testing have been developed by Lee *et al.* [12]. However, the nature and extent of the effects architecture-based software testing offers during software testing has not been sufficiently investigated yet.[a] Among the benefits that architecture-based software testing can offer from various aspects such as productivity, reusability, and maintainability for software testing, its main benefit is that with it we can enhance the fault detection ability of testing in a cost-effective manner. Thus, our investigation in this paper focuses on validating the effectiveness of architecture-based software testing with respect to fault detection and error locating[b] and compares the outcome with the conventional testing approaches that do not rely on test architectures. We chose the empirical study of Mouchawrab *et al.* [15, 16] for comparison because it provides experiment results about fault detection effects of the state machine-based testing, the structural testing, and a testing approach that combines the two techniques. The main goals of this study are (1) to validate the fault detection and error locating effectiveness of architecture-based software testing by comparing it with the empirical evidence presented in these earlier reports; and (2) to find the contributing factors of architecture-based software testing that enable such enhancement in fault detection and error locating.

This paper is organized as follows: Sec. 2 provides the motivation behind the study with a brief description of test architecture concepts; Secs. 3 and 4 conduct two case studies and describe answers to three research questions. Section 5 discusses the results; Sec. 6 presents the existing research related to the concepts of test architecture and architecture-based software testing; finally, in Sec. 7 we offer concluding remarks and suggest future work.

2. Revisiting Architecture-based Software Testing for Motivation and Background

This section presents the motivation behind this study and the background knowledge of architecture-based software testing. In Sec. 2.1, the concepts and definitions of

[a] Only Keum *et al.* [13] show the effects of test architecture during fault detection and error locating. However, it is not readily applicable to systems which do not have SOA.
[b] Error locating refers to the finding of approximate locations of errors that cause a fault.

architecture-based software testing are briefly explained. In Sec. 2.2, the motivation for test architecture is explained with a vending machine application devised to explain briefly the fault detection capability of our method.

2.1. *Architecture-based software testing*

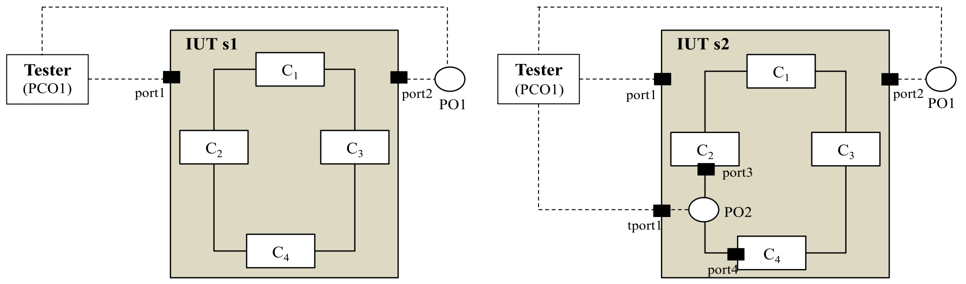
Architecture-based software testing [12][c] is a testing approach that uses test architecture obtained from software architecture that is captured by architectural views, such as logical views, module views and execution views, in order to observe and control the interaction points at which the interactions between components or modules occur. Point of observation (PO), which observes the behavior of the implementation under test (IUT) and point of control and observation (PCO), which observes and controls the behavior of IUT for observation are test elements and test architecture for a given software system is software architecture of the system augmented with a set of such test elements together with the relationships between the system components and the test elements and the relationships between the test elements [12]. A tester is a special PCO that contains a set of test cases, executes test cases, produces pass or fail verdicts based on test oracles, and controls and coordinates POs and other PCOs. By comparing the expected responses and the actual responses from the components, the tester makes the final decision of the correctness of the IUT with respect to a given test case.

A test architecture is formally represented as a 4-tuple $\langle S, T, P, C\rangle$, where S is an IUT, T is the tester, P is a set of POs and C is a set of PCOs. In the test architecture of Fig. 5(b), S is $\{S2\}$, T is {Tester}, P is {PO1, PO2} and C is {PCO1}. A PO is formally represented as a triple $\langle L, R, M\rangle$, where L is the location of the PO, R is a non-empty set of responses and M is a set of ports for the PO. A PCO is formally represented as a quadruple $\langle L, S, R, M\rangle$, where L is the location of the PCO, S is a non-empty set of stimulus, R is a non-empty set of responses and M is a set of ports for the PCO. The syntax 'PO1? *msg*' denotes that a response '*msg*' is received at PO1 and the syntax 'PCO1! *msg*' represents that a stimulus '*msg*' is sent out for control from PCO1.

Test architecture can be classified as IUT-independent test architecture or IUT integrated test architecture depending on whether POs and PCOs are placed only at locations external to the IUT or both at internal locations within the IUT as well as at external locations. Figure 1 illustrates these two types of test architectures. The IUT-independent test architecture shown in Fig. 1(a) has one PCO, i.e. the Tester and one PO1. The IUT integrated test architecture in Fig. 1(b) has one PCO and two POs, of which PO2 is placed within the IUT to allow the tester to observe and/or control the internal behavior of the IUT.

In Fig. 1, port1 through port4 are the ports of IUT whereas tport1 is a test port that is added to connect the Tester through the IUT to PO2. Figure 1(b) shows port3

[c]Lee *et al.* [12] provides the detailed definition of test architecture. In order to make this paper self-contained, many descriptions in the subsection are reused from [12].



(a) IUT-independent test architecture example (b) IUT integrated test architecture example

Fig. 1. Test architectures examples.

and port4 as the ports of C2 and C4, respectively. They are not shown in Fig. 1(a), however, because they are not visible to the Tester. A test architecture is realized by implementing POs, PCOs (including the tester) and interaction mechanisms for them.

In the case of IUT-independent test architecture, POs/PCOs can be implemented using message queues or test components with ports, for example, while IUT integrated test architecture can be implemented using code instrumentation such as test probes, assertions and specialized testing interfaces. Tracking interface, built-in tracking code and systematic component wrapping for testing are yet other possible implementation techniques for POs/PCOs. For example, for the test architecture of Fig. 1(a), PCO1 can be implemented with a test component with one input port and one output port. PCO1 in Fig. 1(a) stimulates the IUT through its input ports, receives the stimulation result through its output ports and determines pass or fail by comparing the result with the expected result. Likewise, a PO receives the observation result through its output ports.

In the case of IUT integrated test architecture, POs/PCOs can be implemented with separate modules or classes referring to the IUT that includes assertions, console or file input/output statements, or accessors. For example, for the test architecture of Fig. 1(b), built-in testing interface (i.e. Set-To-State operation) and built-in tracking code (i.e. Is-In-State operation) can be used for implementing PCO1.

An interaction mechanism can be implemented using built-in test interface, built-in test code, or mappings and connections for sending stimuli and receiving responses. For the details for test architecture design, implementation and test derivation based on test architecture, the reader can refer to Lee *et al.* [12] In order to apply architecture-based software testing method, architectural specification for IUT should be available. Then test engineers can decide the location and numbers of POs/PCOs based on the architectural specification. Also test engineers should be provided with the environment in which they can perform code instrumentation to add necessary implementations for the POs/PCOs.

Any test case generation method can be turned into an architecture-based test generation method. For example, if a state machine-based test generation method is used, its architecture-based version would just generate test cases in the same way but with the difference that, in addition to the input values and expected results for the application, as the input values to the relevant PCOs and the expected results from the relevant POs should also be determined as required by the defined test architecture. Thus, test case description for architecture-based testing should include relevant POs/ PCOs of the test architecture. For example, in the cruise control system in Sec. 3.3, there is a test case derived from a state machine consists of a sequence of events, which are 'engineOn', 'on' and 'accelerator'. A PO 'throttle' in the test architecture is a PO that is related to the response of each event. In architecture-based testing using a state machine, a test case may be described using the TTCN-3 test description style as follows:

```
Tester! engineOn
   PO? throttle
      Tester! on
         PO? throttle
            Tester! accelerator
               PO? throttle
```

The syntax 'Tester! engineOn' represents that 'engineOn' event is sent out and the syntax 'PO? throttle' represents that the value of 'throttle' is observed at PO.

2.2. *Motivating example*

Figure 2 shows the architecture of a vending machine application to be used in this section. In this example, the 'Vending Machine' component of the vending machine application controls operation of the vending machine; the 'Display' component shows the status of the vending machine; and the 'Dispenser' component dispenses items and change.

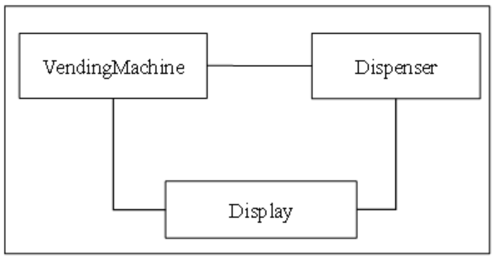


Fig. 2. Architecture of a vending machine application.

When these components are integrated, there may be errors that cannot be detected by the conventional testing. For example, consider the following test case TC1:

```
Tester! insertMoney(Amount)
   Tester! itemOrder(Req_Item)
      Tester?Req_Item
```

After the Tester sends a 'insertMoney' message with the 'Amount' number of coins and sends the 'itemOrder' message with the ordering item 'Req_Item', the 'Vending Machine' component invokes the 'itemDispense (Req_Item)' method of the 'Dispenser' component, which dispenses the 'Req_Item' and decreases the number of items. After dispensing an item, the 'Dispenser' checks whether there are any items remaining. Meanwhile the Tester waits for 'Req Item'.

As shown in Fig. 3, when there are no remaining items, the 'Dispenser' component should invoke the 'displayEmpty (Req_Item)' method of the 'Display' component to display an "empty" sign. If the 'Dispenser' does not invoke the 'displayEmpty (Req_Item)' method, it is an error but the Vending Machine component may not detect it because it does not observe the properties of the 'Dispenser' component. The number of items in the 'Dispenser' component is decreased whenever an item request is made from the 'Vending Machine' component, and whether this behavior includes defects can be confirmed after all three components have been integrated. It may be different depending on the order of integration, but let us consider the case when they are integrated in the order of the 'Vending Machine' component, the 'Dispenser' component and the 'Display' component. Even though an interaction between the 'Dispenser' component and the 'Display' component mentioned above has been caused by the 'Vending Machine' component it is not visible to the 'Vending Machine' component. This defect may not be found unless the system state is observed by a test case that is defined in accordance with the described conditions. If we recognize the existence of unobservable interactions, faults caused by such errors may be easily detected. Referring to the architectural specification of the Vending Machine System in Figs. 1 and 2, we may recognize such unobservable interactions.

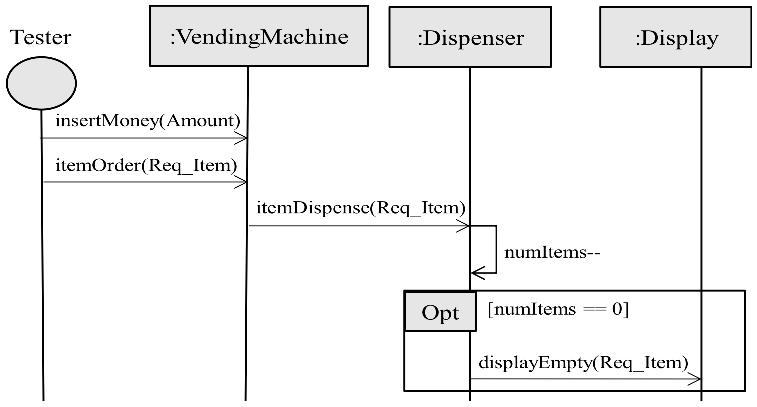
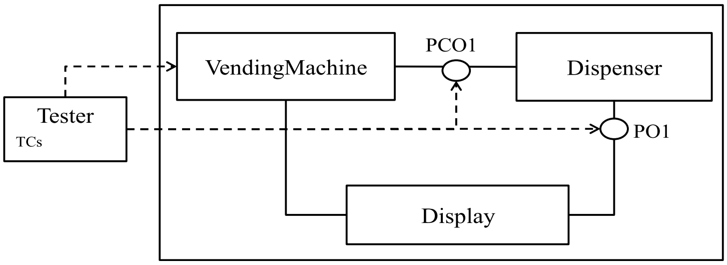
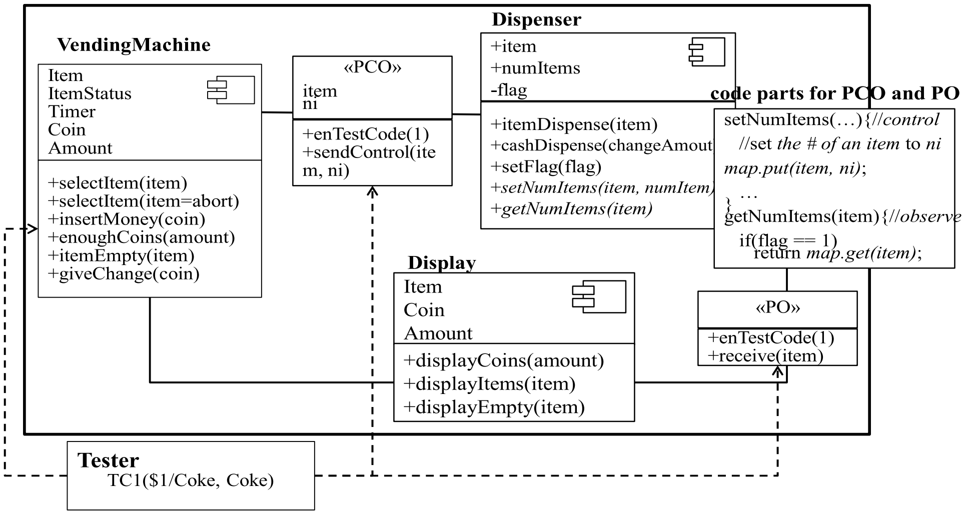


Fig. 3. A vending machine operation when TC1 executes.

As depicted in Fig. 4(a), we can detect the faults if a PO1 or PCO1 is inserted into the vending machine application between the 'Dispenser' and the 'Display' components, where interactions could not be observed without a PO or PCO. Figure 4(b)



(a) A test architecture for the vending machine



```
setNumItems(...){//control
  //set the # of an item to ni
map.put(item, ni);
} ...
getNumItems(item){//observe
  if(flag == 1)
    return map.get(item);
```

(b) Implementation of the test architecture in (a)

Fig. 4. IUT integrated test architecture–vending machine application.

shows PO1 that is instrumented its test code into the ‘Display’ component. PO1 provides an interface that enables test codes (enTestCode()) and allows observation of the interactions between the ‘Dispenser’ and the ‘Display’ components. As ‘Tester’ is a specially designated PCO that orchestrates the overall testing process, it can check through PO1 whether the ‘Dispenser’ component correctly decreases the value of the ‘numItems’ variable while executing TC1.

However, if the number of items exceeds 1, the ‘displayEmpty()’ method need not be invoked. In the presence of a PCO such limitation can be overcome. Figure 4(b) shows an example PCO, PCO1, which sends a message to control the value of ‘numItems’ and observes the result through PO1. The Tester sets the current number of items to ‘1’ through PCO1. By executing TC1, it can detect this error through PO1.

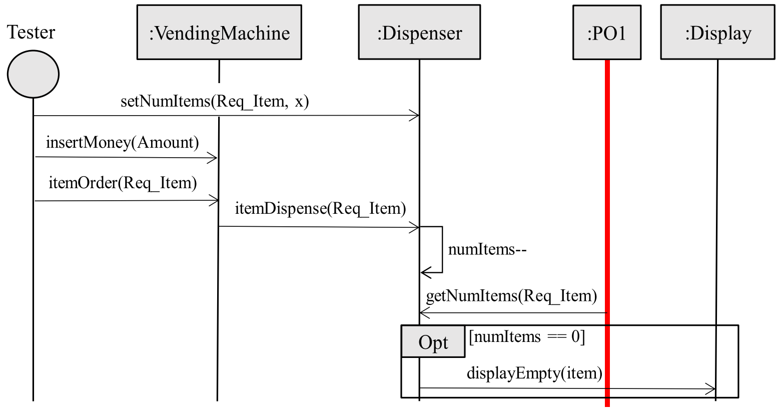


Fig. 5. A vending machine operation when TC2 executes.

Figure 5 shows the interaction of the vending machine when the following test case, TC2:

```
Tester! setNumItems(item, numItem)
   Tester! insertMoney(Amount)
      Tester! itemOrder(Req_Item)
         Tester? Req_Item
            PO1? getNumItems(Req_Item)
```

is executed under the test architecture of Fig. 4. In TC2, the Tester sends the 'setNumItems' message to set the number of items 'Req_Item' to '1'; from the 2nd through the 4th lines of TC2 are similar to those of TC1; and PO1 waits for 'getNumItems' to observe the value of 'Req_Item'. Unlike TC1, which is defined without test architecture, TC2 includes test elements for controlling and observing the intermediate states.

Even from this simple example we can clearly see that test architecture defined by adding POs/PCOs to the application architecture and the test code implementation based on the defined test architecture enable efficient fault detection and error locating. Architecture-based software testing was instrumental, when compared with the conventional testing, in achieving the specific effects in Table 1 in the case of this motivating example. In the remainder of this paper, we investigate to what extent

Table 1. Summary of differences between architecture-based software testing and conventional testing in the motivation example.

	Architecture-based software testing	Conventional testing
Testability	Testability between 'Dispenser' and 'Display' components is ensured even after the components are integrated	Testability between 'Dispenser' and 'Display' is not guaranteed after integration

Table 1. (*Continued*)

	Architecture-based software testing	Conventional testing
Fault detection capability	Testing does not miss faults on the interactions between 'Dispenser' and 'Display' components that remained after the system integration because focusing on the unobservable interaction improves observability	Careful design of test cases can detect faults on the unobservable interactions between 'Dispenser' and 'Display' components that testing did not find before the system integration, but it may still easily miss them
Error locating capability	A PO or PCO is used as information for error locating, so only the statements containing 'numItems', which is used as a PCO, and the statements involved in unobservable interactions with it become suspicious statements which may contain error	If 'Empty' sign is not turned on, all statements in Fig. 5 that the test case executes become suspicious statements which may contain error

and in what ways such benefits of architecture-based software testing over the conventional testing are achieved through case studies.

3. Design of Case Study Investigation

3.1. *Research questions*

In our case study investigation, we undertake a quantitative analysis of differences in terms of fault detection and error location effectiveness and also a qualitative analysis in order to understand the reasons for these differences. Through this investigation, we aim to obtain answers for the following research questions:

- RQ1: Does architecture-based software testing improve fault detection capability beyond the testing without architecture?
- RQ2: Can faults be located efficiently when architecture-based software testing is used?
- RQ3: Can oracles be generated with minimal efforts when architecture-based software testing is used?

3.2. *System selection*

Among many systems for which their source code is provided and their case studies report detailed experimental results and reproducible outcomes, we selected the systems that were used by Mouchawrab *et al.* [15, 16] because their experiments with the systems included fault detection experiments that compare the state machine-based testing, structural testing and combinations of the two approaches. The systems are the cruise control system and the elevator system from Software Infrastructure Repository (SIR) [18].

The cruise control system is a system that simulates a car engine and its cruise controller and the elevator system is a system for servicing stop requests to move

Table 2. Overview of the systems used for case studies [15].

Aspect \ System	Cruise control system	Elevator system
# classes	4	8
# operations	34	74
# attributes	14	37
# LOC	358	581
# transitions	17	50
# states	5	6
# events	7	10

passengers to other floors in an elevator and up or down requests from floors to move passengers from one floor to another. Table 2 provides the details of the two target systems. As shown in Table 2, the two systems have suitable sizes for controlled case studies.

Mouchawrab *et al.* [15, 16] report in detail the experimental results from a series of controlled experiments including both black-box and white-box testing with concrete experiment inputs. Hence, comparing architecture-based software testing, which is a type of gray-box testing, with the experiment results of Mouchawrab *et al.* [15, 16] will be illuminating.

3.3. *Plans for case studies*

With the selected systems, we compare the fault detection capabilities of the state machine-based testing, the structural testing, and one that combines the state machine-based testing and the structural testing. Our comparative experiments just use the experiment results of Mouchawrab *et al.* [15, 16] without reproducing their experiments here because we will apply architecture-based software testing to the state machine-based testing of Mouchawrab *et al.* [15, 16] with no modification.

For each research question of RQ1–RQ3 we devised the following plans to obtain answers:

- *Plan for selecting and inserting POs*: After identifying unobservable interactions based on architectural specifications, we manually select the variables to observe among the variables participating in interactions as POs. The number of POs depends on the size and complexity of unobservable interactions. The determined POs are implemented in the form of test code and inserted into IUT.
- *Plan for RQ*1: We investigate this question by comparing in detail the fault detection rates of the state machine-based testing with and without using architecture-based software testing. We then compare the results pertaining to fault detection with those of the structural testing and the testing that combines the state machine-based testing and the structural testing. State machine-based testing has been widely used as a means of integration testing [16]. It is well known that detecting faults related to certain specific parameters and unspecified or

sneak paths is challenging. Given the evaluation plan, we define the following two sub-questions for our evaluation:

— RQ1_1: Does architecture-based software testing improve fault detection capability for faults requiring specific parameter values?

$$\text{FaultDetectionEfficiency}(\%) = \frac{\text{Number of faults detected}}{\text{Total number of faults}}$$

— RQ1_2: Does architecture-based software testing improve fault detection capability when faults are related to unspecified paths or sneak paths?

- *Plan for RQ*2: We investigate this question by checking whether we can pinpoint suspicious components or code regions.

$$\text{ErrorLocatingEfficiency}(\%) = \frac{\text{Number of errors directly located by POs}}{\text{Number of faults detected}}$$

- *Plan for RQ*3: We investigate how we can precisely define oracles for test cases in architecture-based software testing. Oracles of architecture-based software testing are the values to be compared with those observed at the POs of test architecture. Because both systems selected for the case study include multiple threads, we check that architecture-based software testing incurs little effort for oracle decisions, even for the systems that execute with threads.

Table 3 summarizes the test environments of our case study investigation. We inserted faults into the two systems. In the cruise control system, we automatically

Table 3. Test environments of the testing of Mouchawrab *et al.* [15, 16] and the architecture-based software testing.

Approach / Aspect	Mouchawrab *et al.* [15, 16]	Architecture-based software testing
Basis for test case generation	State-machine diagram, including a transition tree Structure of source code	State-machine diagram, including a transition tree Use case diagram or sequence diagram at the architecture level for selecting POs
Test coverage	RTP Extending the RTP criterion	RTP Test coverage at test architecture [12]
Test data	Test coverage-based test data generation	Test coverage-based test data generation
Oracles	State invariants, operations' contracts	State invariants Expected values of the selected POs
Fault seeding	Automatically seeded using MuJava	Automatically seeded using MuJava only for four mutation operators
Data collection	Perl script to automatically execute test driver to inspect the correctness of the system	Tester to execute test cases and collect test results to check oracles
Effectiveness of approaches	Mutation score for each mutant	Mutation scores of four selected mutants The number of errors that are located and narrowing down the scope of the review

generated mutants using MuJava [19, 20] in the same manner as in Mouchawrab *et al.* [15, 16], whereas in the elevator system we used source code including mutants provided in SIR [18] without modification. The test cases for the two systems were derived in the same manner as in Mouchawrab *et al.* [15, 16] so they were generated based on state-machine diagrams, including class public interfaces, transition trees, class diagrams, operation contracts and state invariants. We also used all possible paths of the state machines with the Round-Trip Path (RTP) coverage [21].

According to Mouchawrab *et al.* [15, 16], in their experiment results the authors could not detect faults on unary operators, relational and logical operators when they tested the selected systems using the state machine-based testing. In our case study investigation, we check whether architecture-based software testing based on the same state machines can enhance faults detection capability compared to the state machine-based testing for those kinds of faults. We also compare the capability of architecture-based software testing based on state machines for those faults with the testing that combines state machine-based testing and structural testing, which is the one that showed the best fault detection rate in the experiments of Mouchawrab *et al.* [15, 16]

To that end, we seeded faults using the automatic tool MuJava for the following four mutation operators: Arithmetic Operator Insertion-Unary (AOIU) for the mutations that negate numeric variables; Arithmetic Operator Insertion-Short-cut (AOIS) for the mutations that insert the increment (++) or decrement (−−) operators into numeric variable uses; Relational Operator Replacement (ROR) for the mutations that replace binary Boolean operators; and Logical Operator Insertion (LOI) operator. The LOI operator is the mutation that computes the 1's complement of numbers by replacing X with $\sim X$. We selected these four mutation operators because they are the mutants used in the experiments of Mouchawrab *et al.* [15, 16] and the authors reported that their approach could not detect them well.

4. Case Studies

In this section, architecture-based software testing is applied to two selected systems for an evaluation of its effects on fault detection and error locating processes. Case studies are conducted in accordance with the process of architecture-based software testing defined in [12]. The major steps of architecture-based software testing process are (1) design test architecture, (2) design test cases, (3) implement test architecture and (4) execute test. This section describes the execution of the process and the resulting artifacts of the case studies, respectively in Secs. 4.1 and 4.2. We implemented and used the tester and test elements defined in the test architecture for test execution and evaluation of test results.

4.1. *Test architecture and test cases design*

Test architectures for the case study systems were designed based on their module view architectures and behavioral view architectures. We depicted the test architectures using the diagrams provided by SIR [18] and Mouchawrab *et al.* [15, 16]. In the case of the cruise control system, the sequence diagram was used to describe most frequently occurring interactions in order to identify POs. We generated test cases based on the test architectures and the state-machine diagrams for the systems. In this subsection, we describe test architecture for the selected system and test cases generated based on it.

4.1.1. *Test architecture and test cases for the cruise control system*

The cruise control system has unobservable interactions between the 'Controller', 'SpeedControl' and 'CarSimulator' modules, as shown in the module view architecture of bold square part of Fig. 7. In this system, the values of the variables change in real time, making it difficult to ascertain their exact values. From the state-machine diagram of Mouchawrab *et al.* [15, 16] and the module view architecture of the cruise control system, we derived four invariant variables 'state_of_speed_control', 'setSpeed', 'ignition' and 'distance'. They are variables related to unobservable interactions, and therefore we chose them as POs.

We derived one PO for the variable 'throttle' from the architecture specification shown in Fig. 6 because the variable 'throttle' was related to the most frequent interactions between 'Controller' and 'SpeedControl', where both observable and unobservable interactions existed. However, we could not know the exact value of throttle because it executes as a thread. Instead we could find the range of throttle values from the architecture specification.

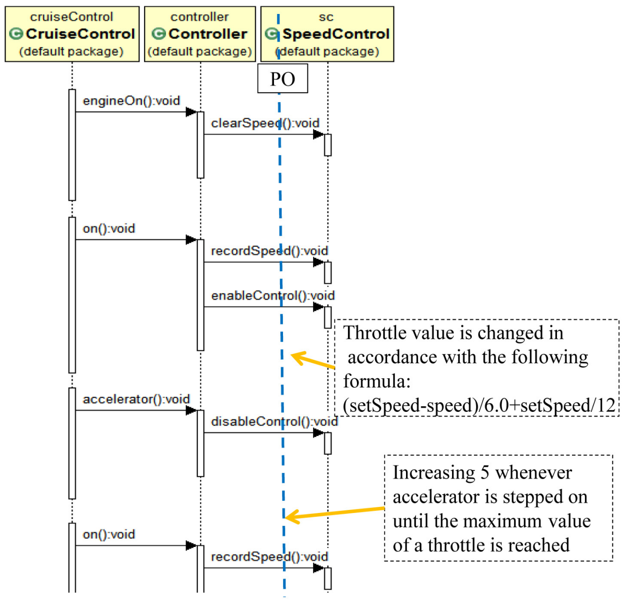


Fig. 6. Selecting a PO for the cruise control system.

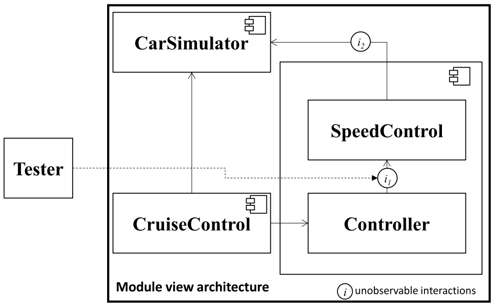


Fig. 7. Test architecture of the cruise control system.

Thus, five POs were inserted to i_1, where interactions between the SpeedControl and Controller. And a tester for executing test cases, monitoring and judging test results. Figure 7 is a test architecture for the cruise control system.

As for test cases for the cruise control system, we designed 12 test cases with the same method of Mouchawrab *et al.* [15, 16], i.e. using state-machine diagram with the RTP coverage.

4.1.2. *Test architecture and test cases for the elevator system*

The experiments on the elevator system in Mouchawrab *et al.* [15, 16] established the numbers of elevators and floors in the cluster, the types of requests and the floors or elevators from which to send the requests in order to ensure the covering of a specific path in the transition tree in the elevator system. The elevator system has many unobservable interactions from the 'FloorInterface' and 'ElevatorInterface' modules, as shown in the module view architecture of the elevator system of bold square part of Fig. 8. The elevator system contains threads, making it difficult to observe the values of the observation points.

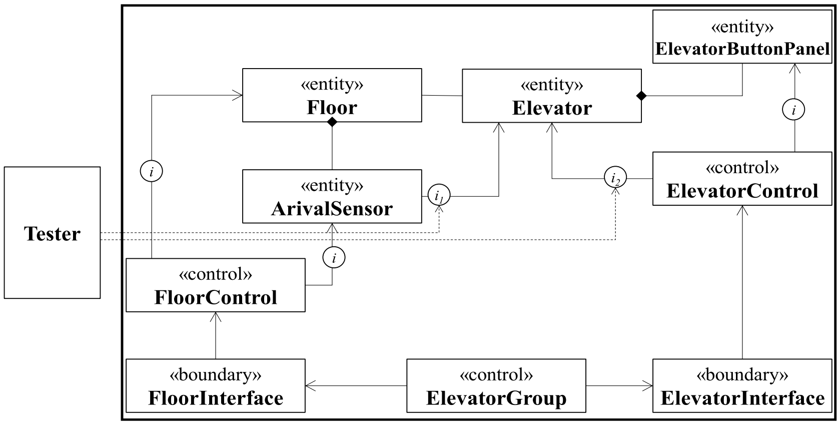


Fig. 8. Test architecture of the elevator system.

From the state-machine diagram of Mouchawrab *et al.* [15, 16] and the module view architecture of the elevator system, we derived four POs 'state_of_elevator', 'doorOpen', 'motorMoving' and 'direction', which were state invariants of a state-machine diagram. In addition to the four POs for state invariants, we selected four POs, 'elevatorID', 'floorID', 'bestElevatorID' and 'thread status' and observed their values. As a result, 8 POs were inserted into position i_1 and i_2 as shown test architecture of the elevator system in Fig. 8. We did not need to implement additional code to observe the selected POs because the elevator system already has code that checks their values.

For the elevator system, we designed 40 test cases from the same state-machine diagram of Mouchawrab *et al.* [15, 16] and with the same method in accordance with the RTP coverage.

4.2. *Results of case studies*

Test cases of the state machine-based testing are obtained typically using the paths of the state machine. To check a state of an object oracle can be based on the values of all attributes or the state invariants of the object in accordance with the events [15]. In the case of architecture-based software testing, oracles should also be determined for POs. In our case studies, we obtained oracles for values/ranges of POs from architectural specifications and also from executing all test cases on the target systems before mutants are injected. To quantify the effects of architecture-based software testing in locating errors, we counted the number of detected faults for which we could narrow down the scope of the error locations to components or statements relevant to the faults.

4.2.1. *Results for the cruise control system*

Table 4 shows the evaluation results for the cruise control system. In this case study, we investigated in particular whether architecture-based software testing could detect mutants related to unobservable paths. The number of mutants related to unobservable paths were fewer than that of the mutants inserted by the state

Table 4. Architecture-based software testing results for the cruise control system.

	State machine-based testing [15, 16]		Architecture-based software testing				
Mutation operator	# of mutants	# of mutants killed	# of mutants only related to unobservable path*	# of mutants killed		# of directly located error by POs	
AOIU	31	14	15	12	80%	4	33.3%
AOIS	148	66	124	86	69.3%	55	64%
ROR	79	21	54	25	46.3%	3	12%
LOI	49	18	30	20	66.7%	10	50%

*The number of mutants in our approach is fewer than that of [15, 16] because mutants related to the 'CruiseControl' and the 'Controller' modules are excluded.

machine-based testing, but we could see that most of mutants were related to unobservable paths as shown in Table 4. The results depicted in the number of the mutants killed by architecture-based software testing show that architecture-based software testing could detect more faults than the state machine-based testing. And POs inserted based on system's architecture could reduce the error locating time by as much as 64%.

The locations of the mutants seeded using the ROR operator cannot easily be approximated even by architecture-based software testing. This occurs because the ROR operator is assumed to seed a mutant to a conditional statement such that the fault introduced by the mutant is not directly observed through a PO, although it could be if the entire conditional statement is inspected. With regard to the remaining detected faults for which we could not approximate error locations, we could narrow down the scope to the modules or statements that are relevant to the faults.

According to the report by Mouchawrab *et al.* [15] the mean of the mutation scores of the cruise control system was 36.7%, whereas the evaluation results of the cruise control system indicate that the mean of the mutation scores of architecture-based software testing is on average 65.6% (see Table 5). The comparison results of Table 5 show that the effects of architecture-based software testing on the fault detection rate in the cruise control system are higher than that of the state machine-based testing and that of the structural testing, respectively, but is lower than that of the combined testing by 19.2%.

Table 5. Comparison with the three experiments of Mouchawrab *et al.* [15, 16] for the cruise control system.

Systems / Mutation operators	Mouchawrab *et al.* [15, 16]			
	State machine-based testing	Structural testing	Combined testing (state machine + structural)	Architecture-based software testing
AOIU	45.2	0	87.1	80
AOIS	44.6	3.4	91.2	69.3
ROR	26.6	1.3	67.1	46.3
LOI	36.7	0	93.9	66.7
Average	36.7	1.2	84.8	65.6

4.2.2. *Results for the elevator system*

For the evaluation of architecture-based software testing, we seeded the mutants to the 'Elevator' module only because it was involved in many unobservable interactions. As shown in Table 6, about a half of the mutants were related to this module and most of them were killed by POs defined based on system's architecture.

For the mutant operator AOIS, architecture-based software testing approach detected 205 of the mutants (i.e. 76.5% of the mutants was killed), but 150 of them were killed through POs while those remaining were detected during compilation or

Table 6. Architecture-based software testing results for the elevator system.

	State machine-based testing [15, 16]		Architecture-based software testing				
Mutation operator	# of mutants	# of mutants killed	# of mutants only related to unobservable path*	# of mutants killed		# of directly located error by POs	
AOIU	88	5	26	23	88.5%	15	65.2%
AOIS	556	42	268	150 (205*)	56% (76.5%*)	5	3.3%
ROR	107	10	53	30	56.6%	1	3.3%
LOI	192	18	93	41 (60*)	44.1% (64.5%*)	14	34.1%

*The number of mutants killed in parentheses includes the number of mutants killed during compilation with the Tester.

Tester execution. Similarly, 60 mutants (64.5%) introduced by the LOI operator were also detected, but 19 mutants were excluded for the same reason with the AOIS operator. The numbers in parentheses of Table 6 are the numbers of such mutants. However, when we ran the Tester to execute the test cases, the system threw exceptions or entered an infinite loop. This is certainly a result caused by mutants, but it is not clear whether it could be detected by architecture-based software testing. Actually the cause of an abnormal termination of the system was incorrect allocation of values to the variables for the elevator direction and elevator state, whose consequence is that elevator threads cannot terminate normally. These threads were terminated through the Tester, which is a PCO that executes test cases.

Architecture-based software testing was effective in locating the errors related to the AOIU mutation operator, while it was less effective for the AOIS and ROR mutation operators (see the last column of Table 6). In the case of the ROR mutation operator it is the same reason with the cruise control system case. However, In the case of the AOIU it is difficult to specify the reason.

The comparison results of Table 7 show that the effects of architecture-based software testing on the fault detection rate are on average 68.4% for the elevator system, which are higher than that of the state machine-based testing and that of the

Table 7. Comparison with the three experiments of Mouchawrab *et al.* [15, 16] for the elevator system.

Systems / Mutation operators	Mouchawrab *et al.* [15, 16]			
	State machine-based testing	Structural testing	Combined testing	Architecture-based software testing
AOIU	5.7	13.6	83.0	88.5
AOIS	7.6	23.2	91.4	76.5
ROR	9.3	19.6	67.3	44.1
LOI	9.4	18.8	79.2	64.5
Average	8.0	18.8	80.2	68.4

structural testing, respectively, but is lower than that of the combined testing by 11.8% for the elevator system. Architecture-based software testing, as a kind of grey-box testing, achieves quite high effects without relying on the source code.

4.3. *Answers to research questions*

Mouchawrab *et al.* [15, 16] reported that they could not detect faults related to sneak paths or specific parameters. To handle faults related to sneak paths, they repeated certain command calls and conducted replication experiments. In order to detect faults related to specific parameters, they used a combination of boundary value analyses and category partition techniques. However, for architecture-based software testing, we simply inserted POs into the unobservable interactions identified in the module view architecture and the behavioral view architecture. This approach allows checking whether such POs can detect faults in the specific paths and parameters.

RQ1: Does architecture-based software testing improve fault detection capability beyond the testing without architecture?

With respect to this question, we found that architecture-based software testing can better detect faults than the state machine-based testing or the structural testing, but no better than the combined testing.

In the two case studies, the fault detection rates of architecture-based software testing ranged from 44.1% to 88.5%. Unlike state machine-based testing or structural testing, architecture-based testing showed similar fault detection rates for both systems. The low rate of 44.1% was due to the many undetectable mutants inserted in the 'Elevator' module. For example, for the elevator system, many live mutants were related to the function of choosing the best elevator. As with choosing the 'best elevator', where the outputs can differ from state to state of a system, subsystem, or module, it is difficult to give pass/fail verdicts even when we observe the outputs and compare them with the architectural specifications. Moreover, there exist mutants generated by the ROR and LOI operators that can never be terminated. This is the case with the mutants inserted in the forms of '-0' or '~ 0'. These are undetectable mutants that can hardly be regarded as faults.

Architecture-based software testing was particularly effective for detecting four types of faults, e.g. AOIU, AOIS, ROR and LOI, which would not be easily detected by a single test case design technique alone. However, faults related to unspecified paths or sneak paths were not detected by architecture-based software testing either (Cf. Discussion section).

RQ2: Can faults be located efficiently when architecture-based software testing is used?

The answer to this question is that, not in all cases but in most cases, we can trace the exact locations of errors. As shown in the last column of Tables 4 and 6, the POs allowed directly locating from as little as 3.3% to as much as 65.2% of the errors. Even in the cases when we could not pinpoint error locations, we could usually narrow down the scopes of the suspected components because we had information about interacting components and monitored variables. POs could narrow down the ranges of suspected components to as specific as the methods or as rough as the modules related to the interaction. In architecture-based software testing, test cases include associated PO(s) and their expected outputs. When faults are detected, i.e. when the values observed at POs differ from the expected outputs, we could find the locations of the corresponding errors by checking statements that define, use, or change the values observed at the POs.

RQ3: Can oracles be generated easily when architecture-based software testing is used?

Architecture-based software testing provided a means of defining exact oracles with ranges of values by using architectural specifications defined by the scenario view, the module view and the execution view. In the case of the cruise control system, we could only determine oracles as ranges of values, for example, positive or negative values for the variables ‘throttle’, ‘setSpeed’ and ‘distance’ observed through POs. The exact value of ‘throttle’ could be determined while an ‘On’ event occurred and we could approximate its value with a range of values. It was difficult to get oracles for the variables ‘setSpeed’ and ‘distance’ because they were used by multiple threads.

5. Discussion

In the case studies of Sec. 3, fault detection rates of architecture-based testing were highest with the AOIU operators, next with the AOIS and LOI operators and lowest for the ROR operators. In the experiments of Mouchawrab *et al.* [15, 16] the conventional testing methods killed a small number of mutants for AOIU, AOIS and LOI operators. Many of those mutants were killed only after using the combined testing. In both case studies, architecture-based software testing yielded similar fault detection results without relying on combined conventional testing methods. AOIU, AOIS and LOI, where architecture-based testing showed good fault detection, were mutations for unary operators. This is because architecture-based testing makes observable the variables that are unobservable in certain states of the system. With the same amount of test effort put for observation, architecture-based testing enables us to focus on locations and variables that are architecturally important and therefore are necessary to observe or control. But, we cannot guarantee that the mutants terminated through our approach are identical to those through in the experiments by Mouchawrab *et al.* [15, 16].

Our case studies also illustrate that architecture-based software testing can be particularly effective for detecting certain types of faults (Cf. Sec. 3.3). That is, in the case studies, architecture-based software testing could detect faults that are related to specific parameters and thus would not be easily detected by a single test case design technique alone. For example, faults related to specific parameter values (such as a specific value of the ‘setSpeed’ variable of the cruise control system) are difficult to detect with the state-machine-based testing. And the variable 'setSpeed' should be set to a specific value in order to detect mutants seeded in the 'throttle' variable using the AOIS operator. In contrast, architecture-based software testing can detect such faults by observing the intermediate values of the interactions that would not be easily observable with the conventional testing. This indicates that architecture-based software testing can be utilized effectively for systems with a lot of unobservable interactions or with parameters involved in complicated predicate or computation relations.

Many faults undetected in the case studies were related to unspecified paths, or sneak paths. The state machine-based testing could not detect such faults. Neither could architecture-based software testing although it monitors the intermediate values between unobservable interactions. This is due to that architecture-based testing of this case study uses a state-machine diagram for deriving test cases and state invariants for defining oracles. As with the state machine-based testing, there exist uncovered state sequences of systems and states. Undetected mutants, among those generated by the AOIS and LOI operators, were on the paths that are not covered by the state-machine diagram.

Concerning the effects of error locating, we would not typically design architecture-based testing for the purpose of error locating but such effects can be achieved by architecture-based testing as the result of the fault detection capability that it supports. Therefore, this paper does not present systematic procedures or methods for error locating that are based on system's architecture-based testing. In our case study investigation, we aimed at finding out what effects architecture-based testing would have on error locating. As explained in the experimental results of the two case studies, it is an effect that is introduced by the POs that have been inserted to increase observability. When a fault is detected by a PO, the location of the error that caused the fault can be narrowed down to the statements or the components that are directly or indirectly linked to the PO. So if an error occurs in a variable being observed, error locating is immediately terminated; otherwise the error is located by examining the statements that have the computation-use relationship or the predicate-use relationship with the variable. This result indicates that using architecture-based software testing together with conventional testing methods facilitates error locating.

6. Related Work

In this section, we describe works related to the test architecture concepts and PO/PCO concepts as well as discussions for works that use the term, architecture-based software testing.

6.1. *Test architecture*

The term 'test architecture' was used in UML Testing Profile (UTP) [9] and by Nishi [5]. In UTP, test architecture is a set of elements that participate in realizing the behavior of test cases with specific roles in testing. Test architecture in UTP supports black-box testing and defines the test structure and its behavior. Unlike the architecture-based software testing approach that we applied to the case studies in this paper, UTP does not use architectural specifications as a basis for defining test architecture. The role of test architecture in UTP is to specify test configuration and the focus of UTP is not effective fault detection or error locating or enhancing testability. The research of Nishi [5] attempts to define the scope of test system architecture, presenting notations associated with test design patterns and test architecture styles, which, however, does not show the details of such concepts. Test system architecture is similar to that of UTP in that IUT is merely one component of the test architecture. Test architecture notion in our approach is more general than that of UTP [9] and Nishi [5] because ours approach puts software architecture that is defined with various architectural views [10] as a base for designing test architecture design.

The concept of test architecture used in the research of Keum *et al.* [13] is similar to that of this paper, but the specific context of their test architecture is distributed SOA. Keum *et al.* [13] use a control flow graph (CFG) as a test model to describe flows of interactions among services, and show a means of generating architecture-enabled test scenarios based on the test model and test architecture. However, the research of Keum *et al.* [13] cannot be directly applied to the systems that do not have SOA style because its test case derivation method relies on the CFG architectural model.

6.2. *POs/PCOs*

The notions of POs and PCOs have existed in many different forms. The assert and print statements in programming languages can be viewed as POs and the test drivers and testing interfaces[e] in built-in contract tests [11] can be viewed as PCOs. Yu *et al.* [27] proposed a controllable and observable testing framework termed *SimTester*, which inserts breakpoints (a type of PO) into critical code locations and controls (a type of PCO) execution events to observe interactions between applications, device drivers and interrupt handlers. *SimTester* is a virtual platform used to test software systems for concurrency faults such as races and deadlocks. *SimTester* uses PO/PCO concepts, but its observers and controllers are modules developed to simulate interactions between applications and device drivers. The PO/PCO of architecture-based software testing differs from that of *SimTester* in that it is derived from architectural artifacts.

Vranken *et al.* [28] use the PCO concept to enhance the testability of hardware-software systems. Their PCO [28] is conceptually similar to ours, but the description of the boundary between the hardware and software is not clear, and it is difficult to adopt it for software testing. This research focuses on testing on hardware even if the authors note that they discuss testing on hardware and/or software. However, Vranken *et al.* [28] define three PCO operating modes, i.e. the transparent, observation and test modes. In the transparent mode, PCO is not used, whereas in the observation mode it plays the same role as POs of our approach. In the test mode, the control input is passed to both the PCOs, akin to the PCO role in our approach. We use transparent and observation modes as an on/off switch of test codes in our approach.

6.3. *Testing approaches using architectural artifacts*

Ali *et al.* [22] summarize integration testing approaches using architectural artifacts such as collaboration diagram as well as state-chart, use-case and sequence diagrams. Other approaches [23] that use collaboration diagram are TESTOR, which utilizes state-chart and collaboration diagram; UIT, which utilizes use-case diagram and sequence diagram; and the SeDiTec approach, which uses testing interactions between the classes involved in a sequence diagram. The approaches above are similar to ours in that they use architectural artifacts from module view and scenario view to obtain interaction information. They aim to generate test paths and test cases which include interaction information with which to detect faults, whereas our approach uses architectural artifacts to locate POs/PCOs and thereafter generates test cases, including interaction information.

Bass *et al.* [10] found that software architecture can play an important role in testing by supporting the production of a wide variety of test artifacts and test automations if a proper mechanism to connect the architecture and code is provided. There are studies that use formal descriptions for software architecture or software architecture specification models [12, 24–26, 33, 34] for testing. The works of [24, 25, 33, 34] use architectural specifications to generate test artifacts including test cases, to detect architectural faults, or to locate errors. Our approach differs from these studies in that ours is not limited to architectural defects and also checks whether the system behaves as required at the integration test level and at the system test level. Using test architecture explicitly enhances the testability, fault detection and error locating capabilities at both test levels. Soria *et al.* [26] uses a use case map as an architecture model defined for mapping from architecture to code. This research uses architecture model to localize faults at the code level, not to generate test artifacts for testing or detect faults efficiently. Thus, the problems addressed by this research are a kind considerably different from those that our architecture-based software testing addresses by generating test cases based on test architecture with POs and PCOs. Lee *et al.* [12] presented a framework of architecture-based software testing by providing a method that derives test architecture based on software architecture together with the foundational concepts and principles of architecture-based software testing to help utilize test architecture for software testing. It conducted case study experiments but did not provide quantitative results pertaining to the effects of fault detection and error locating.

7. Conclusion and Future Work

In this paper, we conducted two case studies for investigating the effects of architecture-based software testing with respect to fault detection and error locating. The investigation results show that architecture-based software testing can improve fault detection rate by 44.1%–88.5% and reduce error locating time by 3%–65.2%, compared to the conventional testing that does not rely on test architecture. From the analysis of the case study results, we found the following effects of architecture-based software testing. First, architecture-based software testing shows higher fault detection rates than the conventional testing method and, unlike the conventional testing, resulted in similar fault detection rates in both case studies. It has been confirmed by comparing the fault detection rate of architecture-based software testing with those of the conventional testing methods and the combined testing method. This effect was possible because architecture-based testing enables us to focus on the locations and variables that are architecturally important and therefore are necessary to observe or control. Second, concerning error locating, architecture-based software testing allows precise error locating. Even when the error location cannot be precisely pinpointed, it narrows down the scope of components that are suspected to have an error. This is because the location of the error that causes a fault can be narrowed down to the statements or the components that are directly or indirectly linked to the POs that observed the fault. Third, architecture-based software testing is particularly effective for detecting certain types of faults. Because architecture-based software testing enables us to observe the intermediate values of the interactions that would not be easily observable with the conventional testing. Lastly, architectural specifications can provide a means of defining oracles with exact values or ranges of values that architecture-based software testing could utilize but the conventional testing would not, as discussed in the answer to RQ3 in Sec. 3.

Although effectiveness of architecture-based testing in fault detection and error locating were confirmed through the case studies of this paper, they are limited in their domains and scales. Therefore, for a more thorough evaluation we will study our research questions with larger scale real-world software systems in the expanded set of domains.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2017R1D1A3B03028609), by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A1A3005162), and by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017M3C4A7066210).

References

1. J. Lee, S. Kang and D. Lee, Survey on software testing practices, *IET Softw.* **6**(4) (2012) 1–8.
2. International Software Testing Qualifications Board, Worldwide Software Testing Practices Report 2015–2016, 2018, https://www.istqb.org/documents/ISTQB_Worldwide_Software_Testing_Practices_Report.pdf.
3. Capgemini, World Quality Report 2018–2019, 2018, https://www.capgemini.com/service/world-quality-report-2018-19/.
4. T. Katayama, Motivation to establish a concept of test architecture, in *Proc. International Workshop on Software Test Architecture*, 2011.
5. Y. Nishi, Perspective of research on test architecture design, in *Proc. International Workshop on Software Test Architecture*, 2011.
6. P. Clements and M. Shaw, The golden age of software architecture revisited, *IEEE Softw.* **26**(4) (2009) 70–72.
7. M. Shaw and P. Clements, The golden age of software architecture: A comprehensive survey, *IEEE Softw.* **23**(2) (2006) 31–39.
8. I. Schieferdecker, Test automation with TTCN-3 — State of the art and a future perspective, *Testing Software Systems*, Lecture Notes in Computer Science Vol. 6435, 2010, pp. 1–14.
9. Object Management Group, Inc., UML Testing Profile (UTP), 2018, http://www.omg.org/spec/UTP/1.1/.
10. L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, 3rd edn. (Addison-Wesley, 2013).
11. H. G. Gross ,*Component-Based Software Testing with UML* (Springer, 2004).
12. J. Lee, S. Kang and C. Keum, Architecture-based software testing, *Int. J. Softw. Eng. Knowl. Eng.* **28**(1) (2018) 1–8.
13. C. Keum, S. Kang and M. Kim, Architecture-based testing of service-oriented applications in distributed systems, *Information Softw. Technol.* **55**(7) (2013) 1212–1223.
14. R. M. Hierons, Generating complete controllable test suites for distributed testing, *IEEE Trans. Softw. Eng.* **41**(3) (2015) 279–293.
15. S. Mouchawrab, L. C. Briand, Y. Labiche and M. D. Penta, Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments, Technical Report No. TR SCE-08-09, Carleton University, 2009.
16. S. Mouchawrab and L. C. Briand, Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments, *IEEE Trans. Softw. Eng.* **37**(2) (2011) 161–187.
17. C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler and S. Schulz, *An Introduction to TTCN-3*, 2nd edn. (John Wiley and Sons, 2011).
18. Software-artifact Infrastructure Repository (last accessed Jan. 2019), http://sir.unl.edu/portal/index.php.
19. Y.-S. Ma, J. Offutt and Y. R. Kwon, MuJava: An automated class mutation system, *Software Testing, Verification Reliability* **15**(2) (2005) 97–133.
20. Muclipse, An Open Source Mutation Testing plug-in for Eclipse (last accessed Jan. 2019), http://muclipse.sourceforge.net/.
21. P. Ammann and J. Offutt, *Introduction to Software Testing* (Cambridge University Press, 2008).
22. S. Ali, L. C. Briand, M. J. Rchman, H. Asghar, M. Z. Iqbal and A. Nadeem, A State machine-based approach to integration testing based on UML models, Technical Report No. SCE-05-02, Carleton, 2006.

23. A. Abdurazik and J. Offutt, Using UML collaboration diagrams for static checking and test generation, in *Proc. 3rd International Conference on the Unified Modeling Language*, 2010, pp. 383–395.
24. D. J. Richardson and A. L. Wolf, Software testing at the architectural level, in *Proc. ACM SIGSOFT '96 Workshops*, 1996, pp. 68–71.
25. A. Bertolino, P. Inverardi and H. Muccini, Software architecture-based analysis and testing: A look into achievements and future challenges, *Computing* **95** (2013) 633–648.
26. A. Soria, J. A. D.-Pace, M. R. Campo, Architecture-driven assistance for fault-localization tasks, *Expert Systems* **32**(1) (2015) 1–22.
27. T. Yu, W. Trisa-an and G. Rothermel, SimTester, A controllable and observable testing framework for embedded systems, in *Proc. Virtual Execution Environments*, 2012; T. Yu, An observable and controllable testing framework for modern systems, in *Proc. ICSE Doctoral Symposium*, 2013, pp. 1377–1380.
28. H. Varanken, M. F. Witteman and R. C. van Wuijtswinkel, Design for testability in hardware-software systems, *IEEE Design Test J.* **13**(3) (1996) 79–87.
29. I. Salman, A. T. Misirli and N. Juristo, Are students representatives of professionals in software engineering experiments? in *37th IEEE Int. Conf. Software Engineering*, 2015, pp. 666–676.
30. ISO/IEC, ISO/IEC IS 9646 Part 1–7, Information technology-open systems interconnection — Conformance testing methodology and framework, Standard, 1995.
31. T. Walter and I. Schieferdecker and J. Grabowski, Test architectures for distributed systems: State of the art and beyond, in *Proc. Int. Workshop on Testing Communicating Systems*, 1998, pp. 149–174.
32. B. Uzun and B. Tekinerdogan, Model-driven architecture based testing: A systematic literature review, *Inform. Softw. Technol.* **102** (2018) 30–48.
33. B. Uzun and B. Tekinerdogan, Model driven architecture based testing tool based on architecture views, in *Proc. 6th Int. Conf. Model-Driven Engineering and Software Development*, 2018, pp. 404–410.