Component Testability and Component Testing Challenges

Jerry Gao, Ph.D.

San Jose State University
One Washington Square
San Jose, CA 95192-0180
Email:gaojerry@email.sjsu.edu

Abstract

Building high quality and reusable software components is very important for component-based software development projects. The testability of software components is one of the important factors determining the quality of components. As the concept of component engineering receives the wide acceptance in the real world, many practitioners begin or plan to begin to use the component engineering approach to develop component-based software. To develop high quality components, they are looking for answers concerning component testing and component testability. What is component testability? How to evaluate and measure? How to build software components which are testable? This paper shares our thoughts and understanding of component testability, and discusses and identifies the challenges and issues concerning to component testability.

Keywords: component testing, program testability, software engineering, software testing, and component engineering

Introduction

Software testability is one of important concepts in design and testing of software program and components. Building programs and components with good testability always simplifies test operations, reduces test cost, and increases software quality. As pointed out by James Bach [1], there is a set of program characteristics that lead to testable software, including operability, observability, controllability, understandability, and so on. Jeffrey M. Voas and Keith W. Miller [4] view software testability is one of three pieces of the reliability puzzle. They pointed out that software testability analysis is useful to examine and estimate the quality of software testing using an empirical analysis approach. In the component engineering paradigm, software development of component-based software, engineers have several questions concerning component testability. What is component testability and related factors? How to check, measure, or evaluate the testability of software components? How to design and develop testable components to achieve good testability?

Understanding of Component Testability

Here we address different perspectives of component testability in component engineering, including component observability, component traceability, component controllability, and component understandability.

- ***Component Observability and Traceability***

  According to Roy S. Freedman [2], *software observability* indicates how easy to observe a program in terms of its operational behaviors, input parameters, and outputs. This implies that the design and definition of a component's interfaces (such as incoming and outgoing interfaces) affects its observability. We can use the technique proposed by Roy S. Freedman to check a component's interfaces to evaluate how easy to observe its operations and outputs corresponding to its inputs. In the practice of component engineering, we found that the component traceability is another very important factor that affects the component observability.

  "Traceability" of a software component refers to the extent of its build-in capability of tracking the status of component attributes and component behavior. It has two folds: a) *behavior traceability*, and b) *trace controllability*. *Behavior traceability* refers to the degree to which a component facilitates the tracking of its internal and external behaviors. In the real world, component engineers have learned to check and monitor the internal and external behaviors of software components by adding a program tracking mechanism in software. In the real practice, engineers are not successful in delivering software components with good behavior traceability due to the following two reasons. First, in the development of software components, engineers used to pay much attention to track component internal behaviors than external behaviors. Therefore, component testers, integration engineers and customers have the difficulty in monitoring and checking the external behaviors of software components. Next, component developers are learning how to design and develop traceable components due to the lack of standardized component trace formats and tracking mechanism [3].

Trace controllability refers to the extent of the control capability in a component to facilitate the customization of its tracking functions. With trace controllability, engineers can control and set up various tracking functions such as turn-on and turn-off of any tracking functions and selections of trace formats and trace repositories. Although the current commercial components and most in-house components do not provide this capability, it is an ideal and cost-effective feature for testers and maintainers to support debugging, component integration, and system. Trace controllability is really useful and necessary for complicated components with customization functional features in a distributed environment.

As we discussed in [3], there are six types of component traces. They are operation, performance, error, state, GUI event, and communication traces. We can add the tracking capability into software components using different approaches. To support the access of the built-in tracking functions, we must define a standard tracking interface inside components. Standardization of the component tracking interface and trace formats is very important to build a systematic tracking solution for component-based programs.

- ***Component Controllability***

Controllability of a program (or component) is an important property which indicates how easy to control a program (or component) on its inputs/outputs, operations, and behaviors [2]. Component developers look at the *"controllability"* of a software component in three aspects: a) behavior control, b) feature customization, and c) installation and deployment. The first has something to do with the controllability of its behaviors and output data responding to its operations and input data. The next refers to the built-in capability of supporting customization and configuration of its internal functional features. The last refers to the control capability on component installation and deployment.

Test engineers and component customers concerned about other factors of component controllability, which may directly affect the component testability. The trace controllability in software components is a typical example. Testers and customers expect components to provide a set of control functions in software components so that they can use them to monitor and check diverse component behaviors according to their needs. We believe component trace controllability is very useful in component debugging, component integration, and system testing. In addition, component testers and customers expect software component vendors to generate components with test controllability to support acceptance testing and unit testing in a standalone mode. Component test controllability refers to a component's capability of retrieving and exercising component tests. Unfortunately, most current component vendors do not provide components with this capability due to the lack of research results on how to design and develop testable components.

- ***Component Understandability***

Component understandability depends on how much component information is provided and how well it is presented. Presentation of component documentation is the first factor. For a component, there are two sets of documents. The first set is written for users. It includes component user manual, user reference manual, and component application interface specifications. The other set is written for component engineers, including component analysis and design specifications, component testing and maintenance documents. The second factor to component understandability is the presentation of component program resources, including component source code and its supporting elements, such as its installation code, and test drivers. The final factor is the presentation of component quality information, including component acceptance test plan and test suites, component test metrics and quality report. Most current third-party components did provide users with component user manual and application interface specifications. However, only some of them provide user reference manual, and most of them do not provide any quality information. Although it seems reasonable for component vendors to hide detailed test information and problem information, customers will expect them to provide quality information, acceptance test plan, and even test suites for components in the near future.

Challenges to Test Software Components

In the component engineering paradigm, one of the primary goals is to generate reusable software components as software products. The third-party engineers use the components as parts to build specific software systems according to the requirements given by customers. Therefore, the testability of a program highly depends on the testability of involved components and their integration. There are several concerns on building and testing of software components.

- ***How to reuse component tests?***

  Considering the evolution of software components, we must pay attention to the reuse of component tests. The primary key to the reuse of component tests is to develop some systematic methods and tools to set up reusable component test suites to manage and store various component test resources, including test cases, test data, and test scripts. In the current engineering practice, software development teams use an ad-hoc approach to creating component test suites through a test management tool. Since existing tools usually depend on different test information formats, repository technologies, database schema, and test access interfaces, it is difficult for engineers to deal with diverse software components (for example, third-party components) with a consistent test suite technology. This problem affects the reuse of component tests in the component acceptance testing and component integration.

  To solve this problem, there are two alternatives. The first is to create a new test suite technology for components with plug-in-and-test techniques. With this technology, engineers are able to construct a test suite for any component, and perform component tests using a plug-in-and-test technique. Clearly, it is necessary for us to standardize software component test suites, including test information formats, test database schema, test access interfaces, and to define and develop new plug-in-test techniques to support component unit testing at the unit level.

  The other alternative approach is to create component tests inside components, known as build-in tests. Unlike the first approach, where component tests created and maintained in a test suite outside of a component, this approach creates component tests inside components. Clearly, it simplifies component testing and reduces the component test cost at the customer side if a friendly test-operation interface is available. To execute the built-in component tests, we need extra function facilities to perform test execution, test reporting, and test result checking. Therefore, there is a need to standardize test access interfaces supporting the interactions among components, test suites, and built-in tests.

- ***How to construct testable components?***

  An ideal testable software component is not only deployable and executable, but also testable with the support of standardized components test facilities. Unlike normal components, testable components have the following features.

  - Testable components must be traceable. As defined in [3], traceable components are ones constructed with a built-in tracking mechanism for monitoring various component behaviors in a systematic manner.

  - Testable components must have a set of built-in interfaces to interact with a set of well-defined testing facilities. The interfaces include a) a test set-up interface, which interacts with a component test suite to select and set up black box tests; b) a test execution interface, which interacts with a component test driver or test execution tool to exercise component functions with a given test; and c) a test report interface, which interacts with test reporting facility to check and record test results.

- ➢ Although testable components have their distinct functional features, data and interfaces, they must have a well-defined test architecture model and built-in test interfaces to support their interactions to component test suites and a component test-bed.

- ➢ Testable components with built-in tests must use a standardized mechanism to enclose the built-in tests. With this mechanism, we can access and exercise the built-in tests in a consistent way.

There are three questions regarding to design of testable components. The first question is how to design and define the common architecture and test interfaces for testable components. The next question is how to generate testable components in a systematic way. The final question is how to control and minimize program overheads and resources for supporting tests of testable components.

- ▪ ***How to construct component test drivers and stubs in a systematical way?***

Now, in the real world practice, engineers use ad-hoc approaches to developing module-specific or product-specific test drivers and stubs based on the given requirements and design specifications. The major drawback of this approach is that the generated test drivers and stubs are only useful to a specific project (or product). It is clear that the traditional approach causes a higher cost on the construction of component test drivers and stubs. In the component engineering paradigm, components might have the customization function to allow engineers to customize them according to the given requirements. This suggests that the traditional way to construct component test drivers and stubs is very expensive and inefficient to cope with diverse software components and their customizable functions. We need new systematic methods to construct test drivers and stubs for diverse components and various customizations. The essential issue is how to generate reusable, configurable (or customizable), manageable test drivers and stubs for a component.

Component test drivers must be script-based programs that only exercise its black-box functions. There are two groups. The first group includes function-specific test drivers, and each of them exercises a specific incoming function (or operation) of a component. The second group contains scenario-specific test drivers, and each of them exercises a specific sequence of black-box operations (or functions) of a component.

Component test stubs are needed in the construction of component frameworks. Each test stub simulates a black-box function and/or behavior of a component. There are two general approaches to generating test stubs. The first approach is model-based, in which component stubs are developed to simulate a component, in a black box view, using a formal model, such as a finite state machine, a decision table, or a message *request-and-response* table. Its major advantage is model reuse and systematic stub generation. To cope with component customization, we need a tool to help us change and customize an existing model, and generate a new model. The second approach is operational script-based, in which test stubs are constructed to simulate a specific functional behavior of a component in a black-box view. The major advantage of this approach is the flexibility and reusability of test stubs during the evolution process of components. However, the challenge is how to generate these function-specific test stubs in a systematic manner.

- ***How to build a generic and reusable component test-bed***?

In general, a program test execution environment consists of several supporting functions: test retrieval, test execution, test result checking, and test report. Clearly, a component test environment must include the similar supporting functions. The primary challenge is how to come out a new component test-bed technology that is applicable to diverse components developed in different languages and technologies.

References

[1] Roger S. Pressman, *Software Engineering: A Practitioner's Approach (fourth edition),* McGraw-Hill, 1997.

[2] Roy S. Freedman, "Testability of Software Components***", IEEE Transactions on Software Engineering,*** Vol. 17, No. 6, June 1991.

[3] Jerry Gao, and Youjin Zhu, "Tracking Software Components", Technical report (http://www.engr.sjsu.edu/gaojerry/techreport/) in San Jose State University, in 1999. Submitted for publication.

[4] Voas, J. M. and Miller, Keith W.," Software Testability: The New Verification", ***IEEE Software,*** Vol. 12, No. 3: May 1995, pp. 17-28.