Abstract— Component-based software engineering (CBSE) today offers rapid development of system using fewer resources and efforts. The CBSE gave the idea of reuse and cutting down the development cost. Thus component based testing becomes more complicated if we have not reliable and bug free components. We can find the reliable and bug free components only if the components are tested by a suitable testing tool. In this review paper our aims to find the existing component testing techniques and tools in CBSE. This systematic literature survey is based on International Journals collected from multiple-stage selection process. These journals have been published within the time span of 1999-2010.

Keywords— CBSE, CITO, CASCAT, MORABIT, NHPP

I. INTRODUCTION

Component-based software engineering (CBSE) is a branch of software engineering that emphasized the separation of concern in respect of the wide-ranging functionality available throughout a given software system. It was a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems.

In Component-Based Software Engineering both customers and suppliers has expected much from CBD, but their expectations have not always been fulfilled. Experience has been shown that component-based development required a systematic approach to focus on the component aspects of software development. Traditional software engineering disciplines must be adjusted to the new approach, and new procedures must be developed. Component-based Software Engineering (CBSE) has become recognized as a new sub-discipline of Software Engineering.

The major goals of CBSE are:

- To provide support for the development of systems as assemblies of components.
- To support the development of components as reusable entities.
- To facilitate the maintenance and upgrading of systems by customizing and replacing their components.

The building of systems from components and the building of components for different systems requires established methodologies and processes not only in relation to the development/maintenance aspects, but also to the entire component and system lifecycle including organizational, marketing, legal, and other aspects. In addition to specific CBSE subjects such as component specification or composition and technologies, there were a number of software engineering disciplines and processes, which require specific methodologies for application in component-based development. Many of these methodologies were not yet established in practice, some not theoretically sufficiently refined. The progress of software development in the near
future will depend very much on the successful establishment of CBSE and this is recognized by both industry and academia.

Component-based development usually decreases the development time and effort, but also the possibility of guaranteeing extra-functional properties. For example, the main problem when using commercial components in safety-critical systems is the system designer’s limited insight into the safety-critical properties of components. Increasing the number of test cases may decrease this uncertainty. They also need specific test methods to be applied to components. One way of performing tests was to use fault injection which can reveal the consequences of failures in components to the rest of the system. As in general, the trustworthiness of commercial components was less than that of software developed in-house, they must perform tests as much as needed, but not more. If a component was extensively tested in one configuration, do we need to repeat all the tests performed or can we assume some of the results of previous tests? Must we add new tests? This depends on the system requirements and on the system configuration. By reasoning about changes in requirements, changes in the system environment and changes in the entire environment in which the system is performing, we could do some extent ascertain which test cases are already covered by the previous tests.

Component-based real-time systems (systems in which the correctness is also determined by time factors), and hence real-time components, must take into consideration timing constraints. Very often these systems are dependable systems (i.e. reliable, robust, safety-critical, etc.). General-purpose component models do not provide real-time support. There were many open questions how to build component-based real time systems: what is real-time component, what are its properties, how a real-time component can be specified.

II. LITERATURE REVIEW

Hoijn Yoon and Byoungin Choi [1] proposed that component customization failures by proposing the inter-class test technique between the black box class, which represents the ‘implementation’, and the white-box class, which represents the ‘interface’ for component customization failures. Their proposed test technique was based on a fault injection technique where a fault was injected into the ‘interface’ of the component. For their technique, they first extracts the component customization patterns and fault injection targets. The authors then defined the fault injection operators, which are applied to the fault injection targets. Since the fault injection operators can cover all possible failures that can occur within component customization, the proposed testing technique was suitable for component customization testing. Wu et al. [2] proposed that component-based software engineering was increasingly being adopted for software development. Although more work has been proposed for building component-based systems, techniques for testing component-based systems have not been well developed. The author has presented a test model that depicted a generic infrastructure of component based systems and suggests key test elements. The test model was realized using a Component Interaction Graph (CIG) in which the interactions and the dependence relationships among components are illustrated. By utilizing the CIG, they proposed a family of test adequacy criteria which allowed optimization of the balancing among budget, schedule, and quality requirements typically necessary in software development. The methodology proposed was efficient and effective, as demonstrated by promising results obtained from a case study. The author has presented a new approach for testing component-based software. Their empirical studies show that testing component-based software was necessary yet expensive. The technique they proposed includes several criteria for determining test adequacy. The all-context/ some-content dependence criterion used only 41% of test cases yet detected 84% of the faults; even the weakest criterion, the all interface, used 26% of test cases and detected 26% of the faults. Their method, which can be applied to all types of component-based systems, does not rely on the knowledge of source code.

Marlon Vieira and Debra Richardson [3] proposed that component-based development has become an important area in the software engineering field. In spite of this, there has been little effort to understand and to manage the different forms of dependencies that can occur in systems built from components. Dependencies reflected the potential for one component to affect or be affected by other components that compose the system. Understanding dependencies was an essential requirement to perform important tasks, such as evolution and testing, during component-based system’s life cycle. The author presented a technique to analyse dependencies in large component-based systems. Components communicate, share information, and depend on each other in a CBS. Identification of the dependencies embedded was the key to checking the semantic integrity of CBS. Therefore, it was important to research scalable and flexible ways to apply dependence analysis over large and complex CBSs. The author believed that their approach was scalable and able to give a broad idea of the system interaction network, thus facilitating analysis of system dependencies. Also, the product of their approach (the CBDM) was capable of helping during system maintenance, including evolution and testing.

Maxville et al. [4] proposed a process for selecting and evaluating candidates for component based software engineering. The process was aimed at developers sourcing components from third party vendors. Component metadata and a formalized specification of the ideal component, including context information, were used to drive the process. This specification was used to shortlist candidate components from commercial repositories and to generate the tests and adaptations for the candidate components. Metrics from each stage of the selection and evaluation process were then combined to compare and rank components for inclusion in the target application. Their approach to component selection, using context information and formal methods, helps to
address issues with component sourcing, selection and testing of third party components. The author has outlined their process for selecting and evaluating third party components. Use of third party components was hindered by such issues as how to source, select and test candidate components. Application developers need to be confident that they have the most suitable component for their system. This approach was aimed at developers sourcing third party components from external repositories. Such components came with varying levels of documentation. Their process provides a systematic approach for sourcing and selecting components. Automation of the process will save time, allow for a wider field of components to be considered, and gives traceable reasons for any choices made.

Aynur Abdurazik and Jeff Offutt [5] proposed that during component-based and object-oriented software development, software classes exhibit relationships that complicate integration, including method calls, inheritance, and aggregation. When classes were integrated and tested, an order of integration must be established. The difficulty arises when cyclic dependencies exist - the functionality that was used by the first class to be tested must be mimicked by creating “stubs” [sometimes called “mocks”], an expensive and error-prone operation. This problem was generally called the class integration and test order (CITO) problem, and solutions might be fully automated for integration and testing to proceed smoothly and efficiently. The author described new techniques and algorithms to solve the CITO problem. New results included improved edge weights that are derived from quantitative coupling measures to more precisely model the cost of stubbing, and the use of weights on nodes, allowing more information to be used. Also, a new algorithm for computing the integration and test orders were presented. The technique was compared with an existing approach with positive results.

Brenner et al. [6] proposed that component and service-based technologies play a central role in many aspects of enterprise computing. However, although the technologies used to define, implement, and assemble components have improved significantly over recent years, techniques for verifying systems created from them have changed very little. The correctness and reliability of component-based systems were usually checked using the traditional testing techniques that were in use before components and services became widespread, and the associated costs and overheads still remain high. The author presented an approach which addresses this problem by making the system verification process more component-oriented. Based on the notion of built-in tests - tests that are packaged and distributed with prefabricated, off-the-shelf components - the approach and supporting infrastructure helped to automate some of the testing process, thereby significantly reduces system testing effort. After providing an introduction to the principles behind component-based verification, and explaining the main features of the approach, the author shown by means of a small example how it can reduce system verification effort.

They have presented the MORABIT approach for run-time testing which can help automate much of the traditional verification effort involved in building software systems from components. The author have explained the basic rationale and philosophy behind the approach and demonstrated in terms of a case study how it helped to verify the correctness and reliability of component-based systems at both deployment- and service-time. Performing automated tests at the component level not only reduces the amount of traditional system testing needed to attain a given level of confidence in a system’s reliability, it also helped to pinpoint the location of faults, and thus reduced the effort involved in finding and removing them. The built-in testing technology developed in MORABIT therefore promised to be of particular benefit in the creation and integration of enterprise computing systems.

Liangli et al. [7] proposed that component metadata was one of the most effective methods to improve the testability of component-based software. The authors firstly gave a formal definition of component, and summarize the basic meanings of component metadata. Based on these, idea of Grouped-Metadata Object (GMO) introduced, which was divided into two types, respectively named descriptive metadata and operative metadata. And a general frame work of descriptive metadata and operative metadata which was consisted of several groups. Each group included several attributes. Furthermore, the author gave a reference model of GMO using class diagram of UML. Combining with the above formal model, the author presented changed model used in GMO and introduced an idea to map all changes component interfaces, mainly referring to changes of available public method and variables. Here the author introduced a concept of Method Dependency Graph (MDG) to implement the mapping. Then the changes were reflected in relevant attributes in GMO provided to component users in order to facilitate component-based software integration testing and regression testing.

Kong et al. [8] proposed that Algebraic testing was an automated software testing method based on algebraic formal specifications. It has the advantages of highly automated testing process and independence of the software’s implementation. The author applied the method to software components. An automated testing tool called CASCAT for Java components presented. A case study of the tool had shown the high fault detecting ability. The author explored the application of algebraic testing method to software components. A specification language CASOCC was designed. An automated EJB component testing tool CASCAT (Common AS-based Component Automatic Testing) was implemented. The approach has the following advantages. First, AS was independent of the implementation details, they were suitable for formal specification of software components. Second, as shown by the CASCAT testing tool, algebraic testing of components could achieved a very high degree of automation, which include test case generation, test driver automation and test oracle generation. Moreover, it allowed software testers to focus on a subset of functions and
properties of the component that they were used. Finally, the method could achieve a high fault detecting ability as shown by their preliminary experiment with testing a software component, which confirmed the experiments done by Doong and Frankl.

Machado et al. [9] proposed an integration testing method for component based software. The method was based on the widely used UML (Unified Modeling Language) notation, covered a complete integration testing process at a contractual component level and it was supported by the use of tools. Components and their interfaces are specified by using UML diagrams and OCL (Object Constraint Language) constraints. Software under test was built from composition of components in a standard component-based software development process. The author proposed a method that addressed the main issues in this kind of testing for component-based software with the goal of minimizing costs and maximizing the chances to detect faults. Fault detection could be maximized by effective test case selection, based on an adequate coverage, and test point selection that considered integration factors. The author considered coverage of execution scenarios in the context of the application under test that were thoroughly investigated to avoid omissions. The method has been successfully used to test a number of applications from small to medium size, including legacy applications. As a result, the author has obtained a suite of tests of reasonable coverage that reveal relevant faults and could be used in different iterations of the development process. Another advantage, as one could expect, was the considerable improvement on component specifications. Due to the support of tools, the effort to use this method has been reduced considerably.

Chen et al. [10] proposed that the reliability and security of software components inhibits the further development of component technology. Enhancing the testing ability of components was very important in components-based software engineering. The author proposed a testing approach of component security (TACS) based on a dynamic monitoring and detecting algorithm CSVD (component security vulnerability detecting) and discussed the dynamic monitoring mechanism, testing approach and detecting algorithm. In addition, Punylib.dll, a third-party component, was analyzed using TACS for its security analysis. The case study shows that TACS has good integrity, validity and better operability.

Weiqun Zheng and Gary Bundell [11] proposed a new contract-based software component testing (SCT) technique, Test by Contract (TbC), which extends the Design by Contract (DbC) concept to the SCT domain, and leverages it with UML-based testing at the modelling level to design model-level test contracts for UML-based SCT. The author introduced a new concept of Contract for Testability as the principal TbC goal, and developed a set of important contract oriented concepts (e.g. test contract, effectual contract scope, internal/external test contract), and useful test criteria for effective model-based testability improvement. A practical stepwise TbC working process was developed to show how to put the TbC technique into practice for contract-based testing activities to undertake UML-based SCT with a case study. Hou et al. [12] proposed an approach of component-based software reliability analysis which included the benefits of both time domain, and structure based approaches. Their approach overcome the efficiency of existing black box and white box techniques that fall short of addressing repair and internal system structures simultaneously. Their solution adopts a method of testing data transformation to cover both methods, and was expected to improve reliability prediction. There paradigm starts with consideration of component-based software testing process in view of the assumption of NHPP models. It accounts for software structures by the way of modelling the testing process. According to the testing model it builds the mapping relation from the testing profile to the operational profile of component-based software which enabled the testing data transformation to build the reliability dataset required by non-homogeneous Poisson process (NHPP) models.

Qiming et al. [13] proposed that the components of different platforms usually need their proprietary testing languages to execute test cases. Thus it was essential to proposed XML API-based test framework, which built on standard component and supports different languages test on cross platforms. They developed a general component extension test Interface (CETI) consisted of four kinds of function ports. Then on the basis of the traditional component interface mutation, it provided an XML API-based extension interface mutation testing approach which uses XML as test specification language. In addition, XACML- based API presented a technique for using mutation analysis to test the semantic correctness of component access control interface. The test framework has a lot of advantages, such as the multi-functional and visual testing interface, cross-platform and general-purpose test script language. The preliminary experiments shown that it could be used towards different application platforms, such as Linux, Windows and component testing environments, for example, CORBA, EJB and COM et al.

Zhongshef Qian [14] proposed that web testing was a promising technique to ensure the high quality of Web applications. Their work regards a Web application as the composition of different interacting components. A composite component consists of several other components. A Component Interaction Diagram (CID) was constructed from the specification of the Web application under test. The CID was described in XML fashion. For each component, a component automaton was established. According to the CID, the Component Test Sequences (CTSs) are generated. The CTSs was in fact the interacting sequences of the components. The test cases, which were individual XML documents, were the CTSs with input data and then the Web testing becomes the testing of component interaction automata. Finally, the evaluation of Component Test Sequence Coverage (CTSC) was performed and a method to increase the CTSC was presented.

Henryk Krawczyk and Adam Rek [15] proposed that a global network provides new opportunities for IT designers. It
allowed building applications from many pieces, known as components, distributed in different network locations. This methodology described how to build all system elements; business logic, data access and a presentation layer from the coarse-grained components. Such a possibility gave us the Java programming language and related open source technologies: Enterprise JavaBeans (EJB) for creating business components and Java Port lets for creating interface components. The presented approach shows how to manage relationships between components and their versions. It described methods to ensure reliable and fast communication between them. The authors presented platforms for building and testing automation of component based applications and explained how the component based approach could help to speed up the team work.

The component architecture perfectly fits the needs of modern organizations. Organizations were changing as fast as their business environment. Too long period of application development could make it useless. On the other hand too big haste may result in neglect of the design and testing phases and a low quality of the product.

III. CONCLUSION

The systematic Literature survey investigated existing component based testing techniques, understanding the behaviour of components and their interactions. The researchers also presented challenges in the area of component based testing. As seen from the year of publication of the articles, it can be seen that the researchers presented different testing approaches from the year 1999 to 2010. They used different testing tools to implement these testing techniques. They studied of exiting testing techniques and model for black box component based software systems. There also exists a need to establish requirements traceability and behaviours of evolving or changing components. The researchers also points investigated components in built in testing. There was good coverage in terms of research in understanding the behaviour of components, interactions and compatibility of components. The author contributed in the field of automated testing of components, testing at run-time and approaches to generate test-cases for evolving components.

IV. CONCLUSION

It has been seen that software components can be tested by applying different testing techniques in component based software engineering (CBSE). The future work can be done in the form of following:

(1) CBSE suitable when there is frequently changing requirements.
(2) Testing tool in CBSE.
(3) To achieves common component standardization and environmental characteristics.

In future work the testing of components can be done by using Net beans rich client platform. The better result can be found by using JUnit testing tool in Net beans Rich client Environment.

V. REFERENCES
