Testing Component-Based Software

Jerry Gao, Ph.D.

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

Abstract

Today component engineering is gaining substantial interest in the software engineering community. Although a lot of research effort has been devoted to analysis methods and design strategies of component-based software, a few papers address the testing of component-based software. The paper identifies and classifies the testing issues of software components and component-based software according to working experience. It discusses component testability in terms of controllability, traceability, test suite, presentation, test support, and configuration management. The paper proposes a model to measure the maturity levels of a component testing process. Finally, it shares our observations and insights on test automation for component-based application systems.

Keywords

Component engineering, software component testing, software testing, software testability.

Introduction

Today component engineering is gaining substantial interest in the software-engineering community [1]. Although there are many published articles addressing the issues in building component-based programs, very few address the problems and challenges in testing components and component-based software. As more third-party components are available in the commercial market, more software workshops start to use the component engineering approach to develop their products. They have encountered new issues and challenges in testing of software components and component-based programs.

In the past two decades, researchers and software testing tool vendors developed many white-box, black-box test methods and tools for traditional software programs [1]. Since 1990's, many researchers have spent their effort on solving the testing problems of object-oriented software [2]. As the advances of the software component technology, people begin to realize that the quality of component-based software products depends on the quality of software components and the effectiveness of software testing processes. As pointed out by Elaine J. Weyuker [3], we need new methods to test and maintain software components to make them highly reliable and reusable if we plan to deploy them in diverse software projects, products, and environments.

Why do we need new methods for components and component-based software? What are the new issues and challenges in testing of component-based software? How to find the solutions to solve these problems? What is an effective test process for components? How to measure it? What obstacles we may encounter on the road to test automation of a component testing process? Today, engineers and managers in the software industry are looking for the answers to these questions. Until now, only a few articles [4][5][6][7] reported recent effort in this area. This paper provides our answers the above questions from the different extent.

This paper identifies, examines, and classifies the test issues in the development of component-based software according to working experience and recent research results. It points out the challenges in component testing, component integration, and component-based software testing in Section 1. Section 2 discusses the testability of software components in terms of different perspectives, and shares our understanding on how to increase the testability of components. Moreover, Section 3 discusses a maturity model for measuring the capabilities and features of a component testing process. This model will be useful to help managers and engineers understand what they can do to improve a component testing process. In Section 4, it shares our insights on the obstacles and solutions for test automation of software components and component-based software in Section 4.

Problems in Testing Component-Based Software

A component-based program consists of four groups of components. The first group includes commercial components from other vendors. The second group contains in-house developed components for other projects. They are reused in the program. The other group consists of in-house developed components, which are updated for reuse in the program. The last group is a set of newly constructed components. The quality of a component-based program depends on the quality of these components, and the effectiveness of component unit tests, integration and system tests.

Problems in Testing Software Components

According to the current practice of component engineering, there are six essential issues in testing software components:

- *Ad-hoc test suites for software components.* Current commercial component vendors do not provide customers with any test suite and test information as the part of component products, even no acceptance tests and quality reports. To check the quality of a component and understand its behaviors, customers must spend a lot effort on understanding the provided component documents, and then create a test suite and test driver to perform acceptance testing. For in-house components, engineers usually create their own test suites to support component testing. However, these suites are developed using an ad hoc manner. In other words, they are created using inconsistent test formats, diverse technologies, various repositories and tools. This has something to do with the fact they are created by different teams in a number of projects. The major side effect of this is that these test suites are very hard to be reused, integrated, maintained, and managed in a systematic way to support component testing, component integration, and system testing.

- *Adequate testing of components.* There are two major issues relating to adequate testing of components in the current engineering practice. The first issue is that most in-house components (such as full-experience and partial-experience components) do not have adequate test sets because they are designed based on specific project requirements and functions. Reusing the components and their test sets in other projects or products without checking may cause the adequate testing problem. The second issue is that current component vendors do not provide any information about their test criteria and test metric in components. It is, therefore, very hard for customers to understand component test adequacy, test coverage, and test quality.

- *Poor testability of reusable components.* Testability of software components usually includes to two folds: a) *observability*, and b) *controllability*. *Controllability* of a program (or component) is an important property which indicates how easy to control a program (or component) on its inputs, operations, behaviors and outputs. *Observability* of a program (or component) is another critical property which indicates how easy to observe a program in terms of operational behaviors and outputs relating to inputs. From a customer's point of view, the testability of current software components is poor due to the following reasons.

  1. No external tracking mechanisms and tracking interfaces in software components for a client to monitor or observe external behaviors.

  2. No built-in controllable interfaces in software components to support the execution of unit tests, checking the test results, and reporting errors.

  3. No built-in tests or controllable test suites for software components to support back-box tests at the unit level.

  Although most in-house components contain some built-in tracking code for error trace and exception reporting, there is no consistent tracking mechanism, trace format, and tracking interface for all components. At best, engineers may add some built-in tests into their components. However, there is no enforcement on the consistency of the built-in mechanisms and test interfaces.

- *Expansive cost on constructing component test drivers and stubs*. In the component engineering paradigm, software components are reusable parts for component-based software products. To support component testing, engineers have to construct test drivers and test stubs. In the past, engineers used to construct product-specific test drivers and stubs (or simulators) for a specific a project based on given requirements. This approach becomes very ineffective and costly for component-based software projects because of the evolution of reusable components and their diverse functional customizations. For in-house reusable components, engineers usually use an ad hoc approach to develop simple test drivers and/or test stubs for unit tests. However, they are not easy to be reused and updated, and integrated to support component integration and system tests due to the following factors.

  - They are developed using ad-hoc methods, technologies, and computer platforms.

  - They are project-specific or product-specific.

  - They have no well-defined standard interfaces between components and test suits, and components with a test bed.

- *Ad-hoc component testing processes and certification criteria.* Today, many software workshops have established in-house quality control processes for software products. During the course of paradigm shift from traditional software construction to component-based software construction, they frequently run into a question about the difference between a component quality control process and a software program quality control process. They are not sure if the existing quality control process and standards can be applied onto software components. Due the lack of a standard component quality control process, they frequently end up an ad-hoc component testing process and certification standards.

- *Ad-hoc management of component test suites and test drivers/stubs*. Configuration management method and tools are very important for supporting the evolution of software products. In the practice of component engineering, we must consider component test suites and test drivers/stubs are parts of a component product, and use a configuration management mechanism to support its evolution. However, in the real practice, configuration management of component test suites and test drivers/stubs may not be performed in a component test process. This explains why some software teams spend a lot of extra cost on maintaining component test suites and their drivers/stubs.

To solve these problems, practitioners in the real world are looking for the answers to the questions listed in Table 1. Although the existing software testing methods and tools can help testers generate black-box and white-box tests for components in a systematic manner, we are lack of solutions to these problems.

Table 1. Questions about software component testing

Component Testability	What is the testability of software components? How to construct testable components? How to increase the testability of software components? How to measure the testability of software components?
Component Test Suite	How to construct well-formatted test suites for software components? How to manage and maintain test suites for software components?
Test Stubs & Drivers	How to construct component drivers and stubs in a systematic approach? How to maintain and manage test drivers and test stubs in a cost-effective way?
Certification and Quality Control	What is a well-defined certification standard for software components? How does a quality control process of software components differ from a regular quality control process for a software product?
Component Test Automation	How to achieve test automation for software components? How to generate a reusable component test platform for components?

There are two major challenges for test managers and quality assurance managers. The first is how to define a rigorous quality control process with certification standards and test criteria for software components. The other is to find systematic solutions to automate the component testing and component integration.

Problems in Component Integration

In a component-based software product line, programs are built based on a set of software components. It includes third-party components, in-house components, and newly constructed application components. In fact, a product is an integration of customized components to meet the specific requirement set [10]. There are two factors, which affect the complexity of component integration. The first is the number of involved components. The other is the customization capability of components. Although the existing integration approaches, such as incremental integration, are applicable to component integration, engineers need new effective integration methods and systematic tools to support component integration. Several issues in component integration are listed below.

- *Difficult to form integration test suites based on component test suites.* There are two reasons. Constructing test suites in an ad-hoc manner is the first one, which causes inconsistent test information format, diverse access mechanism, and different data store technology. These affect the reuse of test suites for component integration. The other cause is the lack of test selection methods to build integration test suites based on component test suites. Most existing techniques [9] focus on the selection of white-box tests based on program structures, such as control flow or data flow. However, they are not applicable for selecting black-box tests for component integration and re-integration due to the following facts.

    - Component integration focuses on a) the interactions between components at the component level, b) the integrated structures of components, and c) the integrated functions based on customized components. Thus, test generation and test selection methods must address their integrated functional features at the component level.

    - The interactions and relationships between components are much more complicated than the interactions between procedures when communications and multithreading are involved.

    Therefore, we need new systematic methods a) to identify and select tests from black-box component test suites to form integration test suites, and b) to select tests from an integration test suite for regression testing.

- *High cost on building integration environments*. In the component engineering paradigm, a product is constructed based a number of components. Each product is the integration result of a set of customized components according to the given requirements. Therefore, as the number of components increases, the number of component integrates also increases. ??Building an integration environment for a component-based software system is expansive difficult and expansive due to the ad hoc construction of component test drivers and stubs. Meanwhile, component customization features complicate the integration infrastructure and environment.

- *Lack of component integration test models and test criteria.* For a component with customization features, it may be deployed in an integrated component system to support a specific subset of its implemented functional features. A component-based program is made of these customized components. They work together to support the integrated functional features. Hence, integration testing becomes complicated due to the customization capability inside components, and their diversified integrates. Supporting component integration needs new integration models and criteria. The models must represent component integration structure and integration functions at the component level, capture components and their customization features in the black-box view, and consider various interactions between components. Integration test criteria must concern the data domain coverage on interactions of components, integrated functional coverage, and integration structure coverage.

Here, we listed the obstacles and challenges in component integration and re-integration.

- How to define new test models and criteria for component integration?
- How to find systematic methods to identify and select tests from component test suites to form integration test suites?
- How to define a reusable integration infrastructure? How to build an integration test platform to cope with diversified component integrates?

Problems in System Testing Component-Based Software

System testing usually focus on checking system behaviors though performance test, stress test, recovery test, and installation test. In the development of a component-based program, we have seen several problems relating to system testing.

- *Hard to understand system behavior.* In system testing, system testers usually have the difficulty in understanding and tracking the system behaviors and functions due to the following facts:
  - Engineers use ad hoc mechanisms to track the behaviors of in-house components. This causes the problem for system testers to understanding the behavior of a system due to inconsistent trace messages and formats, and diverse tracking methods.
  - No built-in tracking mechanisms and functions in third-party components for monitoring their external behaviors.
  - No configuration function for tracking in components to allow clients to control and configure the built-in tracking mechanisms.
  - No systematic methods and technologies to control and monitor the external behaviors of the components in a component-based program.
- *Difficult on error isolation, tracking and debugging*. In a system, components, developed by different teams, may use different tracking mechanisms and trace formats. Inconsistent error tracking mechanisms in components is the major cause. Inconsistent trace format and error code in error trace messages is the other cause.

- *High cost on performance testing and tuning.* Performance testing for component-based programs is a major challenge in system testing due to the fact that current component vendors usually do not provide users with any performance information. Hence, system testers and integration engineers must spend a lot of effort to identify the performance problems and related components in performance testing and tuning. With the evolution of components and the system, performance testing cost and tuning effort will increase continuously.

- *Weak in system resource validation and monitoring.* Since most components do not provide their system resource information, it is difficult for system testers to locate the causes of the system resource problems during system testing.

There are two major challenges. The first is how to design software components with consistent mechanisms and interfaces to support the tracking of behaviors, functions, performance, and resources for components. The other is how to define a systematic method and technology to support engineers to monitor a distributed component-based system at the both component and system level.

Testability of Software Components

Software testability is an important concept in design and testing of software program and components. Programs with good testability always simplifies software test operations, and reduces test cost. In the past, we use the testability as a way to check if a program and components are easy to test. As discussed in [4], software testability can be viewed from two different aspects: *controllability* and *observability. 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. *Observability* of a program (or component) is another critical property which indicates how easy to observe a program in terms of operational behaviors, and outputs relating to inputs. Robert V. Binder [8] discussed the testability of object-oriented programs from six primary testability factors: representation, implementation, built-in test, test suite, test support environment, and software process capability. Each of the six factors is further refined into its own quality factors and structure to address the special features in object-oriented software, such as encapsulation, inheritance, and polymorphism. In this section, we discuss the testability of software components from the component engineering point of view.

Component Observability and Traceability:

"Observability" of a software component has two folds: component "traceability", and the incoming and outgoing interfaces. The component traceability refers to the extent of behavior tracking capability of components. There are different types of tracking scopes. *Tracking component internal behaviors* is the first scope that includes tracking of component internal operations, internal object states, events, and performance. *Tracking component external behaviors* is the second scope that includes tracking of component external operations and performance, external visible events, and public property statuses. *Tracking test results* is the

other one. The generated traces include six types: operational traces, performance traces, error traces, state traces, event traces, and communication traces.

Engineers can implement the tracking capability into software components using different approaches. To support the access of tracking functions, engineers must build a tracking interface inside components for easy of use. Standardization of the component tracking interface and trace formats is very important to build a systematic tracking solution for component-based programs.

Component Controllability:

For component developers, *"controllability"* of a software component includes three folds. The first refers to 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 other is its installation capability. For test engineers and component users, *"controllability"* of software components, must include two additional factors. One of them refers to the control capability on build-in component behavior tracking. With tracking control functions, component testers and users can easily select and check different types of component behaviors through a build-in tracking interface. The second factor has something to do with the control capability on build-in tests in components. Through a control interface, component users and testers can exercise built-in component tests in a testing mode.

Component Presentation:

Component presentation consists of four factors. Presentation of components for users is the first factor. It includes component user manual, user reference manual, and component application interface specifications. According to these documents, users learn how to use a component. Presentation of design and analysis of components is the second factor. Engineers use component analysis and design documents to understand a component in terms of its interfaces, properties, functional capabilities, behaviors, and constraints. The component program is the third factor. It includes component source code and its supporting elements, such as its installation code, test driver and stubs. The last factor is relating to component testing and quality information. It includes component test plan and/or test suites, test metrics, test reports and problem reports.

It seems reasonable for component vendors to hide detailed test information and problem information from their customers. However, it is a good idea to provide components with an acceptance test plan and quality information, such as test metric report.

Component Test Suite:

There are several ways to construct component test suites. Creating external test suites for programs and parts is the traditional way. The basic idea is to use an external tool to help engineers create, update, manage, and maintain test cases and related information in a test repository. Experience tells us that this is a very effective approach to supporting both black-box and white-box tests for in-house software components. Later, engineers can select and reuse the tests in a test suite for integration and regression testing. To use this approach, engineers need a good software test management tool, and a well-defined standard for all test information, such as test cases, test procedures, and test results. "Consistency" is the key to set up component test suites in developing component-based software. To avoid of ad-hoc construction of test suites, we must use standard test formats, compatible technology, and consistent repository.

Another approach is known as build-in tests. The basic idea is to build tests inside components. This simplifies component testing and reduces test cost if a test interface is also available to exercise and access the build-in tests. However, this method has its disadvantages. For example, it is not easy to reuse the built-in tests for other testing unless a built-in access interface is available. The other issue is that testers need some extra facilities (such as test reporting, test retrieval) to support component testing.

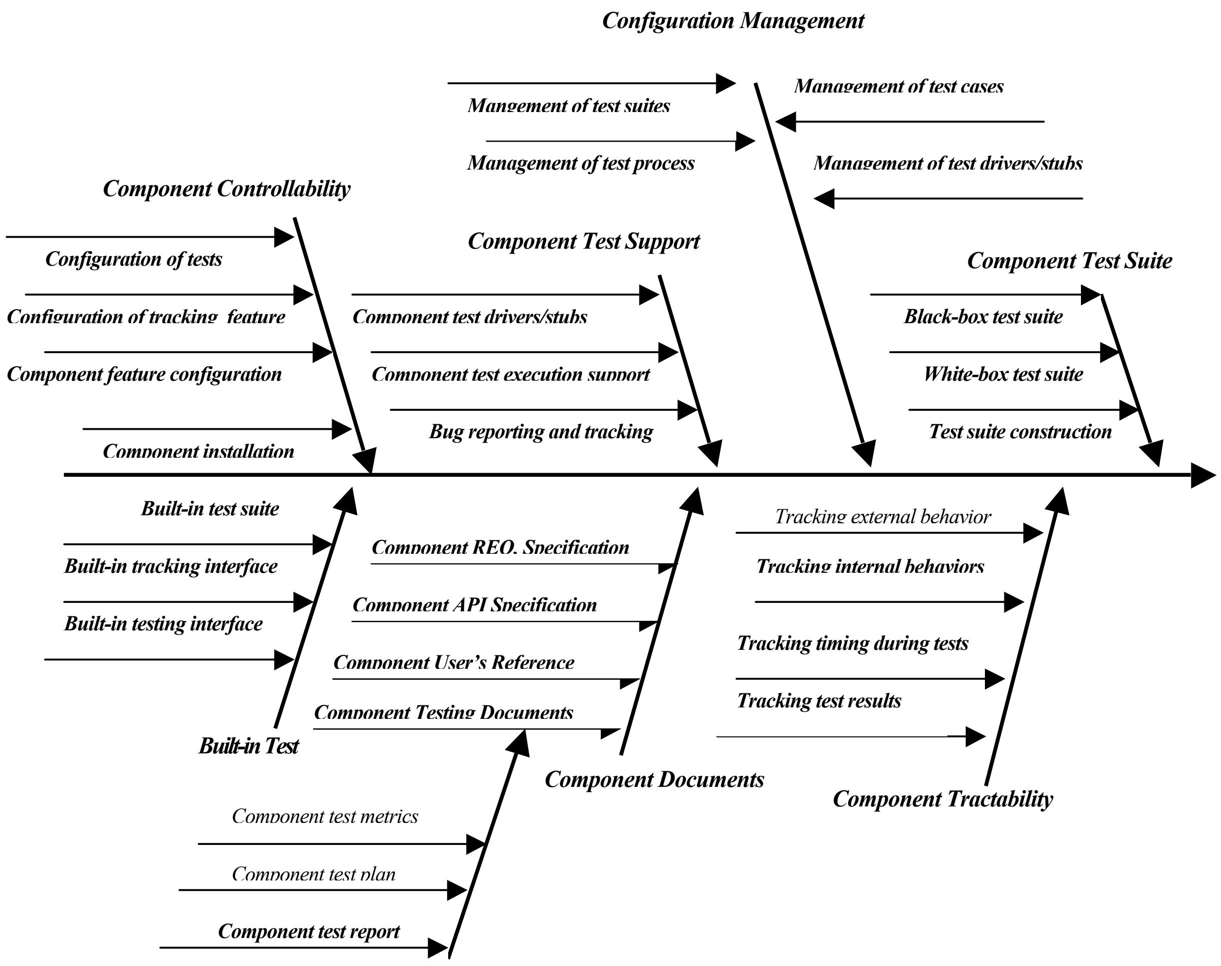
Configuration Management
Mangement of test suites
Management of test cases
Management of test process
Management of test drivers/stubs
Component Controllability
Configuration of tests
Configuration of tracking feature
Component feature configuration
Component installation
Component Test Support
Component test drivers/stubs
Component test execution support
Bug reporting and tracking
Component Test Suite
Black-box test suite
White-box test suite
Test suite construction
Built-in test suite
Built-in tracking interface
Built-in testing interface
Built-in Test
Component REQ. Specification
Component API Specification
Component User's Reference
Component Testing Documents
Component Documents
Tracking external behavior
Tracking internal behaviors
Tracking timing during tests
Tracking test results
Component Tractability
Component test metrics
Component test plan
Component test report

Figure 1. Testability of Software Components

Therefore, to support component testing effectively, we need standard formats for test information in test suites, a consistent access interface between components and test suites, and a standard test execution interface for users to exercise the built-in tests.

Configuration Management:

Configuration management is not only important for software development but also necessary to software testing. Today, software workshops use test management tools and configuration management tools to support the evolution of software products and test suites. Test stubs and drivers, however, are not included in the list of configuration management, because we are used to think they are part of products.

In the life cycle of component-based software, its evolution depends on modifications and upgrades of involved components. After changing reusable components, engineers must update related test suites, test drivers and test stubs to support successive regression tests. To reduce the evolution cost on regression testing, configuration engineers must add component test drivers and stubs into their task list. To support the evolution of components, they must keep tracking the versions and releases of component test suites and problem databases for each component. Managers must enforce version control and release control on all related resources of involved components, including documents, source code and execution code, test suites, test drivers and stubs, and problem databases.

Component Test Support

Testing software components needs several types of supporting facilities. They are test generation tools, component test drivers and stubs, a component test bed with supporting tools for test execution. Since available test generation methods and tools can be used to support test generation for components, let us focus on the construction of component test drivers/stubs, and discuss the set-up of a reusable component test bed.

- ***Constructing component test drivers and stubs systematically***. In the past, we used to generate module-specific or product-specific test drivers or/and stubs according to the given requirements and design specifications. Since these modules and products are different, we, thus, used to construct test drivers and stubs in an ad hoc manner. This works fine for a traditional product-specific software process. However, in the component-based software construction process, this approach causes a high construction cost on component test drivers and stubs due to their poor reusability. To cope with diverse software components and their customizable functions, we need new systematic ways to construct test drivers and stubs for software components.

- ***Building a generic 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 challenge here is how to construct a generic test bed, which is reusable to components based on different languages and technologies. To achieve this goal, we need standardize the following interfaces between components and its test bed. These interfaces are listed below.

- *Test execution interface between components and a test bed.* It supports a test bed to interact with components to support test execution, test result checking, and test reporting.

- *Test information access interface between a test bed and component test suites.* It supports a test bed to interact with test suites to retrieve test information.

- *Interaction interface between a test bed and a component test driver (or a component test stub).*

Maturity Model for A Component Testing Process

As component engineering gains a wide acceptance in today's software industry, many software companies have begun to set up component-based software product lines. Building cost-effective products needs a productive product line. Delivering high quality component-based software needs a well-defined component testing process. To understand the status of a component test process, it is important for a manager to have a tool to measure the effectiveness of the process. A maturity model is a tool that is useful to measure the maturity level of a process in an organization. In this section, we define a maturity model to help managers measure the status of a component test process. It focuses on component test standards, test criteria, management procedures, measurement activities. Figure 2 shows the four process maturity levels. They are defined as follows.

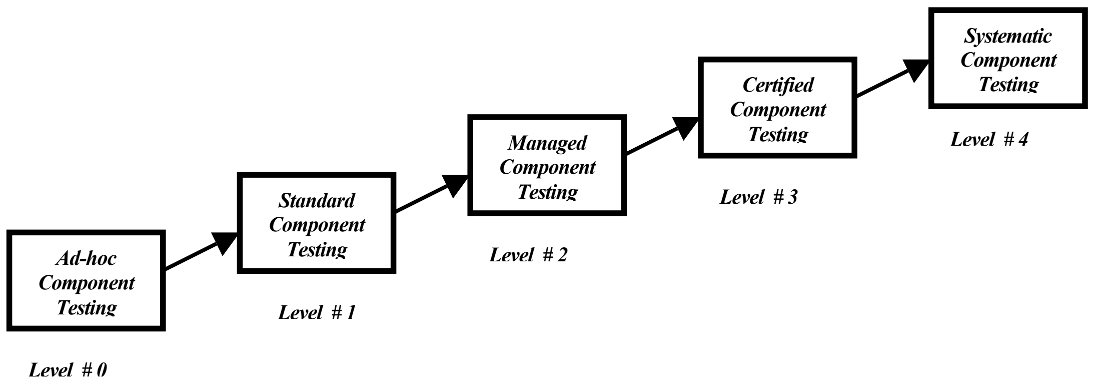
Ad-hoc Component Testing
Level # 0
Standard Component Testing
Level # 1
Managed Component Testing
Level # 2
Certified Component Testing
Level # 3
Systematic Component Testing
Level # 4

Figure 2. Maturity Levels for A Component Testing Process

Level 0: Ad-hoc Component Testing:

A component testing process is considered as an ad hoc process if it has the following characteristics: a) ad hoc test information formats, including test cases, test procedures, test data and scripts, b) ad hoc test generation and test criteria for both white-box and black-box tests, c) ad hoc quality assurance standards and quality control systems, d) ad hoc construction of component testing environment, such as test drivers and stubs, and e) inconsistent requirements and mechanisms on tracking component behaviors.

An ad hoc component test process is inefficient and costly due its poor reusability of test information and test drivers/stubs. Besides, it is very hard for managers to control and manage it. The most critical issue of the process is the quality of delivered components. Without well-defined test criteria, managers can not control the quality of tested components.

Table 1. Activities and tasks for level 1.

Standard Information	Management Procedure	Well-defined Criteria	Well-Designed Mechanisms
Test Case, Test Data	Configuration Management	Component White-box Test Criteria	a) Configuration mechanism for components, including documents, code, test suite, test drivers, and test stubs, problems.
Test Script, Test Report	Component Quality Control	Component Black-box Test Criteria	
Test Plan Document	Component Testing Workflow	Component Acceptance Test Criteria	
Problem/Bug Report	Problem Tracking Workflow	Component Quality Control Criteria	b) Component tracking mechanism
Test Metric Report			c) Problem tracking mechanism
Component Trace Message			

Level 1: Standard Component Testing:

A component testing process is characterized as a standard process if managerial operations and engineering activities are performed based on a set of test information standards, pre-defined management procedures, well-defined criteria, and well-designed mechanisms. The standards define requirements and formats on test plan, test case and data, trace messages, test reports and problem reports. Management procedures refer to a component quality control process, a component testing workflow, a configuration management procedure, and a problem tracking workflow. Test criteria include white-box and black-box test criteria, acceptance test criteria, and quality control criteria. The defined mechanisms support component tracking, problem tracking, and configuration management. All characteristics of level 1 defined in Table 1.

Level 2: Managed Component Testing:

A component testing process is categorized as a managed process if it collects the detailed measures of the process and component quality, including the measures of component test cost, component test metrics, component quality metrics, and process measurements. All detailed features of level 2 are listed in Table 2.

Table 2. Activities and tasks for level 2.

Test Cost Measurement	Test Measurement	Quality Measurement	Process Management
Component test design cost	Black-box test metrics	Component problem & quality metrics	Component testing process
Component test operation cost	White-box test metrics	Component test coverage measures	Component problem tracking
Test drivers and stubs cost	Test effective metrics		Component quality control
Configuration management cost	Test complexity metrics		
Component documentation cost			

Level 3: Certified Component Testing:

A component testing process is considered as a certified process if it has defined and implemented a certification standard and procedure for software components. The certification standard includes certification procedure, test plan, test tools, test platform and environment, criteria, test metrics, and test report. The test plan includes certification tests, which focus on testing of user accessible component features, installation, and customization or configuration functions. Moreover, this process has a well-defined certification procedure and workflow. A designated engineer or group, known as a component certifier, implements this procedure based on the given standard. After completing the certificate tests for a component, the certifier will issue a certificate for a component product according to the test report. This level includes all characteristics defined for level 2.

Table 3. Activities and tasks for level 3.

Component Certification Standard	Component Certification Criteria	Certification Process Management
Component certificate quality standard	Function conformance test criteria for certification	Component certification test procedure
Component certificate document standard	Performance test criteria for certification	Component certification procedure
	Component documentation criteria for certification	

Level 4: Systematic Component Testing:

A component testing process is characterized as a systematic process if it has defined and implemented systematic methods and mechanisms to automate this process. To achieve this goal, engineers need well-defined systematic methods to support their operations and activities. Table 4 lists the essential systematic methods in four areas: a) test suite design and construction, b) component design for testing, c) component test environment, and d) configuration management.

Table 4. Activities and tasks for level 4.

Component Test Suite	Component Design For Testing	Component Test Support	Configuration Management
Systematic white-box test generation	Build-in behavior tracking mechanism	Systematic test driver generation	Component test suites
Systematic black-box test generation	Build-in tracking interface	Systematic test stub generation	Component documents
Systematic component test selection	Build-in test mechanism	Systematic problem reporting	Component programs
Systematic test suite construction	Build-in test interface	Systematic test result checking	Component problems
Systematic test selection for integration		Reusable component test platform	Test drivers/stubs
		Systematic component integration	

Test Automation For Component-Based Software

Software testing is a very important phase in the development of software systems. Its cost is estimated to account for between 40 and 50 percent of total development costs [12]. To reduce test cost, test automation is the answer. The question is how to achieve this for component-based software products. What are the obstacles and challenges we may encounter on the road to the test automation of software components and component-based programs? How to find the solutions to resolve these? Here, we share our vision and observations.

Systematic management of component test information

Systematically managing component test information is the essential step to achieve test automation for component-based software development. To achieve this goal, a well-defined test information standard should be used on all projects in an organization to support test design, test documentation, test planing, and test execution and reporting. Moreover, a management system can be developed to control, manage, and maintain all component test suites in a systematic way. To cope with components built on diverse technologies and languages, we must pay attention to the compatibility, portability, and portability during the design of the management system.

Construction of testable components

A testable software component is not only deployable and executable, but also testable under the support of a standard component test-bed and its component test suites. To build a testable component, engineers must do:

- Increase component traceability. To increase the traceability of components, developers must define standard tracking mechanisms for component behaviors, and design a consistent interface and facilities to track internal and external behaviors of components. Moreover, components must provide a standard interface to allow clients to select and configure the available behavior tracking mechanisms and capabilities.

- Standardize build-in tests for components. Built-in tests method [7] provides a way to insert tests into components. However, the major issue is how to support clients to access and execute the built-in tests of a component in the black-box testing. The solution for that is to standardize built-in tests, including the mechanism, test format, and access interface for tests. Following these standards, engineers can build a reusable test system to access and maintain the built-in tests of components.

- Standardize a test interface for components. A test interface supports the interactions between a component and a test environment. It supports a) test execution, b) result checking, and c) error reporting. Since each component has different application interface, it is very important to design a standardized test interface for each component. With this approach, it is much easier for us to create a reusable test bed for supporting all components.

Systematic construction of component test drivers and stubs

In the component engineering paradigm, a set of reusable components can be customized to form an integration product based on the given requirements. This suggests that the traditional ad-hoc approach, which generates product-specific (or module-specific) test drivers and stubs, is not efficient and cost-effective for component-based software projects. We need new methods to construct test drivers and stubs for diverse components and various customizations. The essential question here is how to generate reusable, configurable (or customizable), manageable test drivers and stubs for a component.

Test drivers of a software component must be script-based programs that only exercise black-box functions of the component. They can be classified into two groups. The first group include function-specific test drivers. They exercise a specific function or operation of a component in the black-box view. The second group contains scenario-specific test drivers. They only exercise a specific sequence of black-box operations (or functions) in a component.

Test stubs of a software component simulate its black-box functions and/or behaviors. There are two general approaches to generating test stubs. In the first approach, a component's behaviors are represented using a formal model, such as a finite state machine, a decision table, or a message *request-and-response* table. A test stub is generated based on the given formal model to simulate the behaviors of a component. We call these test stubs as *model-driven* test stubs. This approach has the advantages on model reuse and stub generation. To cope with component customization, we need a model specification tool to help us change and customize an existing model, and generate a new model.

In the second approach, script-based test stubs are constructed to simulate specific functional behaviors of a component in a black-box view. A test stub manager controls the stubs of a component to respond the invocations of its accessible functions. One major advantage of this approach is the flexibility in supporting various customizations of functional features in a component. The other advantage is the reusability of test stubs because each stub simulates a specific accessible function of a component. However, the challenge is how to generate these function-specific test stubs in a systematic manner.

Systematic component test

Component tests can be generated in two different approaches. The first approach is the program structure-based test generation. Its goal is to uncover the errors of a component's internal structures and behaviors. Methods developed in the past [1] [3] can be used to generate structured-based tests (or white-box tests) for software components in a systematic way. The other approach is known as specification-based (or black-box) test generation, in which tests are generated based on the given specification requirements. Its focus is to find errors relating to external functions, interfaces, constraints, and behaviors. Some people [13][14] believe that formal methods will provide a systematic solution to black-box test generation for components. Until now, people are looking for formal models or approaches to specifying the behaviors of components in a black-box view.

Systematic support for test execution

Automating test execution in component testing needs a test environment, which includes three essential parts:

- A component test controller, which supports: a) automatic retrievals of test scripts from test suites, b) automatic controller - a controllable container for a script-based test driver, c) automatic checking of test results, and d) automatic test reporting of test results.

- A component test stub manager, which consists of: a) a test stub controller - a controllable container for test stubs, b) an access interface to test stub repository, c) a stub model creator, and d) a test stub generator.

- A component test bed, which supports a software component to interact with the component test controller and test stub manager.

The primary challenge here is how to build a generic black-box test environment to support diverse software components that are based on different technologies, interfaces, and implementation languages. To cope with this, we must pay attention to the following guidelines:

  - Use the platform-independent technology, such as Java, to increase the portability of a component test environment.

  - Generate test drivers and stubs using a portable programming language, such as a script language or Java.

  - Create flexible component test-bed, which supports diverse software components in terms of its execution, incoming/outgoing interface, as well as component interactions with the controller and stub manager.

  - Define a consistent interaction protocol and interface between the test bed and the component test controller as well as the component stub manager.

Systematic integration of components

Component integration focuses on checking errors in interactions between components, integrated functions, and integrated structures of components. Although existing integration test strategies [1][3] for traditional programs and object-oriented programs are available, very few research results and practical methods address the following issues in automating component integration.

- How to form black-box component integration test suites based on unit test suites? The key to solve this problem is to find a systematic method to identify and select integration test case (or scenario) based on a set of black-box unit test suites for related components.

- What is an effective component integration test model? What is the adequate test criteria for component integration? Since component integration tests usually focus on black-box functional features of components, the component integration test model and criteria must focus on the integration of component black-box tests at unit level. The existing research results [14] have provided a formal way to drive test data for a function's parameters of a component based on the category-partition method. What we need now is test models, which not only represent function (or operation) interactions between component interfaces, but also the category-partitions for data/objects involved in each function (or operation). Based on these test models, we can easily drive adequate component integration test criteria.

- How to set up a reusable integration infrastructure for component-based software? How to find systematic integration methods support diverse integrates of components on product lines? As mentioned in [10], a component-based software production line generates diverse products by integrating a number of reusable and customized components according to the given requirements. Hence, the component integration complexity depends on the number of involved components and their customizable functions. To reduce the integration cost and time, we need a reusable integration infrastructure and effective integration methods.

Reference

[1] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, New York, NY, 1990.

[2] Elaine J Weyuker, "Testing Component-Based Software: A Cautionary Tale", ***IEEE Software*,** September/October 1998.

[3] David Kung, Pei Hsia, and Jerry Gao, ***Object-Oriented Software Testing***, IEEE Computer Society Press, 1998.

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

[5] David S. Rosenblum, "Adequate Testing of Component-Based Software", Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-97-34, Aug. 1997.

[6] William T. Councill, "Third-Party Testing and the Quality of Software Components", IEEE Software, Vol. 16, No. 4, pp. 55-57, July/August 1999.

[7] Yingxu Wang, Graham King, and Hakan Wickburg, "A Method for Built-in Tests in Component-based Software Maintenance", Proceedings of the Third European Conference on Software Maintenance and Reengineering,1998.

[8] Robert V. Binder, "Design for Testability in Object-Oriented Systems***", Communications of ACM***, Vol. 37, No. 9, Sept. 1994, pp. 87-101.

[9] Gregg Rothermel and Mary Jean Harrold, "Analyzing Regression Test Selection Techniques", IEEE Transactions on Software Engineering, 22(8), pp. 529-551, August 1996.

[10] Patrick Copeland, "Approaches to Testing Componentization in the Windows CE", Proceedings of Quality Week'99, San Jose, May 1999.

[11] Andrea Chavez, Catherine Tornabene, and Gio Wiederhold, "Software Component Licensing: A Primer", ***IEEE Software,*** September/October 1998, pp. 60-69.

[12] B. W. Boehm, “Software Engineering”, ***IEEE Transactions on Computers***, C-25(12):1226-1241, December 1976.

[13] Phil Stocks and David Carrington, "A Framework for Specification-Based Testing***",*** ***IEEE Transactions on Software Engineering,*** Vol. 22, No. 11, November, 1996, pp.777-793.

[14] Paul Ammann and Jeff Offutt, “Using Formal Methods To Derive Test Frames in Category-Partition Testing”, (COMPASS 9) Proceedings of the Ninth Annual Conference on Computer Assurance, pp. 69-79, June 1994.