Software Component Integration Testing: A Survey

Muhammad Jaffar-ur Rehman(1), Fakhra Jabeen(1), Antonia Bertolino(2), Andrea Polini(2)

(1) *Centre of Software Dependability, Mohammad Ali Jinnah University, Islamabad, Pakistan*
(2) *Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy*

SUMMARY

Component-based development has emerged as a system engineering approach that promises rapid software development with fewer resources. Yet, improved reuse and reduced cost benefits from software components can only be achieved in practice if the components provide reliable services, which makes component analysis and testing a key activity. This paper discusses various issues that can arise at component integration phase. The crucial problem is the lack of information for analysis and testing of components externally developed, aggravated by differing perspectives of the key players in component based software development. Several component integration testing techniques have been recently proposed to provide a solution for those issues. These techniques are here surveyed and classified according to a proposed set of relevant attributes. The paper thus provides a useful and comprehensive overview as a starting point for investigation in the field.

KEY WORDS: Software Components, Component Testing, Integration Testing, Metadata

1. INTRODUCTION

In the latest years software developers are facing unrivaled challenges. On one side, information processing systems become increasingly complex, networked, pervasive, and also critical because more and more they are used in risky activities. On the other, the high competitiveness in software production causes an almost unbearable reduction of the time-to-market. As a consequence, developed software must retain maintainability, reusability, testability, and all the other *–ilities* related with software productivity, but at the same time it must assure a high dependability, or else the consequences can be catastrophic.

The most promising answer to these challenges relies on the potential to obtain complex systems by the composition of prefabricated pieces of software called *components*, following in this direction the example provided by other "more traditional" engineering disciplines. Indeed, *component-based* software systems are becoming prevalent as an engineering approach that empowers rapid development with fewer resources. Furthermore, the reuse of subsystems, which have already been tested in operation as part of earlier "successful" systems, should in principle grant a higher reliability. However, some laboratory experiments [1], and even catastrophic events [2] have soon warned that composing software components is not an easy task at all, and much work is necessary to enable this vision. As a consequence, a new, very active, research branch has been established inside the software engineering area, generally referred to as Component Based Software Development (CBSD), as testified by the escalation of devoted conferences (e.g., *CBSE Symposium, CD Working Conference*, *ICCBSS*, just to cite a few), journal issues [3,4] and books [5-7].

Actually, the very idea of producing software systems out of components is older than thirty years [8], but it is only recently that strong efforts are made towards the concrete affirmation of this methodology. Nowadays CBSD promises the benefit of producing quality software utilizing minimum time and resources.

CBSD is the process of assembling components to make them interact as intended. Each component can require pre-specified services from other components, exclusively through interface invocations. As said, CBSD facilitates software reuse and promotes rapid application development, but it is not without its constraints. For example, concealing proprietary information is one of its advantages; however, this privacy affects user understanding for subsequent component testing. In addition, assuring software reliability is made more challenging when commercial pre-tested software components are used. Crnkovic [9] provides a quite long list of research challenges related to CBSD, among which:

- **Modeling languages**: It is necessary to develop languages for modeling software systems at a high level of abstraction, neglecting those technical details that are not relevant at that level. In particular such languages should permit to describe the architecture of the system in term of coarse-grained components and of their required features.
- **Technologies**: It is necessary to develop suitable technologies to make easier components integration and their communication. In particular, technologies should provide high level services that permit to easily bind together components and make them cooperate, freeing the software assembler from networking and non-business logic details.
- **Composition Predictability**: Two different views can be considered for this requirement. The first involves inferring interesting properties of an assembled system, starting from the known properties of the composing components. The second view takes an opposite direction: inferring system properties starting from the study of the logical architecture of the global system.
- **Development Process and Tools**: It is necessary to better understand the steps that can lead to the development of a system starting from available components. Clearly, a highly iterative process is needed which permits to consider features of reused elements starting from the first phases of the development. This process will foresee phases peculiar to CB development, such as a provisioning phase leading to the identification of suitable existing implementations for the components in the architecture. Of course, it is equally necessary to develop tools that assist the CB developer along all the phases of the development process.

With reference to testing, it is particularly important to implement tools that make the testing of externally acquired components easier. Testing could, in fact, be fruitfully used for the evaluation and final choice of an external component, and this step could involve many different components. Indeed, if the rapid expansion of reusable software components requires on one side to ensure they provide adequately high reliability, on the other hand their integration must also be effectively tested. The well known failure of the *"Ariane 5"* launch vehicle [2], which veered off target and blasted in less than 1 minute after takeoff, is a prominent and unfortunate demonstration of the importance of integration testing, as the failure was attributed to insufficient testing of reused software component in the system.

Several testing strategies for effectively utilizing software components exist in the literature, however inadequacies in existing techniques occur due to component characteristics that are not amenable to traditional software development.

In this paper software component testing phases are discussed with a review of recent literature on integration testing techniques. Following this introduction, the paper is organized in two parts: the first overviews the field of CBSD, while the second part mainly focuses on survey of CB integration testing approaches that exist in the literature. Specifically, Part 1 covers the preliminary concepts and characteristics of components and their development process, followed by an introduction of component testing types and component testability metrics. CBSD brings a change to the development process, which further impacts the testing phases of component lifecycle. In Part 2, firstly the issues or problems specific to component testing are emphasized. Secondly integration testing techniques, existing in literature, are discussed with respect to these issues. Part 2 mainly provides an overview of the test elements in component testing phase, and an overview of various research contributions attempting to provide solutions for component integration testing.

Part 1

The first part presents an overview of basic notions, current trends, and topics in the wide area of CBSD. This part is not meant as a comprehensive treatment of the topic, but rather as a basis on which the survey of proposed approaches for component-based testing can be set up in Part II.

1.1 Trends in Modeling Software Systems

Objective of modeling is to provide an abstraction of a (physical) system, thus allowing engineers to reason about that system by ignoring extraneous details while focusing on relevant ones [10]. Software artifacts are probably the most complex systems that have ever been constructed, therefore [11] the potential benefits on the use of models are significantly greater in software than in any other engineering discipline. Indeed, as system complexity increases, the overall system structure becomes a more significant question than the choice of particular algorithms or data structures.

Our interest in modeling approaches stems from the fact that the information carried on by models constitutes the basis for the definition of useful test cases. In particular, in CBSD models become a key source for test cases derivation given the prevalent black box nature of components, especially of Commercial Off The Shelf (**COTS**).

Three interrelated fields of study strongly influence the modeling of CB software systems, namely:

- **Software Architecture**
- **Unified Modeling Language**
- **Model Driven Development**

1.1.1 Software Architecture

Software Architecture (SA) [12, 13, 14, 15] emerged in the last decade as one of the most promising instruments to derive, from the requirements, a formalized specification of the system in terms of components and their interactions. By abstracting away from implementation details, a good architecture description makes a system intellectually tractable. Many researchers are working on this topic and important industries have started to introduce SA.

In general terms, the architectural design of a software system is concerned with its gross structure and the ways in which that structure leads to the satisfaction of key system properties. Structural issues of interest for the SA description of a system include the composition of components, the definition of the global control structures, the definition of the protocols used for the communication, synchronization and data access, the assignment of functionality to design elements, the physical distribution of the component in the architecture, the scaling and performance feature of the system, the dimension of evolution of the system and finally the selection among different design alternatives.

From the architectural definition of a system it must be possible to identify the three basic elements that characterize a SA: **components**, the elements in which the logical computation is “located”; the **connectors**, the elements that mediate the interactions among the components; and the **properties**, such as pre/post conditions, signatures, and non-functional properties as well.

Hence, there exists a strict relation between the SA and the design of a CB system. In particular, among the aspects of software development in which SA plays a basic role [15], the architectural description provides a blueprint to the developers, in which the different components and the relations among them are shown. Moreover, architectural descriptions provide new opportunities for software analysis, also specifically for test derivation. Reuse is a goal, which joins both SA and CB. Reusing pieces of code is central to reduce the time to market in the development of complex system. In this area SA can play a central role grouping at a high level of abstraction the necessary functionalities and at the same time identifying the components that should provide such functionalities.

1.1.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) [16] is a standard language for specifying, visualizing, constructing, and documenting software systems (and non-software systems as well). At the same time, the UML provides a collection of best engineering practices that have proven successful in the modeling of large and complex systems. A detailed description of diagrams for the concerned readers can be found in the UML specifications official documents [16, 17].

In the upcoming version 2.0, the UML provides thirteen different diagrams that can be used to specify a software model. Figure 1, from [17], provides a graphical view of how these diagrams can be organized in two categories:

- **Structure diagrams** mainly represent static concerns about the system under development.
- **Behavioral diagrams**, by which the developer can specify behavioral concerns.

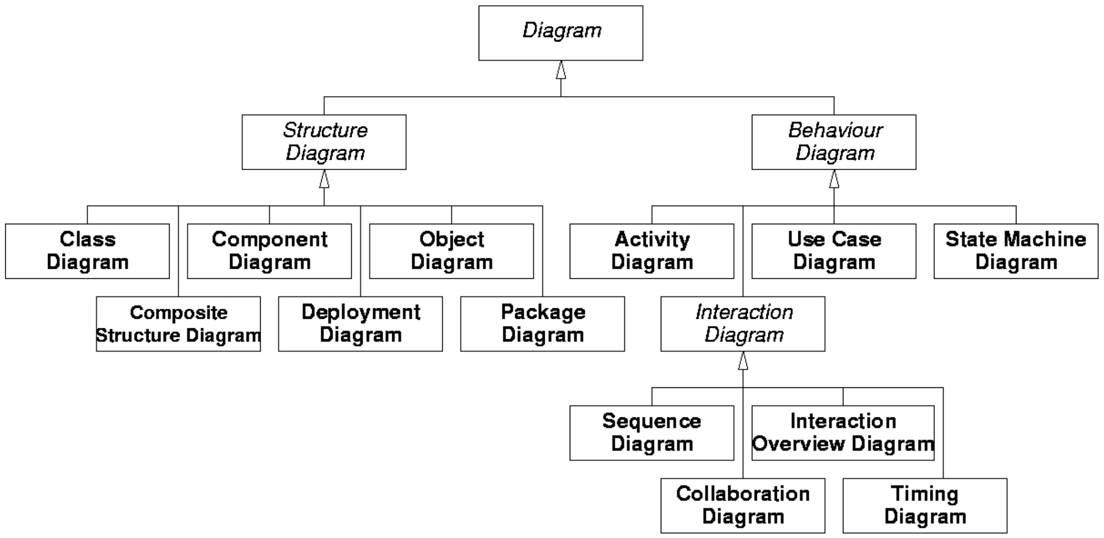

Figure 1: UML Structure and Behavioral Diagram

Besides, the UML provides three standard ways to extend the set of elements available: tagged values, constraints and stereotypes. Using them the modeler can use the UML predefined elements to create new elements with added semantics. An important tool for defining extensions is the Object Constraint Language (OCL) [18], that is a language defined by the OMG that permits to express logical constraint on the elements of a diagram. Another heavyweight alternative to extend the UML is to intervene at the meta level defining in such manner a UML like notation that however will not be recognized by a UML compliant tool anymore.

The UML has been initially defined to aid the development of Object-Oriented systems, however thanks to the extension mechanisms, several initiatives started trying to adapt/extend UML for use in different development fields. The increasing availability of tools for diagrams management, and the spreading diffusion of the language pushed other modeling communities, such as the SA community, to study the possible use of UML for architecture description [19]. A strong relationship in fact exists between the UML and the software architecture, being both two instruments for software modeling. However the use of the UML for SA description, even though it can appear quite natural, is not so obvious. In particular the UML lacks direct supports for modeling and exploiting architectural styles, explicit software connectors, and local and global constraints. Even though the upcoming version 2 of the UML provides a better support for describing connectors, such as ports and protocol state machines, there is not yet a complete inclusion of the software architecture abstractions.

A proposal for using the UML specifically for modeling CB software systems is “UML Components” [20], which defines particular stereotypes to represent software components. However, the “UML Components” proposal seems lacking for what concerns the support for architectural concepts such those highlighted by the SA community (e.g. connectors), and besides it can be considered superseded by similar concepts now embedded in the UML 2.0. Nevertheless, it provides an interesting iterative process for the development of CB software systems.

1.1.3 Model Driven Architecture

The most advanced frontier in the use of models in software systems production is today represented by the Model Driven Development approach. The vision underneath is the possibility of having models automatically transformed into the corresponding final executable code. As a result a new generation of languages and tools will be available to the system developer, providing a level of abstraction never experimented and, from an architectural point of view, the possibility of automatically verifying models that will be successively refined to become the final program.

The most concrete effort towards the realization of this vision is certainly the Model Driven Architecture (MDA) initiative. MDA is based on the use of the forthcoming version of the UML, based on a precise semantic so to enable the unambiguous transformation of software models.

In the MDA view the transformation between the different models is executed in automatic with the support of specific tools. There is not a wide number of tools enabling the MDA view, yet, however considering the benefits that the real success of this methodology to software production will bring it is not difficult to foresee that many tools will be released in the near future [21]. Among the several benefits that the MDA will bring in such as **productivity**, **maintenance**, and **documentation**, of particular importance towards CB development are certainly **portability** and **interoperability**, since these are at the basis for enabling the regular and smooth integration of different software components.

1.2 Technologies Enabling Component-based Development

The development of modern software systems would not have been possible without the great achievements in the area of software technologies of the last years. Indeed, the area of software tools and technologies to aid the development of complex software systems is far more mature than the correspondingly modeling languages and methodologies. Among the technologies defined to assist the implementation of complex software systems, two of them are at the basis of CBSD:

- **Middleware**
- **Component Models**

Whereby the middleware addresses interoperability and distribution issues, while component models focus on managing the reuse issues, and on defining rules for packaging and accessing services. However, these technologies are obviously intertwined and commercial products nowadays, such as, J2EE [22] and .NET [23] provide mechanisms for both categories.

1.2.1 Middleware

In parallel with the trend of modularization of complex systems into components, the last years have also witnessed an enormous raise of the demand for distributed software systems. Among the causes, particularly relevant are both the spreading of companies in many remote places, also as a consequence of the great number of companies merging, and the advent of the World Wide Web that led to the creation of "e-facilities", greatly impacting on the way in which services (by government or commercial companies) are provided and used. Moreover, some general non-functional requirements/features [24], such as Resource Sharing, Scalability, Heterogeneity, Fault-Tolerance, and Openness, lead to prefer a distributed architecture, instead of a centralized one.

The development of distributed software is far more complex than that of centralized one. For this reason it is useful to hide, as much as possible, to system developers the issues strictly related to distribution: this is the rational behind the concept of the *middleware*. As a general idea middleware is a kind of connectivity software that allows applications and users to interact with each other across a network. In particular middleware provides services that are generic across applications and industries, that run on multiple platforms, that are distributed, and that support standard interfaces and protocols [25]. For example a message switch that translates messages between different formats is considered middleware if it makes it easy to add new formats and is usable by many applications.

Components are implemented above the middleware layer. In this regard, the main task of middleware is to make transparent to the CB system engineer the new dimensions of complexity introduced by distributed systems. In particular [24], among desirable transparency features for CBSD, *Access Transparency* requires that the interface to a service does not depend from the location of the components that use it, and *Location Transparency* implies that a request for a service can be made without knowing the physical location of the components that provide the service, and thus allows for physically moving components across nodes.

1.2.2 Component Models

Component models refer to the mechanisms provided specifically for binding together software components. Basically these technologies provide a set of services that permit to produce and use components, by generally imposing a set of rules concerning the packaging of the component and sometimes requiring the implementation of specific interfaces, that will be used by the technology for management purposes.

Two main kinds of component models have been defined. The first, called **desktop components**, provide mechanisms that permit the integration of component deployed on the same system. This is the case of COM [26] and JavaBeans [27]. The second model, referred to as **distributed components**, provides mechanisms for integrating components that could be dispersed on more than one physical system. Distributed component technologies obviously rely on middleware technologies. The implementation choices made by the different component model technology providers can vary and in general it is not possible to take a component from one world (a component model technology) and deploy it into another world.

Apart from implementation details, the basic starting point for defining a component model is the *naming and locating* service, whose task is to provide, at run time, the components that need a specified service with the reference to that component that provides it. Through the implementation of a naming and locating service a component model permits the implementation of software elements that do not contain embedded references to the final providers of the required services.

Several different component models have been defined so far, three of which currently lead the scene: COM/.NET from Microsoft [26], CCM/CORBA from OMG [28], and EJB/J2EE Sun Microsystems [29].

1.3 Component Basic Concepts and Characteristics

So far software components have been discussed resting on the quite intuitive meaning that this terminology suggests. In this section, before diving into methods for component-based testing, which is the main topic, more precise definitions and characterization of software components are provided.

One of the most widely accepted and reported software component definitions is by Szyperski [5]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*

This definition covers different peculiar aspects of components. In particular it has a technical part, with aspects such as independence, contractual interfaces, and composition. A component can be plugged into a system as a unit of composition, with its services accessible via a defined set of interfaces. The importance of interfaces can be determined also from Brown's definition [30] of a software component as "*…an independently deliverable piece of functionality providing access to its services through interface* ". The dependencies of each component on other components or system must be clearly defined for accurate usage. The above definition also has a market related part, with aspect such as third parties and deployment. Each component is in fact deployed as an autonomous unit by organizations other than the developing organization.

Component vs. Object

To start with the characterization of a component, it may be worth to highlight the differences between the concept of a component and that of an object. A component is not a running element, as an object, but rather a static element that can be deployed in a system. An object, instead, generally requires services from other specific objects and this relation is embedded within the object itself. However, given this and other differences, it is true that components and objects are strongly interrelated and share many characteristics. Moreover, components are generally developed using object oriented concepts and languages, and at execution time they take the form of a collection of objects.

Source Code Unavailability

Software components are generally developed by a provider organization and used by multiple users, which do not necessarily belong to the same organization. The implementation of component interfaces is typically not exposed to component users, rather only textual abstractions are attached as interface specifications. The signature of each interface is clearly mentioned with an explanation of the component functionality without any implementation details. Thus component source code is hidden reducing development complexity for application developers at the component user end. The user is simply benefited by using component services without any concern for implementation details, and the provider holds the component ownership rights even after component deployment. Authors generally use the terms of black- gray- and white-box components with reference to different levels of closure of the component internal essence. In particular a black-box component does not disclose anything about its internal implementation, whereas at the opposite end side, a white-box component completely shows it to the user. In between, there may exist different levels of gray-box components depending on how many details are made public. The discussion about the opportunity of using one or the other of these different kinds of transparency is endless.

Service Granularity

The component interface defines the access point to a service provided by the component itself. The service that a component provides should be necessary to someone else, otherwise the component has no market and there is no reason in developing it. In this context also aspects of service granularity play an important role. The component should provide services sufficiently complex to justify its existence as a component. Simple components, in fact, can be more easily developed in house. On the other hand too complex services may reduce the market for the component [5] and therefore the convenience in developing it. The component granularity mainly affects [32] the development environment, in particular the production time and effort by the component developer. This characteristic can be assessed in terms of the interface descriptions, or the number of use-cases supported by the component.

Plug and Play

A software component is an autonomous entity with an inherent plug-and-play nature. It can be deployed in the system to provide services, brought offline, modified, and again deployed in the same system providing modified functionality.

Component Metadata

Metadata consist of augmenting the component with additional information in order to increase the component user's analysis capability. Carney and Long [31] categorized a component with respect to the organizational relationship between the component provider and user. In this categorization, a component can be a commercial item, a version of a commercial item, an existing component from an external source, or a component produced in-house. The categorization of components is defined by the level of modifications allowed in the component source and the artifacts attached with components in each category. This supports the importance of component metadata for the purpose of components maintenance attached with the components.

The motivations of metadata can also be understood by another research effort supporting the addition of component descriptions with component. Brereton and Budgen [32] gathered and analyzed disparate issues of component-based applications, in the form of a framework. According to them [32], reuse benefit of component based application leads to special issues that must be considered by component developers. Issues relating to software component as a product, the development process, and business issues of components are discussed in perspectives of software engineers being developers and integrators of component-based software applications. It is foreseen that integration of commercial off-the-shelf products and in-house components will likely shift the focus of software engineering from the "*specify, design, and implement*" paradigm towards the "*select, evaluate, and integrate*" one. In the latter paradigm, appropriate and explicit component descriptions are clearly needed to perform software engineering tasks.

Contract-Aware Component

In an ideal component world an interface should be completely characterized by a description that provides the system assembler with precise and complete information on the service that is implemented by the component, for instance using some formal mechanism. An interesting and quite successful way to associate semantic to an interface is the use of **contracts,** as advocated in the well-known design-by-contract paradigm [33]. A contract describes a service using first order logic and specifying conditions that should hold before the invocation of the services and conditions that will be true after the execution of the service. At the same time a contract can specify invariant conditions that remain true during the whole execution of the service. Contracts are a really useful mechanism in CB development, although this use can raise some problems, in particular when callbacks, i.e., the called service itself invokes back the caller, are considered.

1.4 Component Based Development Process

Every software product follows a particular process model. In the conventional waterfall model, software development adheres to definite sequential phases that commonly include the following**:**

- Requirements Analysis and Specification
- Design
- Implementation
- Testing
- Maintenance

The software development process uses the well-known “divide and conquer” strategy for solving large problems, i.e., a problem is identified and cut into small sub-problems, so that each sub-problem can be disentangled and solutions are assembled to form the system. The lifecycle phases of software products are controlled by a single organization; and when multiple organizations are involved they communicate and share product development information. Thus, bugs in software can be detected to some extent by using the artifacts of standardized documentation packaged with the software product to facilitate software testing.

CBSD does not follow this standard development process. In general, the typical waterfall process model phases may still apply, but the activities performed in each phase are varied and their relationship is changed. The same “divide and conquer” strategy is applicable to CBSD, but available components are explored in search of a solution, instead of building new ones. Hence the implementation phase mainly deals with the development of what is generally referred to as the *glue code*. This code is the necessary instrument to facilitate the correct interactions among the different components. The components instead are not generally implemented, but are looked for in the in-house repositories or on the market, through what is generally referred to as the *provisioning phase*. After one or more candidate components have been identified, it is necessary to evaluate their behavior when integrated with the other already chosen components. Obviously for this purpose testing plays a key role. In fact on the basis of the specifications for the searched component, testers can develop useful (functional and architectural) tests to be executed to evaluate each candidate component. A testing methodology should thus allow for the effective testing of a component by someone who has not developed it, and within an application context that was completely unknown when the component was developed. The CB testing stages are further discussed in the next section.

With regard to the requirements analysis and specification phase, in the new process the emphasis must be on the re-usability and interoperability of the components, and an important instrument is provided by the already discussed, SA. In fact, using the specification mechanisms developed for the SA, the structure of a system is explicitly described in terms of components and connectors. It is important to establish a direct correspondence between the architectural components and the run-time components. In other terms, the components forming the software architecture and the interconnections among them must remain clearly identifiable also dynamically, during execution. This feature, in fact, affects the quality of the system in terms of reuse, replaceability and then makes maintenance and evolution easier.

A distinctive feature of CB production is the co-existence all along the development process of several and new stakeholders. A CB process must in fact foresee and manage the spreading, in time and space, of different tasks among several uncoordinated subjects [34]. Hence, it is essential to define special engineering processes to assemble the building blocks in the complete system. An overview of CB life cycle processes embedding quality assurance models can be found in [35].

A general process model for CBSD has been presented by Dogru and Tanik [36]. Software system specifications are analyzed and modularized into sub-parts. For each subpart a component is searched from a set of already developed components. The components chosen from the available set are assembled to form the software system. If a valid solution is not found then either an existing component is modified or a new component is built for the sub-part requirement. The end product software is called a “*connected set of abstract components*”, and the end product consists of only the “*connectors and components*” [36]. The connector set represents the interrelationships identified during decomposition and carries on integration process information, which is useful in integration testing of components.

1.5 Software Component Testing Phases

Software testing [37] consists of the dynamic verification of the program behavior on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.

As for the other development phases, the testing stage as well needs a rethinking to address the peculiar characteristics of CB development [34]. The component provider and the component user roles can be compared as a minimum as yielding distinguished responsibilities for testing purposes. The first needs to test the software component in order to validate its implementation, but cannot make any assumption on the environment in which the component will be employed. The second, instead, needs to (re)test the component as an interacting part of the larger CB system under development. In [38], Harrold et al. addressed issues and challenges involved in analysis and testing in CBSD from both component provider and component user perspectives. Table I compares Harrold's [39] widely referenced perspectives, based on the distinct characteristics of software components presented in the next section.

Table I. Perspectives in Component Testing

Component Provider	Component User
Develops component	Uses component
Source code available	Source code unavailable
White box and Black box Testing	Black box of component unit White box testing of component-links or integrations with other components or system (as component interface names being invoked by the system are visible to the user, it can be termed as white box testing)
Context-independent view of component	Context-dependent view of component
All configurations or aspects of behavior of component must be tested	*Subset or all* configurations and only those aspects of behavior of component that are application related are tested
Performs unit test for insuring its correct functionality	Performs integration test by invoking component services
Low bug fixing cost	High bug fixing cost

Traditionally, the development of complex systems involves three main testing phases: unit testing, integration testing and system testing. Another testing level that can also be categorized as a component testing type is regression testing. In CB development, these three traditional testing phases have to be reconsidered and extended (see Figure 2).

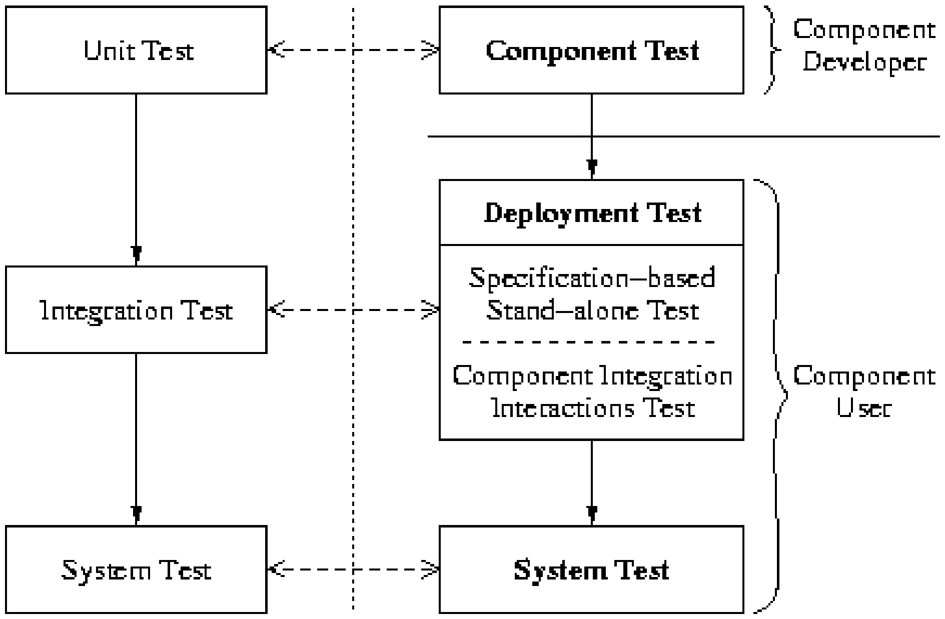

Figure 2: Adapting the Test Process

1.5.1 Unit/Component Testing

The smallest test unit becomes the component. Component testing is performed by the component developer and is aimed at establishing the proper functioning of the component and at early detecting possible failures. Software components are initially tested in isolation to detect errors in internal functionality of unit components. The availability of source code to the component developer permits white-box testing of all component configurations regardless of a specific usage context. The developer, however, must also perform black-box component testing to ensure that correct specifications are attached with the reusable component. However, such testing cannot address the functional correspondence of the component behavior to the specifications of the system in which it will be later assembled. In fact it is impossible for the developer to consider all the environments in which the component could be successively inserted.

1.5.2 Component Integration Testing

IEEE defines integration testing as "*testing in which software components are combined and tested to evaluate the interaction between them*" [40]. Indeed, unit testing cannot confirm the reliable behavior of components in a new system; hence another testing campaign—by the component user—is essential to attain an acceptable reliability level. Councill [41] termed testing by the component user during its implementation in the real environment **second party testing**.

Software components can be incorporated in a system as units, or multiple components may work together to provide system functionality, as shown in Figure 3. The component, whether integrated individually or with other components, requires integration testing--that is, testing component interactions as expected in the actual usage environment--before actual component deployment.

Component integration testing is an indispensable phase. Performed by the component user, the purpose of integration testing is thus the validation of the implementation of the components that will constitute the final system. This phase can be further divided conceptually in two successive sub-phases. In the first sub-phase the component can be tested as integrated in an environment constituted of stubs that roughly implement the components as foreseen by the specifications. In that manner it can be early checked whether the component correctly interacts with the "ideal" environment. In the second sub-phase, the real integration between several chosen components is verified. To do this, the interactions among the actual implementations of the architectural components during the execution of some test cases are monitored, and undesired interactions can be possibly detected. It is worth noting that potential mismatches discovered by the component user during integration testing are not in general ``bugs" in the implementation. Rather they evidence the non-conformance between the expected component and the tested one (and hence the need to look for other components).

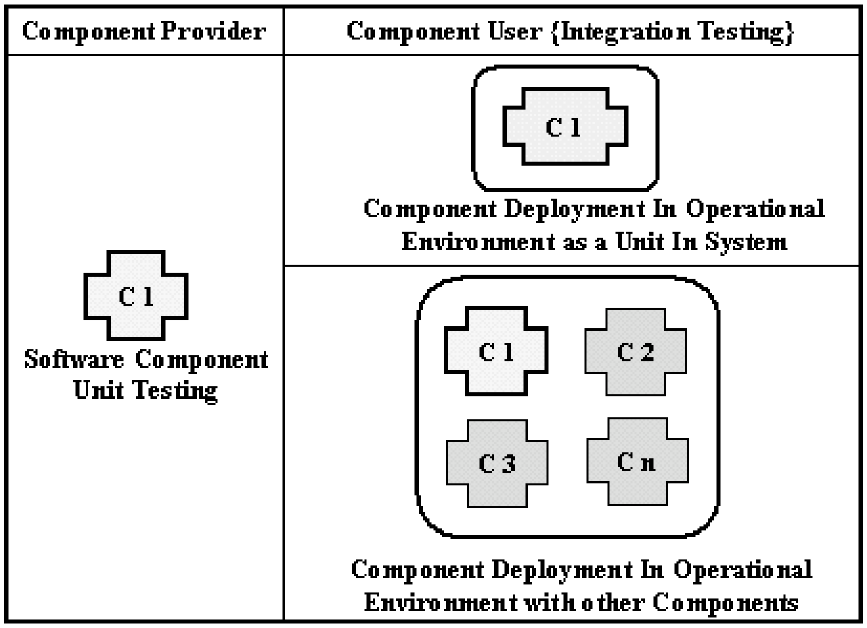
Component Provider
Component User {Integration Testing}
C 1
Component Deployment In Operational Environment as a Unit In System
C 1
Software Component Unit Testing
C 1
C 2
C 3
C n
Component Deployment In Operational Environment with other Components

Figure 3: Component Integration Testing by the Component User

A particular case of integration testing is when a real component comes equipped with the developer's test suite, and the customer re-executes those tests in his/her environment. These tests guarantee that the "intentions" of the developer are respected in the final environment and their execution generally lead to a more comprehensive evaluation. They can possibly include test cases not relevant for the customer's specific purposes, but that can be however useful to evaluate the behavior of the component under customer's unexpected entries.

1.5.3 System Testing

System test does not show major conceptual differences with respect to the traditional process (at this level of the analysis) and is performed by the component user when all the various components are integrated and the entire system is ready to run. System testing encompasses load and performance testing, to test the functionality of the whole system. The system testing process thus does not require white box testing of each component, even though it assumes that each component has been tested and attained a certain level of reliability. The emphasis of system testing is not only restricted to test components, rather, components are again tested in system context to assess the performance of the whole system as a black box.

1.5.4 Component Regression Testing

Regression testing is considered to be the first step in integration testing. According to Binder [42] rerunning accumulated test suites, as components are added to successive test configurations, builds the regression suite incrementally and reveals *regression bug*. The regression test suite does not contain tests for new or changed capabilities of modified components; however, the previously tested code must be tested again in a new context. The integration test suite requires the previously tested system code to re-execute not only in the context of the modified component but also the modified behavior of the component. It is for this reason that regression testing and integration testing are not synonymous. The already cited failure of the Ariane 5 rocket controller was in large part due to an assumption that previously tested code would work when it was reused, obviating the need for regression testing. Two main testing inaccuracies caused this failure. First, the components that interacted were not tested during the integration phase rather they were delayed for system testing only. Second, regression testing was not performed on the entire system. This single most costly bug in software history could have been detected by regression testing.

1.5.5 Standard For Software Component Testing

A generic software component test process is covered in BS 7925-2 standard [43]. The standard mandates the addition of specifications with the component containing results of all executed test cases. The defines 13 test case design techniques and 11 test measurement techniques. The test design techniques do not correspond to the test measurement techniques. An explicit standard compliance mechanism is not defined for a software component testing process to comply with the BS 7925–2 standard. However, it is assumed that generic testing process is followed and component specifications are also provided with component. The term component testing covers all testing types, including unit, integration, and system testing. Although a general process for component testing is defined in the standard, it lacks a description specifically for component integration testing.

1.6 Software Component Testability

Software testability provides the basis for evaluating the software application test process. Component testability is related to component testing, and provides essential metrics that can be used by component vendors and users to assess the component testability level. In the software industry, testable and reusable components are required to fulfill the immense demand of rapid and cost effective software development. To achieve this goal, component providers need more practical research results that provide valid methods and guidelines for developing testable software components. Component testability measures can help component users to select and evaluate components, and to gauge the effort spent on the component by the developer for increasing component testing capability. The IEEE Standard Glossary [40] defines software testability as**:**

> *"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".*

In this definition two different testability concerns can be identified. First, the way a software system is developed, and secondly, the requirements on the software system that serve as the basis for test suite definition. Beyond these aspects, however, component testability can be further extended to include the 5 metrics [44, 45] discussed below.

1.6.1 Component Understandability

The interface descriptions attached to components to explain their functionality also enhance their understandability, allowing component reuse, the major benefit of CBSD. Information attached to the component other than the source code provides a gauge to access the understandability metric.

1.6.2 Component Observability

According to Freedman [46] a software component is observable "*if distinct outputs are generated from distinct inputs*" so that, if a test input is repeated, the output is the same. If the outputs are not the same, then the component is dependent on hidden states that are internal to the component.

Hence, mapping the inputs with their corresponding set of outputs for each component interface can assess component observability. If component behavior can change internally, then it requires determining all possible outputs for one component input. Such information for each interface of the component enhances component observability.

1.6.3 Component Controllability

According to Freedman [46] component controllability is the ease of producing a specific output from a specific input. Component controllability refers to the mapping of component output from a specific set of inputs, i.e., the expected outputs are controllably produced from particular inputs. The normal execution of software components with definite inputs is checked to measure component controllability to produce the expected defined output.

1.6.4 Component Traceability

Software component traceability is classified as black box traceability and white box traceability. Component black box traceability can be assessed by a comparison of component behavior with its execution, while component white box traceability is established by checking the internal state of software components on every interface invocation. Gao et al. [45] explored and demonstrated the concept of traceable CBSD in distributed environments. The traceability metric involves various traces, which include operational, event, state, performance and error trace. These can be easily checked through detailed software component behavior and implementation testing.

1.6.5 Component Test Support Capability

Component test support capability involves the mechanisms to improve component testability. Mechanisms to generate test suites, to execute testing, and to manage the test process are required to improve the testing support capability of software components. Although it is the least explored metric, it is important in assessing the testability of software components.

Part 2

The second part presents a survey of several proposed approaches related to CB testing from the component user's view. Approaches for testing a component from the component developer perspective are purposely left out the scope of this paper, since the information available on the component internal structure (such as the source code) allows the developer to use different, white-box oriented, methodologies (see, e.g., [47]).

The approaches discussed cannot be considered alternative, on the contrary the combined usage of more than one of them can certainly give better results than only selecting one. Obviously the list is not exhaustive, but reflects the authors' comprehension and best knowledge of the literature.

Unfortunately, the same characteristics discussed in Section 1.3, which make components an attractive means for software production, affect the testing process negatively. In fact, typical component features such as source code unavailability, component heterogeneity, and continuously changing nature, are generally recognized as impediments to integration testing. Certainly, the major issue that affects integration testing is the lack of information. Commercially available software components are generally packaged without source code, and this may also cause difficulties in component integration testing [48, 49]. On one side the developer performs unit testing with no system context information (as also highlighted by Weyuker [50]) or knowledge of component deployment environment. On the other side the user chooses and deploys component in his/her application, by analyzing only the component published services but no implementation details.
Thus mechanisms for component integration testing in general demand additional information packaged with the component, and/or additional structure for reliable use of component applications, which is necessary for effectively utilizing the "*reuse*" benefit of components.

2.1 Coverage Notions for Component Integration Testing

The overview of CB testing approaches starts by defining some relevant coverage notions for component testing. One of the goals of integration testing is to ensure that messages from objects in one component are sent and received in the proper order and that messages have the intended effect on the state of components receiving the messages [42]. An operation may consist of a sequence of interface invocations, which further can be an external event or the result of a sub-invocation chain triggered by external event. Integration test coverage criteria are thus required to provide test coverage for the elements [48] including interface, event, context-dependence, and content-dependence. These four test-related elements stem from OO software developments, in which they provide the basis for defining specialized test coverage criteria.

Interface and Event Test Coverage

Software component services are accessible through their interfaces. The first integration test coverage criterion is thus to execute each interface at least once. An explicit definition of each component interface is generally provided through textual specifications, which include a description of interface functionality, its parameters, and return types.

At a higher level of coverage, the criterion may include testing each interface against all its possible invocations, which is equivalent to testing each event at least once. An event is a normal invocation of an interface. The event coverage criterion not only requires the invocation of each interface once, but also requires that each interface be invoked against all its possible events of invocations in the application environment. Many possible events can be generated from a component to fulfill this test coverage criterion particularly for a specific component environment. However it is still considered a weak criterion for integration testing.

Context dependence Test Coverage

Sometimes, events can have sequential dependencies on each other and the order in which they are triggered can result in distinct program behaviors. Such dependencies are termed as context-dependencies. The coverage of such dependencies could only be achieved by testing all possible operational sequences in the component.

Interfaces and possible sequences of their invocation in application must be known prior to testing the context dependence of components. The context dependence provides coverage for interoperability faults as direct and indirect interactions of software components are tested. Interoperability faults testing should be done of all possible interaction sequences. For example, an interface is dependent on another for its execution, which is itself dependent on another interface and the chain of dependence may continue in a similar manner. Operational dependence of each interface must be known for the context-dependence test coverage.

Content dependence Test Coverage

A still higher coverage level tends to test all content-dependence relationships. A content-dependant relationship exists between two interfaces I1 and I2 of a component if an operation of I2 modifies the value of a variable that is used in an operation of I1. In that case, interface I1 is said to be content-dependant on I2. This coverage corresponds to the data flow strategy for structural testing of traditional programs. An efficient testing mechanism requires coverage criterion for these dependence relations in OO components.

However, component interaction clearly affects the respective domain coverage. In [51], realizing the lack of theoretical bases for testing of CB systems, Rosenblum redefined test adequacy criteria for component unit and integration testing, providing a foundational reference notion for component testing. He formalizes the notion of test adequacy (with reference to a particular subdomain-based test criterion C) in the presence of a component *M* that is integrated in a program *P*. The study is based on the well-known Weyuker's axioms of *anticomposition* and *antidecomposition* [52], which are applicable only to programs simply containing sequential statements. In [51] the two new complementary definitions of **C-adequate-for-P** and of **C-adequate-on-M** are introduced. Informally, in CB testing, adequacy is reinterpreted over that subset of *M* input domain that can be ever invoked by *P*, denoted as "P-relevant". In fact, although it is *P* that is being tested in integration testing, the criterion *C* must be chosen and then evaluated in terms of the invocation relationship between *P* and *M*. The concept *C-adequate-for-P* characterizes adequate unit testing of *M,* whereby informally a test set T_M is *C-adequate-for-P* if it includes at least one test input from each P-relevant subdomain of *M*. The concept of *C-adequate-on-M* is instead defined to characterize adequate integration testing of *P* with respect to its usage of *M*, whereby a test set T_P is *C-adequate-on-M* if it traverses *M* with at least one input from each P-relevant subdomain.

2.2 Component Integration Testing Techniques: A Classification

The test inputs in integration testing must be determined in actual usage environment, i.e., integration testing requires knowledge of component usage environment. Hence, only the component user, or a third-party tester having this type of information, can conduct integration testing. The third-party tester may be provided with test information from both the user (component usage requirements) and the developer (component metadata, see Section 2.5).

For testing purposes, a component can be classified depending on the information that is carried on with the component itself. In this sense, a continuous spectrum of component types can be analyzed, at one extreme of which there are fully documented components, whose source code is accessible (for instance, in the case of in-house reuse of components or open-source components). At the other extreme of the spectrum there are components for which the only available information consists into the signatures of the provided services, which is the typical case of COTS components. Clearly, the testing techniques to be used by the component user will be quite different depending on the type of component. For instance, in the case of COTS, the unavailability of code hinders the possibility of using any of the traditional code based testing techniques. Components developed in-house may allow full access to component's internal representations and source code; however, in general, software components do not provide source code access.

A few testing techniques are defined for components that assume the availability of source code for example in-house components. Several testing techniques are defined for components packaged without source code; these techniques may further depend on the developer to append test information or rely on the user to extract test information from component. On the other hand, user and developer can also utilize the services of third-party tester to conduct testing, by providing component test information, and simulating specific component environment.

In the following component integration testing techniques are discussed. For exposition purposes, techniques are classified into five coarse categories:

1. Built-in Testing Approach
2. Testable Architecture Approach
3. Metadata Approach
4. Certification Strategy Approach
5. Customer's Specification-based Testing Approach

A more detailed categorization and comparison of the integration testing techniques is then provided in Table IV after having discussed the various approaches.

2.3 Built-in-Testing Approach

Built-in-testing is a generic approach for testing, earlier adopted in OO programming. In general, BIT is "the test software that resides in an application" [42], i.e., literally the component is augmented with executable test cases that are built in the component together with the normal functions. BIT requires component developers to embed tests in software component implementation to support self-testing. By running the embedded test cases, the component user can thus validate in the final environment the hypotheses made by the component developer. Several techniques have implemented this philosophy.

2.3.1 BIT wrappers

Edwards [53] has proposed a framework to provide BIT wrappers for component testing, using the specification language RESOLVE as an example, but other languages could also be adapted. The RESOLVE specification is assumed to include pre and post conditions for each interface, for which in RESOLVE "*requires*" and "*ensures*" keywords are used, respectively.

A component BIT wrapper is composed of two layers around the component. The internal layer is for handling component's internal errors while the external layer handles component's integration with system or other components. The wrapper adds the benefit of testing software component but diligent effort is required for managing the BIT code. The wrapper must be attached to the component in a way that the component behavior is not modified nor the component internal structure is affected, because this may put at risk the original objective of bounding wrappers around component: the key objective is to test component behavior and to reduce the behavior invalidity, not to introduce further complexity or making any changes in the structure of component. A wrapper is like a filter that permits the passage of only certain inputs and outputs to and from a component. The automation of the framework [53] relies on the RESOLVE specifications provided with the component. The interface specifications can be used to generate test-suite, test-driver, component BIT wrappers, and test oracles for test automation processes. The test-oracles are dependent on the BIT wrappers to be comprehensive enough to hold all the post-conditions. Component behavioral descriptions, i.e., specifications attached with the component are used to generate test cases and test data for black box testing. The test-oracles can also be derived from the specifications but the defect finding rate is improved by using the BIT wrappers, although using the wrapper approach increases complexity.

The BIT wrapper detects any interface violations inside the component or being generated by any invalid input to the component. However, a generalized approach is not presented rather it is restricted only for those components having RESOLVE specifications attached with the component. The specifications are necessary for framework automation, and in case of missing specification the programmer is required to generate the specifications manually by executing component and determining the component behavior. The generality of this approach is limited, as it is dependent on the provider to cater BIT wrapper definition and to provide component specifications. Besides maintenance issues need special care for components with BIT wrappers.

2.3.2 BIT in maintenance mode operation

The BIT approach had been used by Wang at el. [54] for enhancing CB software maintainability. They build tests in component source code as extra member functions; components in their approach operate in two different modes, which include the *normal mode* and the *maintenance mode*. In normal mode components perform the normal required functionality, while in the maintenance mode the component user can invoke particular methods, enclosed with each class constituting the component, that have been added to codify dedicated test cases. Being part of the class, these methods can access every private variable and invoke every method. So the mechanism provides the component user with a powerful means of evaluation, without requiring the component customer to use any specific framework or tool for testing purposes.

2.3.3 Self-Testable Software Component

Mechanisms to improve software component testability have been sought by applying the principle of design for testability in hardware, which relies on the addition of extra pins in integrated circuits to increase the observability and controllability. Similarly in software components additional interfaces can be provided with the software component to improve component testability.

Martin et al [55] used the Transaction Flow Model (TFM) as designed by Siegel [57] for creating self-testable components. TFM is used for unit testing of a class; message sequences for a single class are generated as a transaction starting from object instantiation to its destruction. TFM is generated for a class, and for each transaction, the Concat [58] test tool is used to generate test cases. The test cases once produced are reused, by applying different value of test data with the same test cases. The user accesses these test cases through methods built in the software component. The additional code for testing component is thus built and executed to find inconsistency in component implementation.

This self-testing methodology requires extra effort by the component developer to increase the component testability, and to maintain software component for reusability by multiple applications. The proposed approach results into complex code, which in turn may cause problems in component maintenance.

2.3.4 STECC (Self-TEsting COTS) Strategy

Beydeda and Gruhn [56] proposed the STECC strategy, which is similar to BIT as the software component is itself enabled to hold the information and the mechanisms to access the information, but differs in that it requires information from the component user for stub and driver generation. The test cases are first generated within the component to unit test it, while for stub and driver generation the user has to provide the input data for test execution. The test code in component can also perform test evaluation if the expected results are defined as test oracles in the component implementation, otherwise the results have to be manually verified. This technique relies on the component provider for defining an objective component testing process.

2.3.5 Built In Testing Approach Review

In general, the main advantage of the BIT approach is that it enhances the component testability and allows for easy maintenance. The proposed techniques in this approach however suffer some drawbacks. The first, and technical in kind, is that the memory required at run-time to instantiate the objects from a class, can become huge and mainly dominated by the need of space to allocate the testing methods; these, obviously, are completely useless in normal mode. The second, and more conceptual problem concerns the meaningfulness for the component user of developer's defined test cases. This approach depends on component provider to generate test cases for the software component. Even not considering malicious developers, only those aspects of a component can be tested that are enabled by the component developer.

BIT improves component testability but it is limited up to a certain level as it allows the developer to only provide static component test coverage criteria. As advocated by different authors [49], it is instead important that the customer develops his/her own test suites so to ascertain that a candidate component be “compliant” with the requirements for the searched component.

BIT is a generic testing approach, which could be further improved by the definition of standardized processes for heterogeneous components. In this way the BIT approach may not be restricted only to a specific type like OO components, rather it will be uniformly applicable to all component types regardless of their implementations. The definition of standard procedures for stub and driver generation can further enhance the BIT generality, and also facilitate the execution of component integration testing.

2.4 Testable Architecture Approach

This approach can be seen as a special case of the previously described approach, and in fact shares the same aims. However, differently from built-in testing, this one prevents the problem concerning the huge amount of memory required at run-time. The idea is that the component developer equips the component with a specific testable architecture that allows the component user to easily execute the test cases. The test information is appended by the developer in the form of specifications instead of enclosing them in the component itself.

2.4.1 Component Interaction Graph (CIG)

An effort by Wu et al. [48] is the definition of a test-model called CIG (Component Interaction Graph) for integration testing. As a first step test elements are identified to build CIG, and test cases are generated for each test element in the CIG. Test coverage criteria are then defined based on these test elements, which are accessed directly from component source code if it is accessible to the user. If component implementation is not visible then mechanisms are required to indirectly extract component test elements from component design information. Wu et al [48] define a set of mechanisms to access these elements in different types of component models such as EJB, CORBA and COM based components. The interface list is provided with every component regardless of component model type. Similarly events can also be accessed, and if not provided in the specifications the component developer can attach the interaction diagrams with the component. Otherwise a set of events can be generated exhaustively for testing purposes. On having the interfaces and events, an algorithm is designed for extracting context dependencies. For content dependencies Wu et al [48] require component developer to append the design information such as class diagram. CIG model defines the test coverage criteria to reveal interaction faults by exhaustively testing all test elements, as mentioned above in section 2.1. The results of the case study [48] also reveal that the test elements are useful in deriving test cases for integration testing of OO components. The technique however suffers a drawback, as it would allow the access to component source code through reverse engineering from design information. CIG model requires the developer to append the design information such as sequence diagram for easy test derivation. This information may allow the user to perform the reverse engineering task and get an access to source code thus affecting the implementation transparency, which is an important characteristic of software component.

2.4.2 Introspection Through Component Test Interfaces

In [59], Gao and coauthors require that each component implements a specific interface for testing purposes. This interface has the explicit goal of augmenting the component testability. In that manner the developer can then provide the component user with test cases coded in terms of clients that use the testing interface. By foreseeing the presence in the test-oriented interface of methods that use the introspection mechanisms, which are generally provided by component standard models, the same power of the built-in testing approach can be obtained in terms of access to methods and variables otherwise not visible to clients. The maintenance of such components is not simple, as it requires much effort on the side of component developer.

2.4.3 Contract-based Built-In Test Architecture

Another interesting approach also relying on the definition of a particular framework for component testing has been proposed by Atkinson and Gross [60]. Differently from the previous approach, in this case there is no use of the introspection mechanisms provided by the various component models. As a consequence the framework cannot reach the same power of the built-in testing approach. However this framework is not intended for the execution of generic test cases, but it focuses on providing the customer with specific test cases derived from contract specifications. In order to check the validity of a contract the authors suppose that a component developer implements particular methods for state introspection. In particular these states are defined at a logical level using a component model based on KobrA [61], a modeling tool developed in the area of Product Line (PL) design.

This approach allows the user to define component requirements in the form of contracts. The developer then builds the state introspection mechanism for only the interfaces specified by the user. This architecture thus facilitates the user for establishing confidence in component services. It also increases component reuse on the developer side, as the test specifications generated for one user may get reused with the same component being delivered to another user.

The implementation of this method is achieved through the use of modeling tool. Any user can test the required services by examining the logical definitions of states in the component model. This requires the use of these tools to trigger the state introspection mechanism in component. Hence, the technique does not present a generalized approach equally applicable for heterogeneous component environments.

2.4.4 Testable Architecture Approach Review

The testable architecture approach is slightly different from BIT and also solves some problems in BIT approach. Firstly the memory consumption problem is resolved, as the test specifications are built separately from the component source code. Secondly a few testable architecture approaches allow the user to specify the test requirements thus resolving the issue of static test coverage criteria to some extent. Some Testable Architecture approaches require the developer to define test cases according to the requirement specifications given by the user. In this way the test case derivation process to some extent involves user participation thus simplifying the component integration process.

This approach enhances component testability yet poses considerable challenges. The contract based testable architectures as discussed in section 2.4.3 are tool dependent, and so may not be applicable components without running the modeling tool. This limits the architecture only for those components that support the modeling tool. Similarly the approach cannot be generally applicable to all component types. Extra effort is demanded from developer largely at maintenance time, and most of techniques in this approach are not further valid for heterogeneous components.

The testable architecture must support test case derivation process according to user requirements but at the same time also preserve component implementation transparency, an important characteristics of software components. Component reverse engineering through component design information allows component source code generation. In the same way the UML dynamic diagrams in CIG technique (section 2.4.1) may allow the user to generate the source code from component design. The addition of such component design can affect the implementation transparency. For this reason the developer must equip component with such architecture that does not allow for component reverse engineering. That is the architecture supports the test case derivation process but does not allow the source code generation.

2.5 Metadata Approach

Among the several problems hindering CB testing process, the main issue is lack of adequate information about the acquired component. A lot of research work has attempted to solve this problem by attaching additional information with the component, without making available the source code. All forms of additional information appended with software component, either by the developer, or user or a third party tester, to facilitate software testing can be regarded as *forms of metadata*. In particular, to improve integration testing process by component user, metadata is defined as the information added with the component to increase the component customer's analysis capability and to facilitate component user testing.

Different kinds of information can be provided by the developer such as a Finite State Machine (FSM) model of the component, information on pre- and post-conditions for the provided services, regression test suites [62], and so on. The CIG (component integration graph), discussed in section 2.4.1, for example, can be itself seen as a form of metadata. The Metadata based component integration testing techniques as discussed in following subsections can be compared by the analysis of echelon of metadata appended with the component in each technique. It is important at the same time that suitable tools for easing the management and use of metadata are also developed.

2.5.1 White-box Components Analysis

Orso et al. [63] highlight the need for added information with component for testing and maintenance purposes. In their view, component implementation information is extracted and used as metadata by requiring the developer to enable run-time checking mechanisms. Three types of summary information are provided in this approach:

- *Program slicing* involves associating slices of component source with a set of variables. Program slicing is done to develop an understanding of a software component: program statements are divided into chunks of code (i.e., sub-domains) according to a variable use and summary information is bound with each chunk. A slicing algorithm [64] is used for backward traversal of program to compute the transitive closure of the data and control dependence in this approach. This transitive closure helps in establishing program understanding for analysis and testing. Program-slicing information is then included.
- *Control-flow analysis* associates predicates with each statement to hold true for the execution of statement. Errors that may occur in the component due to external environment are attached with those statements and can be reported by exception generation. At detailed level summary information can be provided with each exception containing information for set of variables and a state defined for the values of variables. This control flow analysis information is defined

for abstract component model, but needs to be redefined according to the requirements for a particular model.

- *Data-flow analysis* at lowest level provides the definition and use of only input variables, while at maximum level this information can contain the definition and use of each variable in the component, as defined in the program-slicing phase. After testing the variable's definition and use as the tester generates initial test suit. For achieving higher level of coverage, the tester can also generate test cases for those sub-domains, which are left uncovered by the data flow analysis. In this way all aspects of program are tested accomplishing high coverage level.

These types provide generally applicable information to abstract component models, and require redefinition for a particular model. The approach does not provide a practical implementation. However, it provides the starting work for recovering the missing information, and can be extended to supply comprehensive component testing information.

A format similar to MIME (Multi-purpose Internet Mail Extension) is proposed for metadata representation. The component metadata consists of information such as parameter types and state invariants of OO programs, in the MIME tags. The tags for this information are defined and can be filled at execution time. These tags need further improvement for complete definition of component behavior.

Also the proposal of Stafford and Wolf [66], who foresee the provisioning of pathways expressing the potential of an input to affect a particular output, can be re-conducted to the Metadata approach. Whaley et al. [67] propose to supply models that express acceptable method calls sequences. In this manner the customers can evaluate their use of the component and check whether legal calls are indeed permitted.

2.5.2 UML Test Model

With the widespread adoption of UML, UML based metadata obviously become an attractive means for the integration testing of CB software in industry.

Wu et al. [68] propose an approach for metadata in the form of UML models, which target the addition of specific test elements of software components by the providers. They define a set of adequacy testing criteria in which the UML interaction diagrams are used for extracting the test elements. The interfaces for components are outlined in the associated specifications and the events are also added in the form of virtual interfaces. Coverage of these elements is straightforward, but does not provide substantial confidence. The level of confidence can be improved, while still keeping the complexity low, by covering the context-dependence relationships. These relationships are modeled conveniently and directly in UML interaction diagrams. Execution of each possible sequence through these diagrams provides the desired reliability estimation, which can be enhanced still further by covering the possible sequences in the associated state-charts. On the other hand, the content-dependence relationships are not modeled directly in any UML diagram, however, the same can be extracted indirectly from collaboration and state-chart diagrams. The collaboration diagrams, that involve entity classes, can show the content-dependence relationship through *update* and *retrieve* messages. An update message only modifies a value (in entity class). This is modeled by a single message flowing into an entity class without any corresponding out-flowing message. Retrieve messages, on the other hand, read a value and are modeled by a pair of in an out-flowing messages. If the execution of an interface I1 involves an update message that has a corresponding retrieve message in interface I2, then I2 is content-dependant on I1. Similar to the use of collaboration diagrams, state-charts can also be employed to identify content-dependence relationships.

The proposed UML Test model provides a comprehensive technique for functional testing of third party components under the unavailability of source code. UML models abstract the complexity of integration of such components and provide only relevant information for test coverage of all test elements of software component. Automatic test case generation from UML models allow software component users to generate test cases from the UML artifacts attached by the component developers. On the other hand, the proposed test-model assumes that one interface corresponds to one operation of software component, but more in general an interface may abstract occurrence of multiple component operations, invocation of one interface in component results in the activation of several operations each with a specific task. The complexity of multiple operation invocation is not handled in the testing process. Wu et al [68] advocate providing UML diagrams with the component only for the context and content dependence coverage while interfaces and events can be outlined from the associated specifications. In addition, manual usage of this technique can be a very slow and tedious process thus automation is desirable. No tool, or any empirical evaluations currently support the work.

Redolfi et al. [65] defined a reference model in the form of class diagram. They take into account the characteristics and properties of components existing in literature for defining this reference model. This model does not directly expose the implementation part, however, implementation specification part in the class diagram allows accessing the component behavior as in the component source code, affecting the implementation transparency feature of component. The model is mainly used in defining a component repository so that a component can be easily accessed based on requirements of component functionality and behavior. Reuse, an important benefit of software components, is achieved through documentation of component properties. By enhancing component understandability, this reference model can also be classified as a form of metadata facilitating component testing.

2.5.3 Checking OCL Constraints

Another approach of metadata based testing is the one proposed by Brucker and Wolff [69]. This technique has been inspired from the design by contract principle [33] and based on it, attempts to generate components that have the capability of testing themselves. This is achieved by building some testing code into the component that executes and tests the component. The self-testing code is in the form of pre and post conditions on the methods that have to be checked on each entry and exit respectively. Violation of any of these conditions represents a bug. This self-testing information is derived from a class diagram, which is annotated with constraints specified in OCL. Once the components are built, these constraints (conditions) have to be added to it. This is automatically done through instrumentation tools already available. Similarly the execution of this code would require executing and checking the constraints that are still specified in OCL format even in the code. Again, tool support for OCL constraint checking exists. In addition to providing built-in test support in components, the work also defines some design patterns for distributed components and support them with a prototype tool.

2.5.4 Metadata Approach Review

Like the testable architecture approach, the metadata approach tends to enhance component testability. In this approach the component user is provided with component minimal details so that the user is allowed to generate test cases from the added metadata. The benefit of the approach is that the test coverage criteria are defined according to user requirements, thus eliminating static test coverage criteria problem. In this way component metadata assists in software engineering tasks such as testing and maintenance by providing analysis and testing information to the component user. Nevertheless there exists a need to define standardized methods to query and generate the metadata dynamically as for the large software components the design information or metadata becomes too complex to manage. The techniques in metadata approach generally do not handle heterogeneous components.

For the existing techniques, the format of component metadata can apparently be classified in some way that facilitates metadata understanding. This heavily impacts the effectiveness of these testing approaches. For instance Cechich and Polo [70] propose to use aspects to provide a more effective categorization and codification mechanism of component services.

Some of the techniques in this approach indirectly tend to affect the implementation transparency of the software component, similarly to the Testable Architecture approach. The design information appended as metadata particularly the UML dynamic structure diagrams (e.g., interaction diagrams) allow source code generation through component reverse engineering processes. In this way component metadata must support component analysis only for component testing and maintenance purposes and must not affect the implementation transparency feature of software component. Another improvement to the approach can be that the techniques be further explored so that the test information like test data and test cases be automatically generation from the metadata. This can further ease the integration testing process at the user end.

2.6 Certification Strategy Approach

The widespread use of software components can only be obtained by building a high level of confidence in the software component, but the customer of a component is generally suspicious about the information and proof of quality provided by the component developer. Indeed, the component developer cannot be entrusted to produce error-free components and moreover possible inconsistencies between the component and the accompanying metadata cannot be detected until the component user executes the software component and verifies the results. Hence, to increase more effectively the trust of a customer on a component, different forms of *component certification* have been proposed [70]. These forms mainly rely on the idea that each component undergoes a standard certification for the development process and for its features and architecture. In general, the component certification process is classified into the following types based on who takes the *role of certifier*:

- Third party component certification
- Component user certification
- Component developer certification

2.6.1 Third Party Component Certification

Third-party certification is an effective way to build component trust, because it can preserve the objectivity in component testing [40, 72, 73]. Once a component is tested by a third-party organization, multiple vendors can use it effectively by reusing the test-results as generated by the third party.

A first proposal [74] involved the constitution of independent agencies, or Software Certification Laboratories, with the main duty of deriving and verifying the qualities of a component. To this end the agency should extensively test (from a functional and performance point of view) the components using an impartial mechanism and then publish the results of the executed tests and the used environments. This can be more beneficial for large software systems, and particularly for safety critical systems.

Councill [40] states that third party testing can provide impartial component testing to component users, and it may be performed voluntarily, contractually required or enforced by legal permission. The results or findings of third party testing are provided to the requesting organization.

Ma et al. [73] have proposed a framework for third party testing of software components. A process is defined for third-party component testing using metadata attached by the component developer for this purpose. The specific elements of metadata are not identified, however it is assumed they generally include information according to the general philosophy of component metadata, i.e., analysis and test support information. The framework defines a three-step process:

- Third party tester provides guidelines and supporting tools to the component developer.
- Component producer generates a test-package using these guidelines. The test-package consists of information for deploying and testing the component and to audit the test suite of the component in the form of metadata.

- Third-party tester checks the conformance of test-package with their guidelines, executes the test package, and generates a test report.

An evaluation of third-party testing framework by Ma et al [73] demonstrates that third-party tester met some problems while executing the test suite provided by the developer. The objective of third-party testing framework was functional testing of a software component by a third party having some knowledge of the component, but the generated test package proved insufficient for the testers to understand and test component functionality. The problems encountered in the evaluation reinforce the need to attach comprehensive metadata by the producer, but the framework lacks an explicit definition and a formalized notation for metadata representation. Some discrepancies also occurred between part of testing information metadata and the software component. In addition component developer could not produce correct test oracles, as the developing team lacked expected testing skills, although guidelines were given to the developers for the generation of test package. These problems arose due to reliance on component producer for the generation of test package, whereas normally developing teams lack standard testing potentials. However, third-party testing framework aims at providing objectivity in component testing, and can be used effectively by the component users to rapidly verify and validate software components, thus increasing reliability in CBSD applications.

2.6.2 Component user certification

The inherent difficulties in establishing these third-part agencies suggested that in alternative warranties be derived as the result of extensive operational usage, following some notable Open Source example (e.g. Linux). By coordinating the users of particular software, a "user-based software certification" [75] could be established.

The component users can check that a selected component performs specified functionality for system by component user certification. It is also an element of integration testing. Voas [74] defines component user certification process by incorporating black-box component testing, operational system testing, and system level fault injection. Table II provides the objective of each testing technique used in the certification process.

Table II: Objectives of testing in the certification process

Testing for certification process	Objective
Black-Box Component Testing	To check component compliance with the specification, and to check component quality
Operational System Testing	To determine system's reliability with operational component
System-Level Fault Injection	To determine system's reliability while generating failures in the component

The component user performs black-box component testing to reveal errors. For black box testing, test cases are generated based on component interface specifications. If the component meets the quality requirements of component user then operational system testing of component is performed. In order to assess the reliability, faults are randomly generated, and instead of testing component with correct inputs, the system is tested with wrong outputs from the component. The wrong outputs generated and passed to the system allow the user to determine the undesirable inputs to the system. Those outputs of a component are revealed, which can adversely affect system. The application developer uses the component wrappers as a filter to such component outputs with the limitation that only expected outputs are filtered by the wrapper.

In this certification approach component buyer defines oracles for testing quality of software component hence an accurate set of oracles is utilized in the certification process. Wrappers designed by the application developer may not filter an erroneous output by a component, thus causing system failure. For this, component developer may append additional specifications with the component, which can assist finding such errors by the application developer, thus making the process more effective in finding the bugs in integration testing.

2.6.3 Component Developer Certification

A different approach to certification has been proposed by Morris et al. [72], starting from the remark that using the services of a certification agency could be particularly expensive for a small software company. To overcome this problem a developer's software certification approach is proposed, i.e., the component developer provides a certification that component functionality complies with the test specifications provided with the component, by testing the component's behavior and attaching testing information (metadata) with the component. It is therefore foreseen that the developer generates test cases and testing information, performs component testing and appends the testing cases and test results with the component as a proof of tested software component. This approach relies on the release, together with the component, of test case sets specified in a formal notation using XML (Extensible Markup Language) [76]. This format should guarantee a better understanding, on the component user's side, of the test cases that have been executed by the developer on the component.

According to the authors, on the one hand this should increase the trust of the customer on the component behavior. On the other hand, by using suitable tools, it should be possible for the customer to (re-) execute the XML test cases in the target environment. Thanks to this feature, the approach could also be seen as another variant of built-in testing.

The test specifications consist of a set of test cases generated for each interface. A Testset consists of multiple TestGroups, which further consist of operations. Each operation is a collection of method calls, thus to perform the operation testing each method call in operation is tested. To be precise, the test case for each operation requires executing a set of method-calls given in the operation. For OO components the invariant-element is specified, which must hold true from object construction, and all through operations in object life cycle till object clean up.

2.6.4 Certification Strategy Approach Review

The three certification strategies discussed above show some dissimilar characteristics. All these strategies require some form of certification but the certification process is governed by the role of certifier, and hence they present different pros and cons in the testing process.

In third party component certification a third party tester enhances the confidence in component services. It allows impartial component testing by an independent organization so can be very useful for safety critical systems. On the other hand the cost of component test execution is raised by third party certification process.

For component user certification the fault injection testing by the user allows increased reliability in the component services at a black box level. The component services are executed in the user's system environment thus allowing the black box testing of software component. The only lack in this certification process is that it does not support white box component testing.

Component developer certification can be very useful at deployment time for user, and the added test related information could also be termed as component metadata as they facilitate the user understanding of component functionality. However, as it is the component developer who performs component testing, he/she cannot take into account any context dependence information while generating test cases and test data for integration testing. Testing performed by the developer fulfills the requirements of component unit testing, but cannot be utilized fully by the user for integration testing. Test specification is generated by component developer and statically kept with the software component. For integration testing the component user is assumed to again generate the test cases, and perform integration testing. This certification has less support for impartial component testing. All test specifications are attached and validated by the component developer thus relying completely on the developer for unbiased component testing.

2.7 Customer's Specification-based Testing Approach

To different extents, all of the above approaches rely on some cooperation and good will on the component developer's side: that some specified procedure is followed in producing the component (in particular, that some test cases are executed and documented), or that some required information or property about the component behavior and/or structure is provided. However this cannot be assumed as the general case as often components are delivered supplemented of really little information. At this point the unique means, in the hands of the customer, to increase his/her trust on the component behavior remains the execution of test cases that he/she has defined on the basis of the specifications for the searched component. This philosophy is at the basis of the Customer's specification approach. The use of test cases developed on the basis of the component user's specification and in the target environment is useful in any case, but especially when only black-box test techniques can be used. The application of this kind of approach requires, however, the development of suitable tools and methodologies for test case reuse and derivation. It is also of primary importance to develop new means for the derivation of relevant test cases from the specifications.

2.7.1 Interface Probing

Interface probing [77] identifies a quite intuitive and ad hoc approach to component testing: very simply, the component user derives a set of test cases, executes the component on them, and analyzes the outputs produced. The goal is to understand component properties, functionality and possible limitations. It is proposed more as an approach to component understanding [78], rather than as an approach to integration testing. However, it can also be included in the set of customer's specification based testing approaches, in that it is in fact the customer that designs the exploratory test cases, based on his/her "mental model" [78] and expectations from the component when he/she first uses it. Korel [77] distinguishes among test cases aimed at finding an input on which a desired property is exhibited, test cases aimed at detecting if there exists any input on which a required property is violated, and finally test cases aimed at identifying component pre-conditions. Then he proposes an approach for partially automating interface probing, by means of a search engine which automatically searches for component inputs on which the component property is revealed using a combination of existing automated test generation methods for black-box testing and for white-box testing. More precisely, the component user first uses assertions to describe the properties of interest in terms of the component inputs, and secondly defines the search scope (which consists of the range of input parameters to be explored) as well as the goal for which the search is performed (depending on the above test case classification). Once these specifications are formalized, different automated test generation methods can be used. In particular, Korel proposes random testing, boundary value analysis or execution-oriented methods, as direct-search.

Interface probing is an exploratory test approach, and may require a high number of test cases. With the support of automated search engines, as suggested by Korel's approach, the number of test cases might be reduced, but still a high effort on the side of the component user is required for specifying the needed inputs to the search engines and then to infer component's properties.

2.7.2 Component Deployment Testing

Taking the move from the observation that in practice COTS components may still today be delivered with scarce, if any, useful information for testing purposes, Bertolino and Polini [79] propose an integration testing framework for easing the execution of test cases derived from the customer architectural specifications, making the least restricting assumptions on how the component is developed or packaged.

The user performs an analysis of component requirements before actually deploying the component. Through this analysis the user identifies a *virtual component*, which partially simulates expected component requirements, but without requiring complete component development. The deployment-testing framework then allows the user to test and compare multiple available components, by matching the *real* component outputs and features accessed at run-time with the virtual component. This process involves the following steps [79]:

Step 1: Component user defines component test cases.

The component user defines interface descriptions of a virtual component by coding minimal functionality into a so-called Spy class (virtual component simulated by the user). The virtual component represents the expected requirements of component user, which are expressed syntactically in Spy class containing method signatures. One Spy class may abstract the functionality of multiple objects of real component. Test cases are coded and stored for Spy class by using JUnit tool, which is a framework for conducting and managing test cases. Tool-based approach eases generation of test cases for modified Spy class.

Step 2: Disparities in real and virtual component are settled.

The user selects a set of available components matching user requirements. Each real component is plugged in the framework and its outputs and features at run-time are validated with the virtual component. For each component to be tested, firstly the Spy class is modified according to the component under test. In this way multiple components can be evaluated for performance in the system. In Spy class the expected functionality is expressed syntactically while in performing comparison of this virtual component with the real component, the syntactic differences are ignored. An XML Adapter and Casting classes are used for executing the real component by invocations on virtual component. The method of virtual component are triggered in real component through the casting class, this requires an extra overhead of method invocation transfer mechanism simulated by the component user. The casting classes attempt to resolve the naming, and syntactic conflicts but this mechanism is not tested over a non-trivial example.

Step 3: Test cases are executed on real component.

Real components in this framework must have their run-time checking mechanisms enabled. The mechanisms are required to access the method signatures of real component. An XML Parser obtains the real component method signatures from the XML Adapter and passes them to the Driver class, which is another element in the deployment framework, to simulate the corresponding test cases of virtual component. To execute test cases, the tests are initialized (as generated in the first step), and the methods in real component are provoked by the execuMethod() in the Driver Class, which resolves the syntactic deviations and executes test cases using the XML Parser.

2.7.3 A Framework for OO Component Testing

Buy et al. [80] have proposed a framework for testing OO components. This framework relies on generating message sequences for component under test and for integration of component in the system. Data flow analysis is achieved in a similar mechanism, as defined by Harrold and Rothermel [81]. CCFG (class control flow graph) is generated for the component, which further consists of a control flow graph (CFG) for each method in component, for performing the data flow analysis. Definition use pairs are identified or derived from CCFG. Simultaneously, using symbolic execution the pre, and post conditions, the relation between input and output variables, and the set of variables defined along each path are identified. The def-use pairs and the information via symbolic execution are used for generation of message sequences for the class through automated deduction. The message sequences for single class are generated starting from the constructor and then proceeding through every possible next message at least once. Two components are integrated at a time, i.e., components are added incrementally for integration testing. For integration testing, the message sequences of both components generated separately are combined and their invocations are tested. Component testing framework presents an essential solution for integration testing of OO software components, by defining a mechanism to generate and test message sequences. The symbolic execution is a static analysis technique, which requires program execution using symbols like variable names, i.e., dry run is accomplished by symbolic variables, rather than actual values for input or output data. The program input and output data in symbolic execution is expressed as logical or mathematical symbols rather than the actual values of data. The only limitation in this testing approach for OO components is that it does not take into account source code unavailability, heterogeneity, and significant characteristics of software components.

2.7.4 Customer's Specification-based Testing Approach Review

In customer specification based testing three techniques have been presented. The aim of each technique is to conduct component testing by generating test specification at the user end. We analyze each technique separately.

Firstly, the interface probing technique is largely based on the component user's understanding of the component behavior. The information that is generated from observing test results is adequate for integration testing as it is generated in the context of the component integration environment, but as said a large number of test cases might be required, and some functional guidance in the exploration of component behavior might also be desirable.

Some kind of guidance is provided in the deployment test framework by the customer's architectural specifications. CDT facilitates component user in testing and evaluating multiple components, however the testing of multiple software components in the system can aggravate the cost issues and require extra effort for diligent testing of each software component so that it does not affect the implementation complexity of software system. Besides, for component testing, the developer must enable the run time checking mechanisms in the component (e.g., reflections API in Java), but this may not always be possible, due to the programming language of component or to developer's copyright issues. Hence CDT is technology dependent, and can only be easily used for component developed using Java language. The test cases are generated, executed, and stored for each component. If a first component does not pass the tests, then these test cases can be reused for next component. With an increase in component size, the complexity of test cases is aggravated, and requires management of large number of test cases.

The framework for component testing by Buy et al [80] suffers the problem that it does not cater implementation transparency, an important characteristics of software component. On the other hand the framework incorporates symbolic execution technique for executing the object-oriented component testing process. Another problem in this framework is that it does not handle heterogeneous components rather it is restricted only to OO software components.

2.8 Component Integration Testing: A Comparison of Approaches

In this survey, various component integration-testing techniques have been discussed according to a preliminary coarse classification approach. The classification proposed was useful for viewing similarities and relations between component integration testing techniques as discussed. However, that classification cannot be considered exclusive in application. Based on the discussion, more detailed comparison attributes can now be established, as defined in Table III below. These attributes are then evaluated in Table IV against each of the surveyed approaches. In particular, the second column in Table IV indicates the coarse category of the testing technique, the third column and then each following column refers to the acronyms of the comparison attributes as defined in Table III below. Obviously, the comparison attributes can be usefully applied for evaluating any other component integration testing technique, even the ones not included here.

Table III: Comparison Attributes For Comparing Integration Testing Techniques

Acronym	Comparison attribute	Possible values of attribute
TDeriveBy	Who defines the test cases?	Component Developer (CD), Component User (CU), Third Party Tester (TPT)
TExecuBy	Who executes the test cases?	CD, CU, TPT
Metadata	Additional information required for test derivation?	YES (Which and How), NO
MetaStruct	Additional structure required for test execution?	YES (Which and How), NO
TSpecEasy	Test Specifications are easily accessible	YES, NO
TSpecComp	Test Specifications Increase Complexity	YES, NO
HH	Handles heterogeneous components?	YES, NO
AM	Allows for Easy maintenance?	YES, NO
FCSyntax	Formal Constructs for Test Specification Syntax	YES-P (partial), NO
FCSemantics	Formal Constructs for Test Specification Semantics	YES-P (partial), NO
TDAuto	Supports Test Data Automation	YES-P (partial), NO
TCAuto	Supports Test Case Automation	YES-P (partial), NO
TOAuto	Supports Test Oracles Automation	YES-P (partial), NO
ToolTC/TD	Tool Support for Test-Case or Test-Data Generation	YES (TestCase (TC)/TestData(TD)), NO

Table IV: Component Integration Testing: A Comparison of Approaches

Testing Technique	Category	TDerive By	TExecu By	Meta data	Meta -Struct	TSpecEas y	TSpec Comp	HH	AM	FC Syntax	FC Semantics	TCAuto	TDAuto	TOAuto	Tool TC/TD
BIT Wrappers	BIT Approach	CD	CU	NO	YES	YES	NO	NO	NO	NO	YES	YES	YES	YES	NO
BIT in maintenance mode operation	BIT Approach	CD	CD/CU	NO	YES	YES	YES	NO	NO	YES	YES	NO	NO	NO	NO
Self-testable Software Component	BIT Approach	CD	CD/CU	NO	YES	YES	YES	NO	NO	YES	YES	YES	NO	NO	YES (TC)
STECC Strategy	BIT Approach	CD	CU	NO	YES	YES	YES	NO	NO	NO	NO	YES	YES	YES-P	NO
Component Interaction Graph	Testable Architecture Approach	CU	CU	YES	YES	YES	NO	YES	YES	YES	NO	YES	NO	NO	NO
Introspection Though Component Test Interfaces	Testable Architecture Approach	CU	CU	NO	YES	YES	NO	YES-P	NO	YES	YES	YES-P	NO	NO	NO
Contract Based Built-In Testing	Testable Architecture Approach	CU	CU	NO	YES	NO	YES	NO	NO	YES	NO	YES	NO	NO	YES (TC,TD)
Metadata For Component Testing	Metadata Approach	CU	CU	YES	YES	YES	YES	NO	NO	NO	NO	NO	NO	NO	NO
UML Test Model, Checking OCL Constraints	Metadata Approach	CU	CU	YES	NO	YES	NO	NO	NO	YES	NO	YES	NO	YES	YES
Third-Party component certification	Certification Strategy Approach	TPT	TPT	YES	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO
Component user certification	Certification Strategy Approach	CU	CU	NO	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO
Component developer certification	Certification Strategy Approach	CD	CD/CU	YES	NO	YES	YES	NO	NO	YES	YES	YES	NO	NO	YES (TC)
Interface Probing	Customer's Specification Based Testing Approach	CU	CU	NO	NO	YES	NO	YES	NO	NO	NO	YES-P	YES-P	NO	NO
Component Deployment Testing	Customer's Specification Based Testing Approach	CU	CU	NO	NO	YES	YES	NO	YES	YES	NO	YES	YES	NO	YES (TC)
A Framework for Software Component Testing	Customer's Specification Based Testing Approach	CU	CU	NO	NO	NO	YES	NO	NO	YES	YES	NO	NO	NO	NO

7. Conclusions

While generally acknowledged as a quite attractive paradigm for increasing reuse and reducing time-to-market, component-based development also gives rise to a new critical problem in the area of software production, generally referred to as the *Component Trust Problem* [82]. This problem points at the component user's exigency of means to gain confidence on what a component produced by someone else does and how it behaves. Obviously this issue is especially hard for components built by third parties and for COTS delivered without the source code. However, also in the case of components reused internally to an organization, the difficulties of communication between teams and the lack of a clear documentation can produce to some extent similar effects.

This survey article discussed software component testing issues and provided an overview of proposed techniques for component integration testing on the component user’s side. The importance of this stage in CB development has already been highlighted also by authoritative sources, and can never be overstressed. In fact, even though a component has already undergone extensive testing by its developer, since complete testing is clearly impossible and the developer cannot know in advance all the possible application domains or what components will interact with the produced component, some kind of testing against the component user's specifications remains always necessary [50]. In this sense, it is also illusory to hope that reuse of components drastically diminishes the need for testing [42, 50].

The most critical problem in component integration testing is the lack of information for component analysis. For application of appropriate testing techniques, packaging of suitable metadata along with the component implementation would be advisable. Testing techniques have thus been proposed which exploit metadata attached with the component; still the existing metadata approaches do not completely fulfill the requirements for component test information. In fact, the inclusion of metadata must be accomplished such that it does not affect the component implementation transparency. The future research in this field requires the development of mechanisms to ensure that proper information is available to the component users at the integration testing time. This also requires the definition of automated processes to append metadata with component before its final shipment and the processes, which display the metadata to user without exposing implementation details.

As a final remark, it may be useful to remind with [82] that “no single technique can produce completely trusted components”. This paper focused on component integration testing approaches, but in practice an appropriate combination of approaches should be applied, both to component construction, such as the mentioned Design by Contract [6] approach or formal specification techniques, and to validation, such as formal proofs, formalized review, and differentiated testing phases. By providing an up-to-date and comprehensive overview of proposed integration testing techniques for CB systems, classified according to a proposed set of relevant attributes, this paper hopefully provides a baseline for continuing investigation in the field and a useful piece of input as well towards addressing the burning problem of component trust.

References

1. Garlan, D., Allen, J. and Ockerbloom, J. Architectural Mismatch or Why it's hard to build system out of existing parts, *IEEE Proc. 17th Int. Conf. on Software Engineering*, April 1995; 179-185.

2. Lions, J.L. *ARIANE 5, Flight 501 Failure, Report by the Inquiry Board*, http://java.sun.com/people/jag/Ariane5.html. 1996.

3. Meyer B., and Mingins, C. (eds). Component-Based Development: From Buzz to Spark, Guest Editor’s Introduction, *IEEE Computer*. **32**(7); July 1999.

4. Crnkovic, I., Schmidt, H., Stafford, J., and Wallnau, K. (eds). *The Journal of Systems and Software - Special issue on CBSE,* **65**(3). 2003.

5. Szyperski, C., Gruntz, D. and Murer, S. *Component Software—Beyond Object Oriented Programming*, The Component Software Series, 2nd Edition,Addison-Wesley. 2002.

6. Crnkovic, I., Larsson, M. (EDS). *Building Reliable Component-Based Software System*, Artech House. 2002.

7. Beydeda, S. and Gruhn, V. (eds). *Testing Commercial-off-the-Shelf Components and Systems*, December, Springer Verlag. 2004.

8. McIllroy, D. Mass Produced Software Components, *Software Engineering: Report on a Conference by the NATO Science Committee*, pp. 138-155, edited by P. Naur and B. Randall. 1969.

9. Crnkovic, I. Component-based Software Engineering - New Challenges in Software Development, Software Focus, (December), John Wiley & Sons Editors. 2001.

10. Brown, A. An Introduction to Model Driven Architecture Part I: MDA and today's systems, *The Rational Edge* (Journal), January-February, available at: http://www-106.ibm.com/developersworks/rational/library/3100.htm. 2004.

11. Selic, B. The Pragmatics of Model-Driven Development, *IEEE Software*, Volume-20, number 5, (September), pp 19-25. 2003.

12. Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison Wesley Professional. 1999.

13. Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, Addison-Wesley Professional. 1998.

14. Garlan, D., and Shaw, M. *Software Architecture: Perspective on an Emerging Discipline*, Prentice-Hall. 1996.

15. Garlan, D. Software Architecture: a Roadmap, *Foundation of Software Engineering*, ACM Press, pp 91-101. 2000.

16. Formal UML Specifications. The Unified Modeling Language (UML), available at: http://www.uml.org. 2003.

17. AA, VV. UML 2.0 Superstructure Specification, August, *Technical Report by Object Management Group.* 2003.

18. Warmer, J., and Kleepe, A. *The Object Costraint Language: Getting your models ready for MDA*, 2nd Edition, Addison-Wesley. 2003.

19. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., and Robbins, J.E. Modeling Software Architectures in the Unified Modeling Language, *ACM Transcations on Software Engineering and Methodology,* (January), **11**(1); pp 2-57; 2002.

20. Cheesman, J., and Daniels, J. *UML Components - a Simple Process for Specifying Component-Based Software*, Addison-Wesley. 2001.

21. MDA List. An MDA Tools list, available at: http://www.modelbased.net/mda_tools.html.1999.

22. Shannon, B. Java 2 Platform Enterprise Edition specification, 1.4 - Final Release, Sun Microsystems, Inc. (November). 2003.

23. Dot Net Resources. . Net resources, available at: http://www.microsoft.com/net/2000.

24. Emmerich, W. *Engineering Distributed Objects*, John-Wiley \& Sons. 2000.

25. Bernstein, P. A. Middleware: A Model for Distributed System Services, *Communications of the ACM,* **39**(2); pp 86-98; February 1996.

26. Component Object Model Technologies. COM: Component Object Model Technologies, available at: http://www.microsoft.com/com/default.mspx. 1999.

27. Sun Microsystems. http://java.sun.com/products/javabeans/. 1994.

28. Corba Component Model Specifications. CORBA Component Model specifications, Object Management Group. http://www.omg.org/technology/documents/formal/components.htm. 1997.

29. Monson-Haefel, R. *Enterprise JavaBeans - Developing Enterprise Java Components*, 3rd Edition, O'Reilly. 2001.

30. Brown, A.W. *Large Scale, Component-Based Development*, NJ: Prentice Hall, Object Oriented Series, Englewood Cliffs. 2000.

31. Carney, D., and Long, F. What do you mean by COTS? – finally, a useful answer, *IEEE Software*, **17**(2). 2000.

32. Brereton, P., and Budgen, D. Component-Based Systems: A Classification of Issues, *IEEE Computer*. Vol. **33**(11). November, pp. 54-62. 2000.

33. Meyer, B. Applying Design by Contract, *IEEE Computer*, **25**(10); pp 40-51; October 1992.

34. Bertolino, A., and Polini, A. Re-thinking the Development Process of Component-Based Software, *In Proceedings ECBS 2002 Workshop On CBSE*, April, Lund, Sweden. 2002.

35. Cai, X., Lyu, M.R., and Wong, K.F. A Generic Environment for COTS Testing and Quality Prediction, in [7].

36. Dogru, A.H., Tanik, M.M. A Process Model for Component-Oriented Software Engineering, *IEEE Software*, Vol. **20**(2). pp. 34-41. 2003.

37. Bertolino, A. Knowledge Area Description of Software Testing, Chapter 5, in *Guide to the Software Engineering Body of Knowledge SWEBOK (v. 0.7)*, April, Software Engineering Coordinated Committee (Joint IEEE Computer Society-ACM Committee), (available from *http://www.swebok.org*). 2000.

38. Harrold, M.J., Liang,D., Sinha, S. *An Approach to Analyzing and Testing Component Based Software*, In Proceedings of the First International Workshop on Testing Distributed Software Systems (at ICSE'99); Ohio State University, LosAngeles, United States. May 1999.

39. Harrold, M.J. Testing: A Roadmap, *22nd International Conference on Software Engineering* (ICSE00), ACM Press; pp. 63-72. June 2000.

40. IEEE Standard Glossary of Software Engineering Terminology. *ANSI/IEEE Standard 610-12-1990, IEEE Press, New York*. 1990.

41. Councill, W.T. Third-party Testing and the Quality of Software Components, *IEEE Computer*, pp. 55-57. August 1999.

42. Binder, R.V. *Testing Object-Oriented Systems Models, Patterns, and Tools*, Object Technology Series, Addison Wesley. 2000.

43. Stuart, C. Reid. *BS 7925–2: The Software Component Testing Standard*, In Proceedings of the First Asia-Pacific Conference on Quality Software, pp. 139-148. 2000.

44. Gao, J. Challenges and Problems in Testing Software Components, In *Proceedings of ICSE 2000's 3rd International Workshop on Component-based Software Engineering: Reflects and Practice*, (June), Limerick, Ireland. 2000.

45. Gao, J.Z., & Wu.,Y. *Testing and Quality Assurance for Component Based Software*, Artech House. 2003.

46. Freedman, R.S. Testability of Software Components, *IEEE Transactions on Software Engineering*, Vol.**17**(6); pp.533-564. June 1991.

47. Memon, A. M. Developer's Role in Making Testable Components, in [7].

48. Wu, Y., Pan, D., Chen, M. Techniques for Testing Component-based Software, *In Proceedings of the Seventh International Conference on Engineering of Complex Computer Systems*. 2001.

49. Bhor, A. Software Component Testing Strategies, *Technical Report, Dept. of Information and Computer Science, Irvine,Universtiy of California ,United States*. 2001.

50. Weyuker, E.J. Testing CBS: A Cautionary Tale, *IEEE Software*, AT&T Labs. 1998.

51. Rosenblum, D.S. 1997. Adequate Testing of Component Based Software, *Technical Report UCI-ICS-97-34,Department of Information and Computer Science, University of California*, (August), Irvine.

52. Weyuker, E. J. *Axiomatizing software test data adequacy*, IEEE Transactions on Software Engineering, vol. 12(12), (December), pp. 1128-1138. 1986.

53. Edwards, S.H. A Framework for practical, automated black-box testing of component-based software, *Software Testing, Verification and Reliability*, (June), Vol. **11**(2) .pp. 97-111. 2001.

54. Wang, Y., King, G., Wckburg, H. *A Method for Built-in Tests in component-based Software Maintenance*, IEEE International Conference on Software Maintenance and Reengineering, pp186-189. 1999.

55. Martins, E., Toyota, C.M., Yanagawa, R.L. *Constructing Self-Testable Software Components*, In Proceedings of the International Conference on Dependable Systems and Networks. 2001.

56. Beydeda, S., and Gruhn, V. *Merging components and testing tools: The Self-Testing COTS Components (STECC) Strategy*, In Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture", Germany, IEEE. 2003.

57. Siegel, S. *"Object-Oriented Software Testing: a Hierarchical Approach"*. John Wiley & Sons. 1996.

58. Toyota, C.M. Improving Class Testability using the Self- Testing concept, *Master Dissertation*, Institute of Computing, (June), State University of Campinas. 2000.

59. Gao, J., Kamal. G., Gupta. S., and Shim, S. On Building Testable Software Components, In *Proceedings of International Conference on Component-based Software System*, pp. 108-121. 2002.

60. Atkinson, C. et al. Built-in Contract Testing in Model-Driven Component-Based Development, *In Proceedings of Workshop on Component-Based Development Processes*, April, Austin, Texas (USA). 2002.

61. Atkinson, C., and Gross H.G. *Component-based Product Line Engineering with UML*, Addison-Wesley. 2001.

62. Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Soffa, M.L.,and Doo, H. Using Component Metadata to support the regression testing of component-based software, In *Proceedings of the International Conference on Software Maintenance (ICSM2001)*, pp 716-725, November, Florence, Italy. 2001.

63. Orso, A., Harrold, M. J., and Rosenblum, D. S. Component metadata for software engineering tasks, *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, Davis, C.A., United States, pp. 129-144. 1999.

64. Liang, D., and Harrold, M. J. Reuse-driven inter-procedural slicing in the presence of pointers and recursion. In *Proceedings IEEE International Conference on Software Maintenance*, pp. 421-430. 1999.

65. Redolfi, G., Spagnoli, L., Hemesath, P., Bastos, R.M., Ribeiro, MB., Cristal, M., Espindola A. 2005. A Reference Model for Reusable Components Description, *In Proceedings of the 38th Hawaii International Conference on System Sciences.*

66. Stafford, J.A., Wolf, A.L. Annotating Components to Support Component-Based Static Analyses of Software Systems. *In Proceedings of the Grace Hopper Celebration of Women in Computing*. 2001.

67. Whaley, J., Martin, M.C., Lam, M.S. Automatic Extraction of Object-Oriented Component Interfaces. *In Proceedings of the International Symposium on Software Testing and Analysis* (ISSTA 2002). pp 218-228, Roma, Italy. July 2002.

68. Wu, Y., Chen, M. and Offut, J. UML-based Integration Testing for Component-based Software. *Proc. 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, (February),Ottawa, Canada. 2003.

69. Brucker, A. D. and Wolff, B. Checking OCL Constraints in Distributed Systems Using J2EE/EJB. *Technical Report 157,* July, *University of Freiburg*. 2001.

70. Cechich, A. and Polo, A. Testing Commercial-off-the-Shelf Components and Systems, in [7].

71. Bachman, F., Wallnau, K. "Volume II: Technical Concepts of Component-Based Software Engineering", May, *Technical Report, Carnegie Mellon Software Engineering Institute*. 2000.

72. Morris, J., Lee, G., Parker, K., Bundell, G.A., and Lam, C.P. Software Component Certification, *IEEE Computer,* (September), Vol. **34**(9). pp. 30-36. 2001.

73. Ma, Y.S., Oh, S.U., Bae, D.H., and Kwon, Y.R. *Framework for Third Party Testing of Component Software*, In Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC, 01). 2001.

74. Voas, J.M.. Certifying Off-the Shelf-Components, *IEEE Computer, Los Alamitos, CA, United States; pp. 53-59.* June 1998

75. Voas, J. M. Developing a Usage-Based Software Certification Process, IEEE Computer, **33**(8); pp. 32-37. August 2000.

76. World Wide Web Consortium. Extensible Markup Language, http://www.w3.org/TR/2000/RECxml-20001006. Centre for Intelligent Information Processing Systems. 2001.

77. Korel, B., Black-Box Understanding of COTS Components. *Proceedings of the 7th International Workshop on Program Comprehension IWPC*, IEEE Computer Society 1999, pp. 92-99.

78. Andrews, A. , Ghosh, S., Man Choi, E., A Model for Understanding Software Components, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)* Montreal, October 3-6, 2002, pp. 359-369.

79. Polini, A., and Bertolino, A. A User-Oriented Framework for Component Deployment Testing, in [7].

80. Buy, U., Ghezzi, C., Orso, A., Pezze, M., and Valsasna, M. A framework for testing object-oriented Components, *Proceedings of the First International ICSE Workshop on Testing Distributed Component-Based Systems.* 1999.

81. Harrold, M. J. and Rothermel, G. Performing data flow testing on classes. In *Proc. 2nd ACM-SIGSOFT Symposium on the foundations of software engineering,* (December), pages 154-163. 1994.

82. Meyer, B., Mingins, C., Schmidt, H. Trusted Components for the Software Industry, available at http://www.trusted-components.org/documents/tc_original_paper.html 1998