Black-box Testing for Evolving COTS-Based Software

Ye Wu
Information and Software Engineering Department
George Mason University
Fairfax, Virginia, USA, 22030-4444
wuye@ise.gmu.edu

Abstract. Black-box testing methods are widely used in system level and integration testing. But in the context of COTS-based engineering, when newly modified COTS components are adopted into the system and the source code is not available, black-box testing is a necessary and feasible approach to assure that these COTS products do not adversely affect the software. In this paper, we first discuss challenges that we are facing when adopting evolving COTS products. Then we describe different black-box testing techniques and how to adjust them to effectively test evolving COTS-based software systems.

1 Introduction

One of the most important objectives of adopting COTS products is to improve the maintainability of COTS-based software systems. Ideally, when COTS products are updated, minimal effort is required to re-evaluate COTS-based software systems. However, to achieve both test effort reduction and quality preservation is extremely difficult, due to many unique features of COTS-based engineering.

COTS products can evolve in many different ways. When faults are discovered in a delivered product, corrective maintenance activities need to be performed. When new features are added in the product or enhancements are made to the previous product, perfective maintenance activities need to be performed. Other activities such as preventive and adaptive activities may also be periodically performed. Maintaining COTS-based software is significantly different from maintaining traditional software. The major difference is that, when maintaining traditional software systems, quite often the same company or even the same group of people who develop the product will maintain the system. They have full control of the maintenance process and complete knowledge of the system. On the other hand, for COTS-based software, COTS products are usually maintained by the COTS provider, while COTS product users have neither control of the maintenance process, nor complete knowledge of the product or the future course of the product enhancements.

Because of these unique features of COTS products, black-box testing is intuitively the most applicable testing strategy for COTS-based software systems. To adequately apply black-box based testing techniques on COTS-based software, many
new challenges that have never been investigated before need to be explored. Therefore, in Section 2, we first analyze the new challenges we are facing when dealing with evolving COTS products. Because of these new challenges, direct adoption of black-box testing is possible, but may induce too much overhead and may also overlook potential faults in the new integrated systems. To adequately apply black-box testing strategies, adjustments need to be made. In Section 3 of this paper, we focus on the issues of adopting black-box testing, and use adjusted partition and boundary value testing to provide an effective way to test evolving COTS-based software systems. Some preliminary results that are described in Section 4 have shown the great potential of this strategy.

2 Challenges facing COTS-based systems

A significant difference between software maintenance for traditional systems and for component-based systems lies in the fact that, for traditional systems, software implementation and maintenance activities will be carried out by the same company. The company has the control to evaluate the impact of the changes and has full access to the source code, so that they can minimize the impact of the change. COTS providers do not have the real context for how their products are used by COTS users, and therefore COTS providers cannot effectively evaluate the impact of change. Intuitively, the best way to restrict the impact to a minimal scope is to maintain the same set of interfaces to the services that will be provided. However, practices have demonstrated that this cannot always be achieved, due to issues such as architectural and technology changes. In this section, we will use some real enterprise COTS products to demonstrate the challenges we could face in maintaining evolving COTS-based systems.

Changes in COTS products can happen in different ways and in different contexts; most of them can be classified as the following:

1. COTS-user transparent changes
Most of the changes in corrective maintenance activities, and a small part of the changes in perfective and adaptive maintenance activities, belong to this category. Corrective maintenance often corrects component internal logic errors or errors due to inadequate interface-interface interaction. These errors are not caused by wrongly defined interfaces. Therefore, very often, the fix for such errors often leaves the interfaces of the COTS product unchanged. Sometimes perfective maintenance may also end up with the similar scenario. For instance, XML parsers are very popular COTS products that have been built into many different enterprise systems, such as WebLogic, Tomcat, the Sun ONE application server, etc. XML parsers have gone through many different versions to improve performance issues for each and every version; the same interfaces are kept to ensure stability for all of its users.
2. Changes that affect the COTS product user, but do not affect user system architecture

Unfortunately, not all changes in COTS products can be classified as user-transparent changes that require minimal efforts for COTS users. There are many causes for these kinds of changes, such as poor design in the previous version, changes in the technology, changes in the standard, or sometimes a strategic move to lock in customers. For instance, the WebLogic Enterprise Service (WLSE) components that have been integrated with WebLogic went through dramatic changes from WebLogic 6.1 to 7.0, the main reason being that version 6.1 does not conform with the mainstream security architecture. Some of the changes in this component require minor changes in COTS users’ code. For instance, the authentication service underwent a significant structure change in the WLSE component, but client code only need to change the APIs that will be called. But changes in some other services require extensive maintenance activities as described below. In general, new COTS products will provide new interfaces to substitute the deprecated interfaces or to provide additional services. Sometime, original interfaces are still provided just for backward compatibility.

3. Changes that affect COTS user architecture

If an interface’s name change is the only change for the COTS products, programmers only need to change the interface name of their code; then their system is ready to be integrated with the new COTS products. But very often changes in interface names also mean architecture changes. In other words, the way that the new component is to be integrated, and how different interfaces communicate with each other, could be changed as well. If this is the case, major revision for the client program is required. For instance, the architecture of the assess control module service within the WLSE component is completely changed. It changed from access control list (ACL), which is resource oriented, to security policy, which is user oriented. To adopt this revision, the WLSE client component may have to completely rewrite all access control-related code.

3 Black-box testing for evolving COTS-based software

3.1 Issues in black-box testing for evolving COTS-based software

While testing evolving COTS-based software, black-box testing techniques can be directly adopted as the techniques that only rely on specifications. Nevertheless, COTS products have usually been tested, and very often a full scale retesting of the integration is usually not acceptable to COTS users. To adequately apply traditional black-box testing techniques for software components, the following issues need to be taken into consideration:

1) COTS information unavailability: COTS products usually do not provide source code to their customers, and detailed changes to their products are very often un-
available as well. Without these types of information, COTS users will encounter
great difficulties when attempting to determine the test adequacy for black-box
testing when they upgrade COTS products

2) **Component customization**: When adopting COTS products into a software sys-
tem, what the system requires and what the COTS product provides usually do
not perfectly match with each other. Therefore, when adopting a component into
the application, customization is required to reconcile the differences between the
two specifications. Consequently, customization is the focal point of the testing.

3) **Component interface**: Unlike traditional software systems, interactions can be
conducted in a flexible way. For instance, we can access an object through its
public methods, or sometimes we can access public data members directly. As
for component-based software, interfaces are the only point of contact. In other
words, the interactions among components have to go through interfaces. There-
fore, the specifications of interfaces play a key role in testing software compo-
nents. If properly conducted, efficient and effective component testing can be
achieved through interface-based black-box testing strategies.

### 3.2 Partition testing and boundary value testing

Black-box testing has been widely used for a long time, and different approaches
can be used for different situations. Basically, the techniques can be classified as: 1)
usage-based black-box testing techniques, such as random testing or statistical test-
ing, 2) error-based black-box testing techniques, which mainly focus on certain error-
prone instances, according to users’ experiences with respect to how the program
behavior usually deviates from the specification. This type of approach includes
equivalence partitioning testing, category-partition testing, boundary-value analysis,
decision table based testing, etc., and 3) fault-based black-box testing techniques
which focus on detecting faults in the software. In this section, we use partition test-
ing and boundary value testing as examples to demonstrate how to test evolving
COTS-based software.

Equivalence partition testing tries to divide the input domains into k different dis-
joint partitions \( p_1, p_2, \ldots, p_k \), where \( p_1 \cup p_2 \cup \ldots \cup p_k = \text{input domain} \), and \( p_i \cap p_j = \emptyset \) for any \( i \) and \( j \) where \( i \neq j \). Values from each partition have the “all-or-none” prop-
erty. I.e. if an element is selected as the input for the component and that element fails
the component, all other elements in that partition will also fail the component. On
the other hand, if the component succeeds with that element as input, all other ele-
ments in the same partition will succeed as well. Therefore, if partitions which satisfy
this characteristic can be generated, then test cases can be easily generated by ran-
domly selecting one element from each partition. In addition, the test cases generated
by the equivalence partition testing strategies can provide us with confidence about
the entire domain without any redundancy. Unfortunately, there is no systematic way
to generate equivalence partitions, and many times it is impossible to develop equiva-
ience partitions. To overcome these difficulties, many systematic partitioning ap-
proaches are developed. If formal specifications exist, sometimes partitions can be
automatically derived. Given the non-formal functional specification, Ostrand and Balcer proposed a systematic partition testing strategy called category partition testing [6]. Category partition testing starts with an original functional specification and continues through the individual details of each subprogram to derive a formal test specification.

For partition testing, input domain will be classified into different disjointed partitions. Ideally, every element in each partition has the same possibility to either reveal or hide a fault. But based on programming experiences, this is may not usually be true. Values that are close to the boundary of the partition are more likely to expose errors. Thus, when boundary value testing each partition, not only one element will be selected. Instead, additional elements close to the boundary will be tested as well [4]. Boundary value analysis addresses test effectiveness – focusing on the high-risk areas. Partition testing addresses test efficiency – reducing the number of test cases needed to obtain a certain level of confidence for the software under test. Therefore, boundary value testing strategies should be used along with partition testing strategies.

3.3 Partition testing and Boundary value testing for evolving COTS-based application

As discussed in Section 2, COTS products can evolve in many different ways. Each scenario requires a different strategy to assure the quality of the integrated system:

- **User transparent changes.** These types of changes often leave the original partition unchanged. According the software quality requirements and other constraints, different adequacy criteria can be adopted. If the COTS user knows where the fault is, and which interfaces have been changed for each modified interface, at least one test case needs to be selected to rerun. However, the modified interfaces can be invoked in different locations, under different contexts. So a stricter criterion can require a test from each partition, which involves an execution of a changed interface in order to rerun. If a changed interface is unknown to the COTS user, similar criteria can be adopted by assuming that all interfaces are potential candidates for changed activities.

- **Changes that involve new interfaces or even new architecture.** These types of changes can potentially change the original partition. This can occur in the following different situations:
  1. **Generalization repartition:** The effects of generalization repartition can cause one partition to expand while the other shrinks. An extreme scenario is to merge the two partitions. To test this scenario, each changed partition needs to be retested. In addition, test cases need to be generated to validate those areas for which ownerships have been changed.
  2. **Specialization repartition:** The effects of specialization repartition could lead to the generation of additional partitions. The testing of specialization requires all affected partitions to be retested, and new test cases be generated for newly derived partitions.
3) **Reconstruction repartition**: Reconstruction repartition is potentially more complex than the first two types of repartition. If this is the case, substantial retesting is required.

4) **Boundary value and repartition**: When partitions are changed, correspondingly boundary values for each partition will be changed. To test the changed partition, in general, at least one test is required to be selected to re-run. To be more careful, more test cases should be selected to ensure that the values around partition boundaries work properly.

In addition to the approaches that are discussed in 3.1 through 3.3, there are many other black-box testing methodologies, for instance, decision table testing, mutation testing [2][3], syntax testing [1], finite-state testing [1], cause-effect graphing [5], error guessing [5], etc. These black-box testing techniques can be adjusted in similar fashion.

### 4 Pilot Study

To demonstrate how black-box testing strategy can be applied and how effective these black-box testing strategies are, we conduct a pilot study on coffee-machine software, which includes a coffee-machine server component and a coffee-machine client. The server component contains 12 classes and exposes seven interfaces, and the coffee-machine client will invoke all seven interfaces. Two versions of the coffee-machine server component are developed. The second version introduces 10 modifications to the first version. Seven of the modifications are corrective maintenance activities, while three of the modifications are perfective maintenance activities. To test the first version, we adopted Ostrand and Balcer’s category partition-testing [6] strategy and obtained 48 partitions. After the modifications, 17 partitions were changed or added. To validate the effectiveness of the testing strategy, we created total 34 faulty versions of the program, each with one inserted fault. After randomly selecting 17 test cases from each of the partitions, 21 faults were identified, while 13 faults remain undetected.

By analyzing the testing result, we discovered that five out of the 13 faults, which were not revealed, are not detectable through black-box testing. They can, however, be detected through unit-level white-box testing [7]. The other eight faults are either caused by the boundary values which are not covered in our test, or by inaccurate partition.

The pilot study has demonstrated that, with minimal testing effort, 76.5% \((21+5)/34\) of the fault can be detected, but for safety-critical software systems, more rigorous retesting is required.
5 Conclusion

When COTS components are modified, effectively and efficiently re-evaluated COTS-based software is necessary. Due to many unique features of COTS-based software, black-box testing is one of the most feasible strategies. In this paper, we have demonstrated the potential problems of applying traditional black-box testing strategies to COTS-based systems, and provided a solution for conducting the testing through partition testing.

References