A component testability model for verification and measurement

Conference Paper · August 2005

DOI: 10.1109/COMPSAC.2005.17 · Source: IEEE Xplore

CITATIONS
45

READS
151

2 authors, including:

Jerry Gao
San Jose State University
223 PUBLICATIONS **3,015** CITATIONS

SEE PROFILE

A Component Testability Model for Verification and Measurement

Jerry Gao, Ph.D. and Ming-Chih Shih
San Jose State University, San Jose, USA
Email: gaojerry@email.sjsu.edu

Abstract

Since components are the major building blocks for component-based systems, developing high quality components is becoming very critical for in component-based software engineering. To generate high quality components, we must pay attention to component testability to ensure that reusable components not only can be tested by component vendors, but also can be easily validated by component users. Therefore, component testability analysis, verification and measurement become very important research topic in testing components and component-based systems. This paper discusses the component testability in a quantifiable approach based on a component testability analysis model. Engineers can use this model to verify and measure component testability during a component development process. Based on this testability model, the paper discusses component testability verification, and proposes a pentagram model for testability measurement.
Keywords: *component testability, testability analysis, testability measurement, component testing, and component-based software testing.*

1. Introduction

In component-based software engineering, reusable software components are the building parts in the construction of component-based systems. Any defects in reusable components will cause serious ripple impact to the quality of component-based systems, which are built, based on them. Hence, the quality control and validation of reusable components is very important to both component vendors and users in the construction of component-based software. According to [1], today's engineers encountered many new issues in testing component-based software and its reused components, including COTS components. One of the issues is poor component testability of reusable components. This not only increases the validation costs for component users, but also causes a great deal of difficulty in component validation.

What is *software testability*? According to IEEE Standard, the term of "testability" refers to *"the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met."* This definition indicates software testability is a measurable quality indicator that can be used to measure how hard it is for testers to achieve a particular testing goal in a test process, such as a test coverage criterion, and how easy to conduct test operations for a given software. As pointed out in [1], we need to answer the following questions to increase component testability.

- How to increase software testability by constructing highly testable components and systems?
- How to analyze, control, and measure the testability of software and components in all phases of a software development process?

Since software testability is one of the three pieces(software testability, software testing and formal verification ,and poor testability) of the software reliability puzzle [2], poor component testability not only suggests the poor quality of software and components, but also indicates that the software test process is ineffective. Like other requirements and design faults, poor testability is very expensive to repair when detected late during software development. Therefore, we should pay attention to software testability by addressing and verifying all software development artifacts in all development phases.

As pointed out by S. Jungmayr [3], testability is *the degree to which a software artifact facilitates testing in a given test context*. In our view, software testability is not only a measure of the effectiveness of a test process, but also a measurable indicator of the quality of a software development process. Hence, it is directly related to testing effort reduction and software quality. This paper addresses component testability issues by introducing a model for component testability analysis. Engineers can use this model to perform component testability verification and measurement during a component development process. In addition, the paper presents a pentagram model for component testability measurement based on the component analysis model; and some application examples and case study results are given for component testability measurement.
This paper is structured as follows. The next section reviews the related work on software testability analysis, verification and measurement. Section 3 discusses the

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

concept of component testability, and its analysis model. Section 4 covers the model application in component testability verification. Section 5 discusses component testability verification, and measurement based on a pentagram model. Some initial application examples and case study results are provided in Section 6. Final conclusion remarks and future research directions are included in Section 7.

2. Backgrounds and Related Work

In the past decades, there were a number of publications discussing the concepts and issues of software testability. In this section, we review these work and relate them with the research work presented in this paper.

R. S. Freedman [4] defines his function *domain testability* for software components based on two factors: *observability* and *controllability*. In his definition, *observability* is the ease of determining if specific inputs affect the outputs of a component, and *controllability* is the ease of producing specific output from specific inputs. His basic idea is to view a software component as a functional black box with a set of inputs and outputs. Measuring component testability can be done by checking these two factors.

Later, R. V. Binder [5] discusses software testability based on six factors: "representation, implementation, built-in test, test suite, test support environment, and software process capability". Each factor is further refined into its own quality factors and structure to address the special features of object-oriented software, such as inheritance, encapsulation, and polymorphism. His six factors do help us to understand software testability in an object-oriented view.

Recently, Jerry Gao et al. in [1] presents the component testability from the perspectives of component-based software construction. They define component testability based on five factors: understandability, observability, controllability, traceability, and testing support capability. For each factor, they also provide further refined factors. According to [1], component testability can be verified and measured based on the five factors in a quality control process. However, how to verify and measure these factors is not given in detail.

In component-based software engineering, component testability verification and measurement become very important because component testability is an important indicator of component quality. There are two basic approaches to verify component testability. They are:

- Static verification approach, in which component testability is verified in all phases of a component development process, including requirements analysis and specification, construction, and testing. The basic concept about the static verification approach is described in [1] [6]. However, engineers are lacking detailed verification guidelines and analysis models to perform this approach.
- Statistical verification approach, in which certain statistical methods are used to analyze and estimate component testability by examining how a given component will behave when it contains faults. One typical example of the statistical verification method can be found in [2].

Beside research work in testability verification, there were a number of research efforts addressing software testability measurement [7][8]. The focus was on how to measure software testability of conventional software at the beginning of a software test phase. The objective is to find out which components are likely to be more difficult and time-consuming in testing due to their poor testability. In the past years, a number of methods have been proposed to measure and analyze the testability of software [7][8][9]. They can be classified into three types:

- Program-based testability measurement methods [7], in which program source codes are used for measuring program testability.
- Model-based testability measurement methods [8], in which a program model, such as dataflow model in [8] is used as a base to measure program testability.
- Dependability testability assessment methods [3] [9], in which some kinds of program dependency are used as a base to measure program testability. For example, A. Bertolino and L. Stringi [9] developed their measurement method based on the dependency between inputs and outputs of black-box components.

In this paper, we propose a testability model based on the five testability factors of software components. The model further refines the five factors by providing a more practical and measurable detailed factors, which could be used to verify and measure component testability during a component development process.

3. A Component Testability Model

As reviewed in the previous section, now people have realized component testability is a very important quality indicator for components. How to verify and measure component testability in component-based software engineering is still a challenging task in today's engineering practice due to the following reasons:

- The current existing static testability verification approach lack of detailed component testability models and guidelines, which are useful for engineers to perform testability verification in a systematic manner.
- The statistical testability verification only can be used after components have been implemented completely. For most project managers, it is too late to discover components with poor testability during the testing phase. They hope to find out this earlier during a component development process.

- The existing testability measurement methods were developed based on implemented program and their structure or component dependency. They have their limited application on software component testability measurement, and have the same issue as the statistical verification methods.

In this section, we present a component testability model, which is developed by further refinements of the five factors of software components given in [1]. The model is presented using fishbone diagrams. We add two symbols '❍' and '☐' to group some factors in a fishbone diagram. The symbol '❍', known as a radio-choice symbol, is used to indicate that only one option out of a factor group can be selected as the dependent factor. The symbol '☐', known as a choice-box symbol, is used to indicate that more than one option in a factor group can be selected as the dependent factors.

Engineers can use this model as a base to develop their component verification guidelines and perform component testability verification during all phases of a component development process. In addition, component testability measurement can be done using a pentagram model, which is developed based on it.

3.1 Component Understandability

Component understandability is one of the five factors of component testability. It refers to the degree to which a component is specified and designed to facilitate the understanding of component users and engineers so that they can easily define component tests and criteria for component validation. This factor focuses on the quality of the generated component artifacts (such as requirements specification, API specification, and user reference manual) to see how well they are generated to facilitate component validation by users and engineers. It can be further refined to the following five factors:

- **Document availability**, which refers to the availability of component artifacts, including requirements specification, API specification, and user reference manual.
- **Document readability**, which refers to how well the given component specification documents are written to enhance the understanding of component users and test engineers.
- **Requirements testability**, which refers how well the component requirements and API interfaces are generated to make sure they are testable. Here, component requirements can be classified into non-functional requirements and functional requirements.
- **Requirements measurability**, which refers how well the component requirements and API interfaces are generated to make sure they are measurable.
- **Component usability**, which refers to how well the generated component user manual, API specification are easy for user operations.

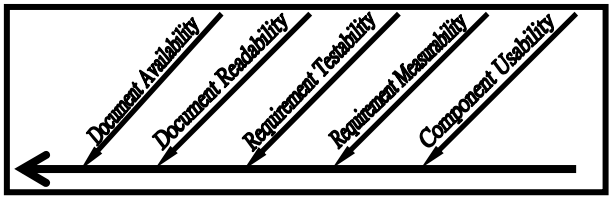

Figure 1. Component Understandability Factors

3.2 Component Observability

Component observability is the second factor of component testability. It refers to the degree to which components are designed to facilitate the monitoring and observation of component functions and behaviors of component tests. To enable engineers to verify and evaluate component observability, we further refine it based on four factors: (see Figure 2)

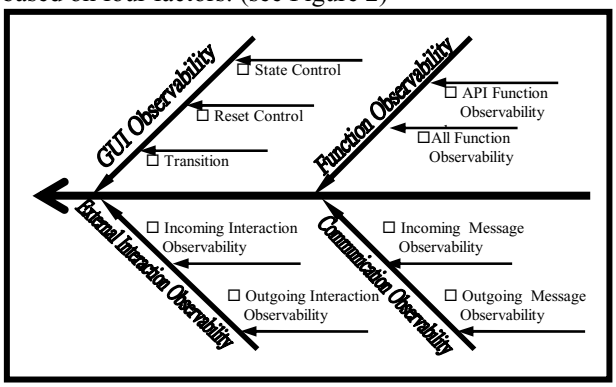

Figure 2. Component Observability Factors

- **GUI Observability** – For a component with GUI interfaces, it refers to a component's capability of supporting the observation of its interactions with its users in user interfaces. It can be checked and measured from three aspects: user inputs, GUI events, and outputs to the user.
- **Function Observability** – It refers to the component capability of supporting its functions in observing their inputs and corresponding outputs during testing. It can be further refined to two levels:
  - API-function observability, which only focuses on the verification and measurement of domain function observability for API functions.
  - All-function observability, which refer to the domain function observability for all defined in the component.
- **External Interaction Observability** – This refers to the component capability in supporting the observations of external interactions with other components in the same computer.
- **Communication Observability** - For a component supporting message-based communications, it refers to a component's capability of supporting the observation of incoming messages and corresponding outgoing messages.

3.3 Component Controllability

Component controllability is the third factor of component testability. It refers to the degree to which a component is designed to facilitate the control of its executions during validation. As shown in Figure 3, component controllability can be further refined to the following five factors.

- **Component Environment Control**: It refers to the built-in component capability that supports component environment installation, configuration set-up, and deployment.
- **Component State-based Behavior Control**: This refers to the built-in capability that facilitates the control of component state-based behavior, such as re-setting component states, state control function and transition control function.
- **Component Execution Control**: This refers to the built-in component execution control capability that enables executions of a component in different modes, such as test mode, normal function mode, control mode, and so on. With this capability, users and testers can start, restart, stop, pause, and abort a program as they wish.
- **Component Test Control**: This refers to the built-in capability that facilitates the component control for component validation. It includes: a) set-up Meta data, b) invoke a function, c) stimulate an event, or d) trigger an incoming message.
- **Component Domain Function Control**: This refers to the built-in capability that facilitates the component function controllability. It can be classified into two different levels: a) all-function controllability for all defined functions, b) API-function controllability, which only refer to the functions defined in its API interfaces.

All these types of component control capability may be required, designed, and implemented to enhance component controllability. During all phases of a component development process, we can check these factors through software reviews and inspection to see what kinds of controllability is required, designed, and implemented. It is easy to review a control capability based on three levels: automatic, semiautomatic, and manual. To achieve the automatic level, the given component must list the select controllability as its part of built-in function requirements, In addition, its design, implementation, and validation must be performed based on the given controllability requirements.

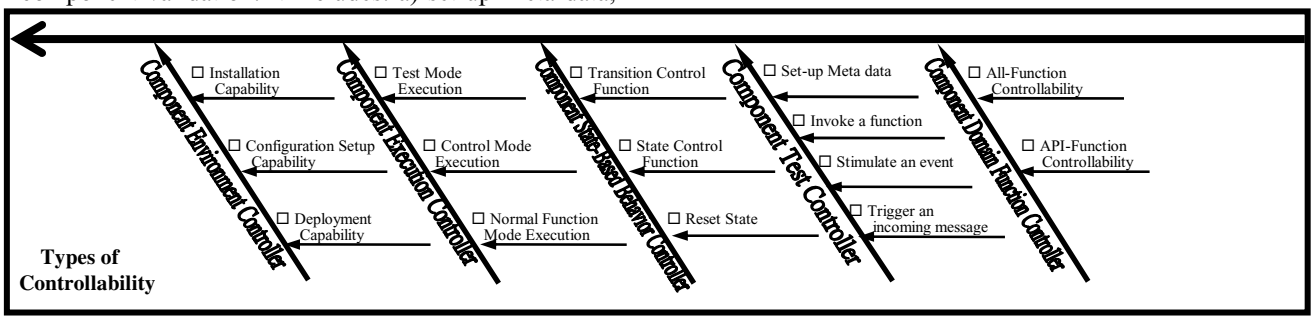

Figure 3. Component Controllability Factors

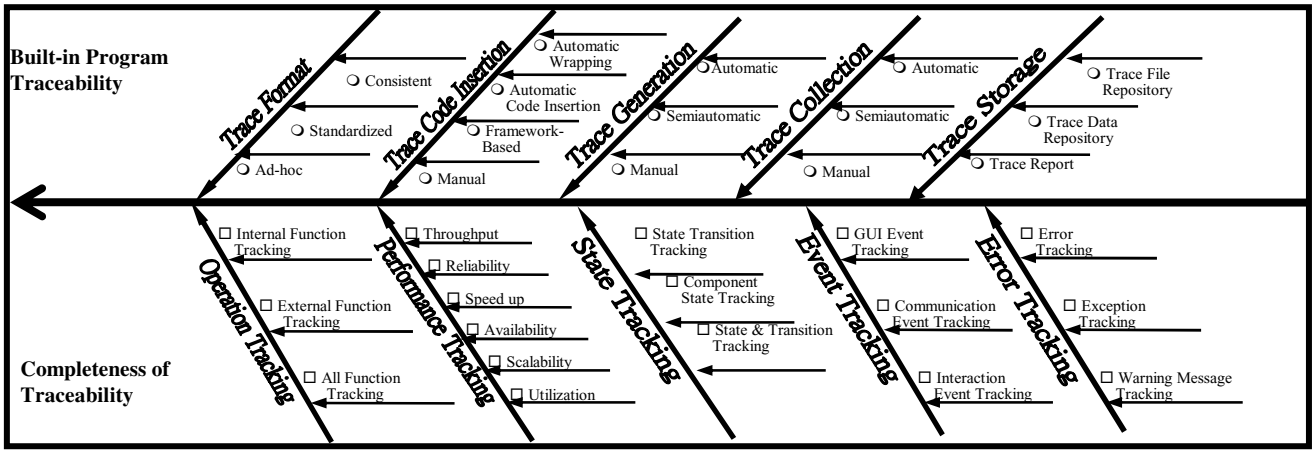

Figure 4. Component Traceability Factors

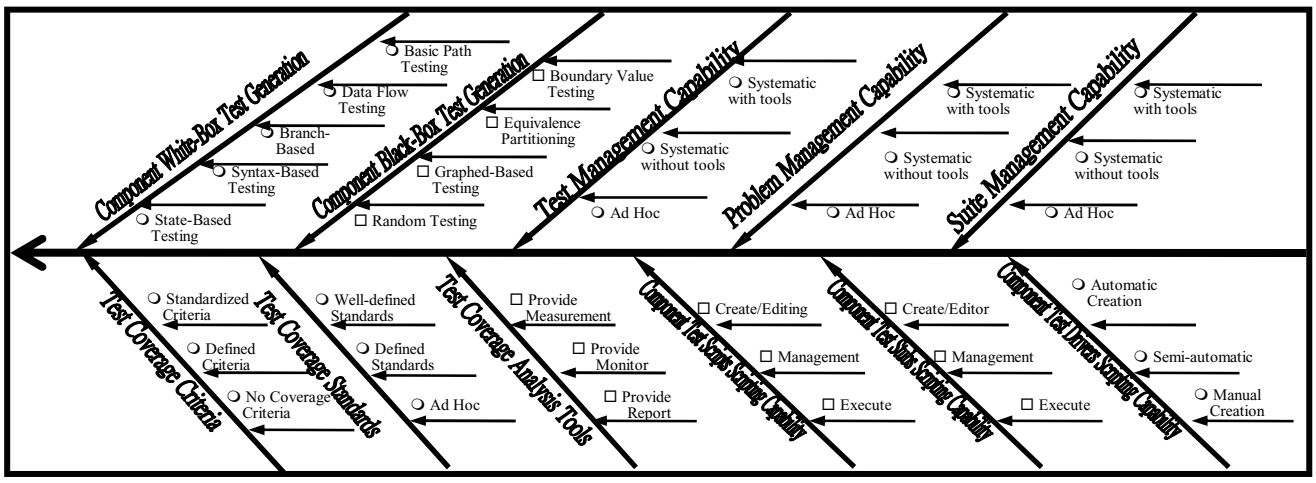

Figure 5. Factors of Test Support Capability

3.4 Component Traceability

Component traceability is the fourth factor of component testability. It refers to the indicator that represents how well a component is developed to facilitate the monitor different types of program behaviors. As shown in Figure 4, it can be further examined from two perspectives: a) built-in program trace capability, and b) completeness of different types of program traceability. Built-in program traceability can be measured based on checking the four properties of a built-in solution for program tracking. They are:

1. Trace Format Consistency: It refers to how well program traces are formatted. This factor can be verified, validated, and measured by checking the formats of program traces generated from a component. While checking a component, we obtain three types of results: ad hoc, consistent, or standardized formats.

2. Trace Code Insertion: It refers to the ability of inserting or deleting trace code into software components. This could reduce the workload for engineers to increase the traceability and the cost for the maintenance of component-based software. The solution for trace code insertion can be categorized into four levels: automatic wrapping, automatic code insertion, framework-based insertion and manual insertion.

3. Trace Generation: It refers to the flexibility on selecting and enabling (or disabling) different component trace types. Since each trace type has its special usage on testing and maintenance of a component-based program, it is important to provide diverse trace types and facilities for engineers to use according to their needs. Component trace generation flexibility can be verified, validated, and measured based on three classes: automatic, semiautomatic, and manual.

4. Trace Collection: It refers to the ability of collecting program traces in a component. The trace collection capability can be verified, validated, and measured based on three levels: automatic trace, semiautomatic, and manual.

5. Trace Storage: It refers to the ability to allow users to manage and store the generated program traces by a component. Component users and engineers need this capability in debugging, system testing and maintenance to help them manage and monitor various types of component trace messages. This can be verified, validated, and measured based on three types: trace file repository, trace data repository, and trace report.

Figure 4 shows that the completeness of program traceability for components can be further refined based on five types of program tracking capability given in [13].

1. Operation Tracking: It refers to the ability of recording the interactions of component operations. Also it can be further categorized into three types: (a) internal function tracking, which only supports for internal component function invocations, b) external function tracking, which only supports interactions between the function interactions with external entities, and c) all function tracking.

2. Performance Tracking: It refers to the ability of recording the performance data and benchmarks for each function of a component in a given platform and environment. Performance tracking is very useful for developers and testers to identify the performance bottlenecks and issues in performance tuning and testing. Performance tracking can be further looked at in six areas: throughput, reliability, speed-up, availability, scalability, and utilization.

3. State Tracking: It refers to the ability of tracking component state-based behaviors, including states and transitions. It can be further categorized into three levels: state tracking, transition tracking, both state and transitions.

4. Event Tracking: It refers to the ability of recording the events and their sequences that have occurred in a component. It can be further categorized into three types: a) GUI event tracking, which tracks user interface events, b) communication event tracking, which track incoming and outgoing communication messages, and c) interaction event tracking, which supports tracks event messages from one component to others.
5. Error Tracking: It refers to the ability of recording the error messages generated by a component. It can be further categorized into three groups: error tracking, exception tracking and warning message tracking.

3.5 Component Test Support Capability

Component test support capability is the last factor for component testability. Unlike the previous four, it only can be validated and measured during a component test process, and it focuses on the test operation support capability during component validation. It can be further refined into four types. As shown in Figure 5, they are:
1. Test Generation: It refers to the extent to which component tests and test scripts can be generated by using systematic test generation methods and tools. There are two types of test generation: white-box and black box test generations. White-box test generation can be categorized into five types: basis-path-based, dataflow-based, branch-based, syntax-based, and state-based. Black-box test generation can be also categorized into five types: boundary-value-based, equivalence partitioning based, graphed-based, random-based, and requirements-based.
2. Test Management Capability: It refers to how well a systematic solution is provided to support the management of various types of test information. There are three common test managements, which are problem, test and suite management. Each of them can be evaluated at three levels: systematic with tools, systematic without tools, or ad-hoc.
3. Test Coverage Analysis: It refers to the extent to which the component test coverage could be easily measured, monitored, and reported based on a selected test criterion. In order to increase this capability, engineers need two things; the first is a set of well-defined component coverage criteria and standards, and the other is the test coverage analysis function with testing tools that provides measure, monitor and report.
4. Component Test Execution Capability: It refers to how easy component tests can be executed. This capability is essential to component testing and evaluation for users.

4. Component Testability Verification and Measurement

In the past years, researchers began to look for the solutions to check program testability. J. M. Voas and K. W. Miller [2] present one verification approach to checking software testability after program implementation. Other researchers also realized verification methods useful during component development. As pointed out in [1][6], it may be too late to check component testability after component implementation. Hence, it is ideal to verify component testability during the earlier phases of a component development process.

4.1. Component Testability Verification

Based on the proposed component testability model, it has the potential to verify component testability during the component requirement analysis and design phases. The approach is to perform software verification and reviews for all generated component artifacts by checking component testability. To carry out the technical reviews, well-defined testability verification guidelines and standards need to be established based on the proposed model. The general procedure for component testability verification can be carried in different phases.

During a component specification review, engineers can evaluate the component understandability by checking its five factors based on a well-defined scoring scheme. In general, they need to answer the following questions:
1. What is the availability of required component artifacts for the given component?
2. How well the component artifacts are written to help component users' understanding?
3. Is the given component can be easily used based on its user interface and API reference manual?
4. How easily the given non-functional requirements can be measured?
5. How easily the given functional requirements can be tested?

To evaluate component observability, engineers must check its three factors as follows.
- **Checking Communication Observability:** They check if the requirements specify the capability of observing its incoming and outgoing messages. In addition, they should be able to check if the requirements specify the required capability of supporting its communication observation.
- **Checking GUI Observability:** They check if the requirements include the ability of observing GUI input and output data/event. In addition, they can also check if the requirements describe the required support for component GUI observation.

- **Function Observability:** They check if the requirements include the ability of observing component function in terms of its input and outputs. Moreover, they should check if the requirements describe the supporting capability for function observation.

To evaluate component controllability, engineers need to check its five factors as follows.

- **Checking Environment Control** to see what requirements are given for component environment control, and which type of environment control capability is specified.
- **Checking execution control** to see what requirements are given for component execution control, and which type of execution control capability is specified.
- **Checking state-based behavior control** to see what requirements are given for controlling component state-based behaviors, and what kinds of control features are required.
- **Checking test control** to see what requirements are given for supporting test control, and which kinds of test control functions are specified.
- **Checking function control** to see what requirements are specified for supporting function controllability.

To evaluate component traceability, engineers check component requirements in two following perspectives:

- **The required component tracking capabilities**, including error, function operation, performance, event, state tracking.
- **The specified built-in component tracking solution**, including component trace format and generation, trace code insertion, and trace collection & storage.

Clearly, in the earlier phases of a component development process, checking component test support capability only concerns the test planning and requirements for component validation.

During design phase engineers can check component testability by perform component design review and inspections to achieve the following two objectives:

- All component requirements for testability are traceable to component design.
- All required component capabilities for testability are correctly designed.
- All required solutions for testability are properly provided in the component designs.
- All component API interfaces can be checked based on the given component testability factors to see if they are well designed to achieve good testability. More detailed procedure and step-by-step guidelines are needed to support engineers to perform this.

Due to the limited space of this paper, we report our recent findings in the future publications.

4.2. Component Testability Measurement

As we mentioned in Section 2, although a number of proposed approaches have been proposed for program testability measurement, most of them are program-based. The rest of the approaches only can be used during the program-testing phase. Now practitioners in the real world are looking for more cost-effective ways to evaluate and measure component testability before and during component validation. In this section, we propose a new way to measure component testability, known as testability pentagram model. This approach can be used based on the component verification results during component analysis and design. As shown in Figure 6, component testability is measured and presented using a pentagram diagram based on the measurement of its five factors in the proposed component testability model. Each factor is measured based on the verification results of their refined factors during component verification in the earlier phases of a component development process. Detailed procedures and guidelines for component requirements verification and design inspection are given in [11]

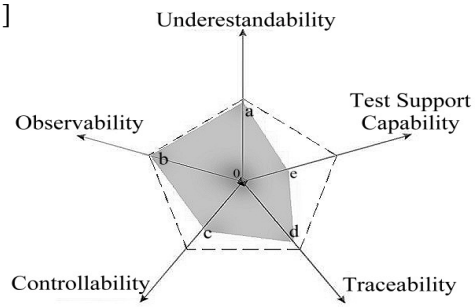

Figure 6. Testability Pentagram Model

Let's assume the measurement result of each factor is a value from 0 to 1. The value "1" indicates the maximum value for each factor, and "0" indicates the minimum value. The area of the pentagram is used as the measurement of component testability. Clearly, the smallest value of this pentagram area is 0, and the maximum value is approximately 2.4. Since the pentagram consists of five triangles. The area of each triangle can be computed 0.5 x l_1 x l_2 x sin λ as: where l_1, l_2 represent the sides of the triangle, and λ represents the 72-degree angle between the two sides. The letters a, b, c, d, and e in Figure 6 are used to represent the five factors of component testability respectively. When each factor is measured, then, component testability can be computed below:

Testability =

$$\frac{1}{2}\sin 72^{\circ}(ab+bc+ce+de+ad) \cong \frac{1}{2}\times 0.951\times(ab+bc+ce+de+ad)$$

$$\cong 0.48\times(ab+bc+ce+de+ad)$$

5. A Case Study and Application Examples

A case study has been conducted to check the feasibility of the proposed testability model for components in the development of a software component. The component selected for the case study is a Binary Search Tree component developed by students of a graduate software testing class. There are two purposes for this case study. The first is to check if the proposed component testability model is useful in measure component testability. And the second is to check the testability improvement of a component between its two versions.

During software verification, we collected: check lists, evaluation tables, metrics, and pentagrams, are used to verify and measure software component testability based on the given component function requirements specifications, and API interface specification. The collected results are listed in the following table.

	UMR	OMR	CMR	TMR	TSMR
Version1	0.52	0.6	0.03	0.01	0.08
Version2	0.75	1.0	0.78	0.46	0.66

Note: UMR, OMR, CMR, TMR, and TSMR respectively represent the understandability, observability, controllability, traceability, and test support capability metric of its requirements verification

Applying testability pentagram model for the five factors, we have the followings results:

Testability (Version 1) =

0.48 ((0.52*0.6)+(0.6*0.03)+(0.52*0.01)+ (0.01*0.08)+(0.03*0.08))=0.16

Testability (Version 2) =

0.48* (0.75*1+1*0.78+0.75*0.46+0.46*0.66+0.78*0.66) = 1.29

Based on the component requirements verification, the measurement results of component testability for both versions of the Binary Search Tree component are shown in Figures 7. It is clear that the testability of version 2(Right) is higher than the testability of version 1(Left).

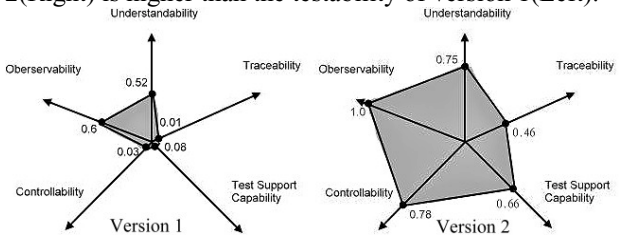

Figure 7. Comparison

6. Conclusions

This paper proposes a component testability model to facilitate testability verification and measurement during a component development process. Unlike the existing research work, our intention is to develop a component testability model to assist engineers to measure how well a component is constructed to facilitate component validation from requirements to testing. Although it presents our preliminary work in component verification and measurement by applying some examples, there is a potential to develop a more systematic solution for component verification and measurement before component testing if more engineering guidelines are available for requirements verification, design review, and program inspection. Hence, the focus of the future direction of this research is related to detailed testability verification guidelines for requirements analysis, design, and coding. Developing an automatic testability analysis tool is also interesting future project.

7. References

[1] Jerry Gao, J. Tsao, and Ye Wu, *Testing and Quality Assurance for Component-Based Software*, MA: Artech House.

[2] J. M. Voas, and K.W. Miller, Software Testability: The New Verification. *IEEE Software 12* (3), 17-28, May 1995.

[3] S. Jungmayr, *Testability Measurement and Software Dependencies: Proceedings of the 12th International Workshop on Software Measurement, October 2002.*

[4] R. S. Freedman, Testability of Software Components. *IEEE Transactions on Software Engineering, 17(6),* 553-563, June 1991.

[5] R. V. Binder, "Design for Testability in Object-Oriented Systems", Communications of the ACM, pp. 87-101, September 1994.

[6] S. Jungmayr, "Reviewing Software Artifacts for Testability" Proceedings of the EuroSTAR '99, Barcelona, November 10th - 12th 1999.

[7] Jin-Cherng Lin, Ian Ho, and Szu-Wen Lin, "An Estimated Method for Software Testability Measurement", Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97), 1997.

[8] Jin-Cherng Lin and Pu-Lin Yeh, "Software Testability Measurements Derived From Data Flow Analysis", Proceedings Of 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98), Deqli Affari, Italy, March 8-11, 1998.

[9] Antonio Bertolino and L. Strigini, "On the Use of Testability Measurement for Dependability Assessment", IEEE Transactions on Software Engineering, Vol. 22, No. 2, pp. 97-108, February 1996.

[10] Jerry Gao, Eugene Zhu, and Simon Shim,. "Monitoring Component Behaviors for Component-Based Software" *Journal of Object-Oriented Programming, October/November, 2001.*

[11] Ming-Chih Shih, "Component Testability Verification and Measurement", Master Thesis, August, 2004.