Testing software components for integration: a survey of issues and techniques

Muhammad Jaffar-ur Rehman¹, Fakhra Jabeen¹, Antonia Bertolino²,∗,† and Andrea Polini²

¹Centre of Software Dependability, Mohammad Ali Jinnah University, Islamabad, Pakistan
²Istituto di Scienza e Tecnologie dell’Informazione ‘A. Faedo’, CNR, Area della Ricerca di Pisa, Via Moruzzi 1, I-56124 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, thereby rendering component analysis and testing a key activity. This paper discusses various issues that can arise in component testing by the component user at the stage of its integration within the target system. The crucial problem is the lack of information for analysis and testing of externally developed components. Several testing techniques for component integration have recently been proposed. These techniques are surveyed here and classified according to a proposed set of relevant attributes. The paper thus provides a comprehensive overview which can be useful as introductory reading for newcomers in this research field, as well as to stimulate further investigation. Copyright © 2006 John Wiley & Sons, Ltd.

Received 27 October 2005; Revised 21 June 2006; Accepted 7 September 2006

KEY WORDS: software components; component testing; integration testing; metadata

1. INTRODUCTION

In the last few years software developers have been facing unrivalled challenges. On the one hand, information processing systems have become increasingly complex, networked, pervasive and also

∗Correspondence to: Antonia Bertolino, Istituto di Scienza e Tecnologie dell’Informazione ‘A. Faedo’, CNR, Area della Ricerca di Pisa, Via Moruzzi 1, I-56124 Pisa, Italy.
†E-mail: antonia.bertolino@isti.cnr.it

Contract/grant sponsor: Marie Curie TAROT network; contract/grant number: MRTN-CT-2004-505121
Contract/grant sponsor: PLASTIC; contract/grant number: FP6 STREP 26955

Copyright © 2006 John Wiley & Sons, Ltd.
critical, given that they are being increasingly used in activities that carry an element of risk. On the other hand, the high competitiveness in software production causes an almost unbearable reduction of time-to-market. As a consequence, developed software must retain maintainability, reusability, testability, and all the other ‘-ilities’ related to software production, whilst at the same time assuring high dependability, or else the consequences can be catastrophic.

A very promising answer to these challenges relies on the potential to obtain complex systems by the composition of prefabricated pieces of software called components, following the example provided by ‘more traditional’ engineering disciplines. Indeed, component-based (CB) 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 higher reliability. However, some laboratory experiments [1], and even catastrophic events [2,3] have warned that composing software components is not an easy task at all; on the contrary, much work is necessary to make this approach succeed. As a consequence, a new, very active, research branch has established itself inside the software engineering area, generally referred to as component-based development (CBD), as testified by the escalation of conferences (e.g. the Component-Based Software Engineering Symposium, the Component Deployment Working Conference and the International Conference on COTS-Based Software Systems to cite just a few examples), special issues of journals [4,5] and books [6–8] devoted to the topic.

In fact, the very idea of producing software systems out of components is more than 30 years old [9], but it is only recently that greater efforts have been made towards the concrete affirmation of this methodology. Nowadays, CBD promises the benefit of producing quality software, whilst minimizing time and resources, facilitating software reuse and promoting rapid application development. However, such accomplishments come with drawbacks. For example, concealing proprietary information is one of the advantages of CBD. This privacy, however, affects component user’s understanding for subsequent component testing.

Crnkovic [10] provides a long list of research challenges related to CBD, several of which closely affect the capability of testing the components, such as the need to develop languages for modelling software systems at a high level of abstraction, by describing the architecture of the system in terms of coarse-grained components and of their required features, or the need to develop suitable technologies to facilitate component integration and their communication. In general, it is necessary to better understand the steps that can lead to the development of a system starting from available components. From a theoretical viewpoint, the toughest issue is composition predictability, which involves the capability of inferring interesting properties of an assembled system, starting from the known properties of the components used in the composition.

With regard to testing, it is particularly important to implement techniques and tools that make the testing of externally acquired components easier. Indeed, if the rapid expansion of reusable software components requires that their reliability as individual parts be ensured, it requires that their successful integration must also be effectively tested. The well-known failure of the Ariane 5 rocket launch vehicle, which veered off-target and blew up less than a minute after take-off, is a prominent and unfortunate demonstration of the importance of integration testing, as this failure was attributed to insufficient testing of a reused software component in its system [2,3].

In this paper, software component testing is discussed by reviewing recent literature, with an emphasis on those techniques which address the testing of a component when put in the context of
other interacting components. Following this introduction, the paper is organized as follows. Section 2 covers the basic concepts and characteristics of components, their enabling technologies (Section 2.1) and their development process (Section 2.2), followed by a summary of component testing phases (Section 2.3) and component testability metrics (Section 2.4). CBD brings change into the development process, which further impacts on the testing phases of the component lifecycle. Test coverage notions for components are also outlined (Section 2.5). Section 3 provides a broad overview of various research contributions which attempt to provide solutions for component testing when integrated within a target assembly system. In Sections 3.2–3.6 five categories of approach are distinguished, within each of which several techniques are presented. In Section 4, the reviewed techniques are summarized and compared in terms of a number of identified characterizing attributes. Conclusions are briefly drawn in Section 5.

2. BASIC CONCEPTS AND CHARACTERISTICS OF COMPONENTS

In this section, software components are characterized. One of the most widely accepted and reported software component definitions is provided by Szyperski [6]:

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 several different, peculiar aspects of components. In particular, there is a technical element, with aspects such as independence, contractual interfaces and composition, and also a market-related element, with features such as third parties and deployment. A component can be plugged into a system as a unit of composition, with its services becoming accessible via a defined set of interfaces. The importance of interfaces can also be determined from Brown’s definition of a software component as ‘... an independently deliverable piece of functionality providing access to its services through interfaces’ [11]. The dependencies of each component on other components or systems must be clearly defined for accurate usage. Finally, a component is deployed as an autonomous unit, possibly by organizations other than the developing organization.

Component-oriented versus object-oriented paradigm. The component-oriented paradigm is different from the object-oriented (OO) paradigm. The component, by its very nature, is a compositional element which, as previously stated, must contain information enabling some form of compositional reasoning. This means that the system assembler should, without instantiating the component or looking at its internal code (which is, in general, not accessible), be able to understand the service offered by the component and which other services/components it requires. In an OO paradigm, the basic concept of an object is a low-level concept, strictly related to the execution of the system. In such a setting, the concept which yields the closest relation with that of a component is certainly its class. Many authors [12] actually consider a class as a simplified example of a component. Nevertheless, classes show the service offered only at the code level, and, embedded in the code, the relations with required classes. Rather, a component is a conceptual entity and does not imply a specific form of implementation, although it may often be developed using OO concepts and languages.
In the latter case, instantiated components come to reality as a web of interrelated object instances. On the other hand, an object 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.

**Source code unavailability.** The availability of source code, or otherwise, makes a great difference in system development, and in particular for testing. Software components are normally developed by a provider organization and used by multiple users, who do not necessarily belong to the same organization. The implementation of components is generally not exposed to users, but only textual abstractions are attached as interface specifications. The signature of each interface is mentioned with an explanation of the component functionality, without any implementation details. Thus, the component source code is hidden, reducing development complexity for application developers at the component user end. The user gains the benefit of component services without having to be concerned with the implementation details, while the provider holds the component ownership rights even after component deployment. Authors generally use the terms ‘black-’, ‘grey-’ and ‘white-box components’ with reference to different levels of disclosure regarding the component’s internal workings. In particular, a black-box component does not disclose anything about its internal implementation, whereas, at the opposite end of the spectrum, a white-box component reveals this to the user in its entirety. Between these two extremes, there may exist several different levels of grey-box components, depending on how many details are made public. A more articulated classification is provided by Carney and Long [13], who refer more precisely to the degree to which the user can or must modify the code. The choice of what is the most opportune level of transparency is not straightforward, as both ends of the spectrum have their pros and cons. The availability of code could provide component users with major guarantees. However, they should not use such information, so as to avoid the assembled CB system becoming dependent on the component’s implementation, with potential negative effects on system evolution.

**Commercial off-the-shelf (COTS) components.** This generic term refers to software components, but more specifically to a category of products, which are made commercially available in a pre-packaged form [6]. As highlighted by Carney and Long [13], an unequivocal definition of COTS components does not exist. COTS components can refer to diverse types and levels of software components, from a piece of software that provides a specified function to a CASE tool. This confusion is also testified to by the existence of several attempts at classification, which are usefully surveyed by Morisio and Torchiano [14].

**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 of use to someone else, otherwise the component has no market and there is no reason to develop it. In this context, aspects of service granularity also 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 a service may reduce the market for the component [6], and therefore the economic viability of developing it. Component granularity mainly affects the development environment [15] and, in particular, the production time and effort of 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. Nevertheless, carrying on such activities at run-time clearly depends on the existence of a suitable component platform.

Component metadata. Metadata consists of augmenting the component with additional information in order to increase the component user’s analysis capability. Carney and Long [13] 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 in terms of the level of modification allowed by the component source, and the artefacts attached to components in each category. Brereton and Budgen [15] gathered and analysed disparate issues of CB applications in the form of a framework. Issues relating to software components as a product, the development process and business issues relating to components are discussed from the perspectives of software engineers, as developers or assemblers of CB software applications. It is envisaged that components will probably shift the focus of software engineering from the ‘specify, design and implement’ paradigm towards one characterized by ‘select, evaluate and integrate’, whereby, 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 exhaustive information on the service that is implemented by the component, for instance by using some formal mechanism. An interesting and quite successful way to associate semantic information with an interface is via the use of contracts, as advocated in the well-known ‘design-by-contract’ paradigm [16,17]. A contract describes a service using first-order logic and specifying conditions that should hold before the invocation of the services, as well as conditions that will be true after the execution of the service. A contract can also specify invariant conditions that remain fixed during the whole execution of the service. Contracts are a really useful mechanism in CBD, although their use can raise some problems, in particular when callbacks—i.e. the called service itself reinvokes the caller—are considered, as described in depth by Szyperski [6].

Introspection. Introspection is an important feature in component-oriented development. Through introspection services, access to the internal information of a component is provided at run-time, so that this information can be used for cooperation or simply for gaining a better understanding of the component-provided services. Introspection operates alongside reflection services. For example, the java.lang.reflect package in Java allows access to component features without accessing the source code directly. Similarly, for the same purpose, run-time type identification (RTTI) is supported in C++. Some of the testing techniques that are surveyed later exploit introspection for accessing components undergoing testing, thereby obviating the problem of missing information during component integration testing. Introspection mechanisms provide a very versatile way for dynamically linking components or for implementing adaptation policies. For instance, the component might modify its behaviour depending on the environment (i.e. the interacting components) in which it is deployed.
2.1. Technologies enabling component-based development

The development of modern software systems would not have been possible without the great achievements in the field of software technology of the last few years, in particular:

- middleware, addressing interoperability and distribution issues;
- component models, focusing on managing the reuse issues, and on defining rules for packaging and accessing services.

2.1.1. Middleware

In parallel with the trend for the modularization of complex systems into components, the last few years have also witnessed an enormous rise in the demand for distributed software systems. The development of distributed software is far more complex than the development of centralized software. For this reason it is useful to hide, as much as possible, from system developers the issues strictly related to distribution. This is the rationale behind the concept of middleware. Middleware can be considered as a type of connectivity software that allows applications to interact with each other across a network. In particular, middleware provides services that are generic across applications and industries, run on multiple platforms, are distributed, and support standard interfaces and protocols [18]. 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.

Implementation of CB systems above the middleware layer renders the new dimensions of complexity introduced by distributed systems transparent to the CB system engineer. In particular, among desirable transparency features [19] for CBD, access transparency requires that the interface to a service does not depend on 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.

2.1.2. Component models

Although a generally accepted definition does not exist, in general terms, a component model consists of a set of rules to be followed in component development, and also subsumes a specific platform on which the same component is deployed. The rules to be followed might concern the way in which the component should be packaged (e.g. in a Jar file), the interfaces that must be implemented, which are generally useful for management purpose of the platform, and a set of documents to be filled out by the developer providing additional information on the component itself. At the same time, the platform associated with a component model will define the way in which specific general services, for example naming services, can be accessed and provide the mechanisms for binding together components using the provided interfaces and the additional data.

Two main kinds of component models are considered. The first, called desktop components, provides mechanisms to integrate components deployed on the same system. This is the case of COM [20] and JavaBeans [21]. The second model, referred to as distributed components, provides mechanisms for integrating components that might be dispersed on more than one physical system. In the latter case, the component platform will also enclose connection mechanisms typical of a middleware layer.
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 in another world, even though some research efforts have started to investigate a possible technical solution to this problem [22].

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 the 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 [20], CCM/CORBA from OMG [23] and EJB/J2EE from Sun Microsystems [24].

2.2. Component-based development process

Every software development process relies on the well-known ‘divide and conquer’ strategy for solving large problems, i.e. a problem is identified and broken down into smaller sub-problems so that each sub-problem can be disentangled and solutions assembled to form the system. In a conventional process, the lifecycle phases 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 standardized documentation packaged with the software to facilitate software testing.

In CBD, the activities performed during the lifecycle and the relationship between phases change. The same ‘divide and conquer’ strategy can be applied to CBD, but, in general, available components are explored to search for a solution through what is generally referred to as the provisioning phase rather than building new components. Hence, the implementation phase mainly deals with the development of what is generally referred to as the glue code. This code is the instrument required to facilitate correct interaction among the different components.

After one or more candidate components have been identified, it is necessary to evaluate their behaviour when integrated with the other previously selected components. Obviously, for this purpose, testing plays a key role. In fact, on the basis of the specifications of the component, testers can develop useful (functional and architectural) tests to be executed so as 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. A distinctive feature of CBD is the co-existence throughout the development process of several stakeholders. A CB process must, in fact, envisage and manage the spreading in time and space of different tasks among several uncoordinated subjects [25]. Hence, it is essential to define special engineering processes to assemble the building blocks in the complete system. Cai et al. [26] presented an overview of CB lifecycle processes embedding quality assurance models.

A general process model for CBD has been presented by Dogru and Tanik [27]. Software system specifications are analysed and modularized into sub-parts. For each sub-part, a component is sought from among a set of previously 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

Copyright © 2006 John Wiley & Sons, Ltd.  
Softw. Test. Verif. Reliab. 2007; 17:95–133  
DOI: 10.1002/stvr
Table I. Perspectives in component testing.

<table>
<thead>
<tr>
<th>Component provider</th>
<th>Component user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develops component</td>
<td>Uses component</td>
</tr>
<tr>
<td>Source code available</td>
<td>Source code unavailable</td>
</tr>
<tr>
<td>White-box and black-box testing</td>
<td>Black box of component unit</td>
</tr>
<tr>
<td>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 component user, it can be termed white-box testing)</td>
<td></td>
</tr>
<tr>
<td>Context-independent view of component</td>
<td>Context-dependent view of component</td>
</tr>
<tr>
<td>All configurations or aspects of component behaviour must be tested</td>
<td>Subset or all configurations and only those aspects of component behaviour that are application related are tested</td>
</tr>
<tr>
<td>Performs unit test for ensuring its correct functionality</td>
<td>Performs integration test by invoking component services</td>
</tr>
<tr>
<td>Low bug-fixing cost</td>
<td>High bug-fixing cost</td>
</tr>
</tbody>
</table>

The 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’ [27]. The connector set represents the interrelationships identified during decomposition and carries integration process information, which is useful during the integration testing of components.

With special focus on COTS components, Morisio et al. [28] have analysed several CB projects at NASA, integrating more than 30 COTS products. From their study, new activities (COTS evaluation, COTS familiarization and vendor interaction) and new roles for conducting those activities emerged. They also observed that the decision to use pre-built components, as opposed to development from scratch, was often made implicitly, with no rigorous or formal analysis of the costs and risks involved.

2.3. Software component testing phase

As with the other development phases, the testing stage also needs rethinking in order to address the peculiar characteristics of CBD [25]. The component provider and the component user roles can be compared, thus 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 regarding the environment in which the component will be employed. The second needs to (re)test the component as an interacting part of the larger CB system under development. Harrold et al. [29] addressed issues and challenges involved in analysis and testing in CBD, both from the perspective of the component provider and the component user. Table I compares widely referenced perspectives of Harrold [30], based on the distinct characteristics of software components presented in the next section. Clearly, Table I oversimplifies the respective features for illustrative purposes. In reality, distinctions that are not as neat may well apply. For instance, the component user could also conduct unit testing to some extent by means of self-testing approaches.
Traditionally, the development of complex systems involves three main testing phases (unit testing, integration testing and system testing), while at each level regression testing may need to be applied. In CBD, these three traditional testing phases have to be reconsidered and extended (see Figure 1).

### 2.3.1. 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 the early detection of possible failures. The availability of source code permits white-box testing of all component configurations, regardless of a specific usage context. The developer must also perform black-box component testing to ensure that correct specifications are attached to the reusable component. However, such testing cannot address the functional correspondence of the component behaviour to the specifications of the system in which it will be subsequently assembled. In fact, it is impossible for the developer to consider all the environments in which the component could be successively inserted.

Gao et al. [31] recently proposed a test model which can be used for component unit testing. The test model is named the *component function access graph* (CFAG), since every node in CFAG represents a function. For example, in OO terms it represents a function of the class interface. It is an effective technique, since it also maintains the dynamic aspect of the component in another model: dynamic CFAG or D-CFAG. Referring to the CFAG and D-CFAG models together, some test coverage criteria
are defined, which provide dynamic information flow coverage. A component test coverage analysis tool, termed COMPTest, has also been built by Gao et al. [31] using the models and criteria, and has been shown to be applicable for component testing.

2.3.2. Integration/deployment testing

The IEEE defines integration testing as ‘testing in which software components are combined and tested to evaluate the interaction between them’ [32]. Indeed, unit testing cannot confirm the reliable behaviour of components in a new system; hence, another testing campaign—by the component user—is essential in obtaining an acceptable level of reliability. Councill [33] used the term second-party testing for testing on the part of the component user during its implementation in the real environment.

Component integration testing is an indispensable testing phase. Software components can be incorporated into a system as units, or multiple components may work together to provide system functionality. A component, whether integrated individually or with other components, requires integration testing, i.e. the testing of component interactions as envisaged in the actual usage environment before actual deployment.

The testing of component deployment can be further divided into two sub-phases. In the first sub-phase, the component can be tested as integrated in an environment made up of stubs that roughly implement the components as envisaged by the specifications. In this manner it is possible to check at an early stage whether the component correctly interacts with the ‘ideal’ environment. In the second sub-phase the real integration between several chosen components is verified. To achieve this, the interactions among the actual implementations of the architectural components during the execution of some test cases are monitored, and undesired interactions can possibly be 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 highlight the non-conformance between the expected component and the component being tested (and hence the need to look for other components).

A particular case of integration testing is when a real component comes equipped with the developer’s test suite, and the assembler re-executes those tests in their environment. These tests guarantee that the ‘intentions’ of the developer are respected in the final environment and their execution generally leads to a more comprehensive evaluation. They can possibly include test cases that, although not relevant to the user’s specific purposes, might still be useful in evaluating the behaviour of the component under the user’s unenvisaged conditions.

2.3.3. System testing

System testing 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, so as to test the functionality of the whole system. The emphasis of system testing is not restricted to the components undergoing testing. Rather, these components are tested within the context of the system, in order to assess the behaviour and performance of the whole system as a black box.
2.3.4. Component regression testing

As components are added to successive test configurations, accumulated test suites must be re-run to reveal regression bugs, thus incrementally building the regression test suite [34]. A regression test suite is used to test previously tested code again, in a new context. The previously 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.

2.4. Software component testability

Component testability measures can help component users to select and evaluate components, and to gauge the effort made by the developer on the component to increase component testing capability. The IEEE Standard Glossary [32] defines software testability as:

\[
\text{The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; the degree to which a requirement is stated in terms that permit the establishment of test criteria and performance of tests to determine whether those criteria have been met.}
\]

In this definition two different testability concerns can be identified: first, the way a software system is developed; and second, the requirements on the software system that serve as the basis for test suite definition. Component testability can be further subdivided under the following five metrics [35,36].

- **Component observability.** According to Freedman [37] 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.
- **Component controllability.** According to Freedman [37] component controllability is the ease of producing a specific output from a specific input, i.e. the expected outputs are controllably produced from particular inputs.
- **Component understandability.** The interface descriptions and further information attached to components to explain their functionality also enhance their understandability, facilitating component reuse.
- **Component traceability.** Traceability can be classified as black-box traceability (referring to the comparison of component behaviour with its execution) and white-box traceability (established by checking the internal state of software components at each interface invocation). Gao et al. [36] explored the concept of traceable CBD in distributed environments, whereby the traceability metric involves various traces, which include operational, event, state, performance and error traces.
- **Component test support capability.** The test support capability involves mechanisms to generate test suites, to execute testing and to manage the test process. Although it is the least explored metric, it is important in assessing the testability of software components.

Gao and Shih [38] deal with the issues in component testability analysis, verification and measurement. A component testability analysis model is presented in the form of a fishbone diagram, which is developed by the refinement of the above testability metrics [35,36]. The model enhances
every testability metric, identifying a set of factors affecting the metric, thus helping to analyse and verify component testability. For example, component understandability, one of the considered metrics, is affected by factors such as document availability, document readability, requirements testability, etc. A quantifiable approach for measuring component testability is presented in a pentagram model [38]. Based on the verification results of the component analysis model, each testability metric is assumed to vary between 0, the minimum value, and 1, the maximum. The five metrics are assigned/mapped/represented as the five pentagram points. Component testability is equal to the value of the area of this pentagram. However, since no engineering guidelines exist for requirements verification or design review [38], this approach lacks a systematic solution for identifying the correct discrete values of component testability metrics.

2.5. Coverage notions for component integration testing

Some relevant coverage notions for component testing follow [39].

2.5.1. 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 invocations, parameters and return types.

At a deeper 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.

2.5.2. Context-dependence test coverage

Events can have sequential dependences on each other; thus the order in which they are triggered can give rise to distinct program behaviours. For example, an interface might be dependent on another for its execution, which is in turn dependent on another interface, and so on. The chain of dependences may continue. Such dependences are termed context-dependences. Context-dependences provide coverage for interoperability faults, as direct and indirect interactions of software components are tested. The coverage of context-dependences can only be achieved by testing all possible operational sequences in the component.

2.5.3. Content-dependence test coverage

A still higher coverage level tends to test all content-dependence relationships. A content-dependent relationship exists between two interfaces $I_1$ and $I_2$ of a component if an operation of $I_2$ modifies the value of a variable that is used in an operation of $I_1$. In that case, interface $I_1$ is said to be content-dependent on $I_2$. This coverage corresponds to the data-flow strategy for the structural testing of traditional programs.
3. TESTING APPROACHES FOR COMPONENT INTEGRATION

A survey of several proposed approaches related to CB testing from the perspective of the component user is now provided. Obviously, the current list presented cannot claim to be exhaustive, but reflects the greatest extent of the authors’ comprehension and knowledge of the literature. Approaches for testing a component from the perspective of the component developer are purposely omitted from the scope of this paper, since the information available on the component’s internal structure (such as the source code) allows the developer to use different (e.g. white-box) methodologies [40].

Unfortunately, the same characteristics that, on the one hand, make components an attractive means for software production, on the other may negatively affect the testing process. In fact, typical component features such as source code unavailability, component heterogeneity and a continuously changing nature, are generally recognized as impediments to integration testing. Certainly, the major problem that affects testing is the lack of information. Whilst the developer performs unit testing with no system context information (as also highlighted by Weyuker [41]) or knowledge of the component deployment environment, the component user chooses and deploys a component in their application, by analysing only the component published services, but with no implementation details. Thus, mechanisms for component integration testing, in general, demand that additional information be included with the component, and/or additional structure for the reliable use of component applications, which is necessary for the effective realization of the ‘reuse’ benefit of components.

3.1. A classification of techniques

When testing a component for integration, the test inputs must be determined in the actual usage environment, i.e. integration testing requires knowledge of the component usage environment. Hence, only the component user, or a third-party tester (TPT) in possession of this type of information, can conduct integration testing. The TPT may be provided with test information from both the user (component usage requirements) and the developer (component metadata, see Section 3.4).

For testing purposes, a component can be classified depending on the information that is provided by the component itself. For exposition purposes, the techniques are classified into five categories which are discussed, respectively, in Sections 3.2–3.6:

1. built-in testing (BIT);
2. testable architecture;
3. metadata-based;
4. certification strategy;
5. user’s specification-based testing.

A more detailed categorization and comparison of the integration testing techniques is subsequently provided in Section 4.

3.2. Built-in testing approaches

BIT approaches are a generic approach to testing, previously adopted in OO programming. In general, BIT is ‘the test software that resides in an application’ [34], i.e. literally the component is augmented with executable test cases that are built into the component together with its normal functions.
BIT requires component developers to embed tests in software component implementations 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.

### 3.2.1. Built-in testing wrappers

Edwards [42] has proposed a framework to provide BIT wrappers for component testing, using the specification language RESOLVE as an example (although other languages could also be adapted). The RESOLVE specification is assumed to include pre- and post-conditions for each interface, for which 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 the component’s internal errors, while the external layer handles the component’s integration with the system or other components. The wrapper has the added benefit of testing the software component, although diligent effort is required for managing the BIT code. The wrapper must be attached to the component in such a way that neither is the component behaviour modified nor the component’s internal structure affected, because this may put at risk the original objective of bounding wrappers around the component. The key objective is to test component behaviour and to reduce behaviour invalidity, not to introduce further complexity or make any changes in the structure of the component. A wrapper is like a filter that allows the passage of only certain inputs to and outputs from a component. The automation of the framework relies on the RESOLVE specifications provided with the component [42]. The interface specifications can be used to generate test suites, test drivers, 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 behavioural descriptions, i.e. specifications attached to 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, even if 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 by Edwards. Rather, it is restricted only for those components having RESOLVE specifications attached to the component. The specifications are necessary for framework automation, and, in the case of a missing specification, the programmer is required to generate the specifications manually by executing the component and determining component behaviour. The generality of this approach is limited, as it is dependent on the provider to supply a BIT wrapper definition and to provide component specifications. Besides, maintenance issues need special care for components with BIT wrappers.

### 3.2.2. Built-in testing in maintenance mode operation

The BIT approach has been used by Wang et al. [43] for enhancing CB software maintainability. They build tests in component source code as extra member functions; the 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, which 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 component users with a powerful means of evaluation, without requiring them to use any specific framework or tool for testing purposes.

3.2.3. Self-testable software components

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

Martins et al. [44] used the Transaction Flow Model (TFM) as designed by Siegel [45] to create self-testable components. The TFM is used for the unit testing of a class; message sequences for a single class are generated as a transaction starting from object instantiation to its destruction. The TFM is generated for a class, and, for each transaction, the Concat [46] test tool is used to generate test cases. The test cases, once produced, are reused by applying different values of test data to the same test cases. The component user accesses these test cases through methods built into the software component. The additional code for testing the component is thus built-in and executed so as to find any inconsistency in component implementation.

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

3.2.4. The self-testing COTS component strategy

Beydeda and Gruhn [47] proposed the Self-TEsting COTS Component (STECC) strategy, which is similar to BIT, given that the software component is itself enabled so as 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 the component can also perform test evaluation if the expected results are defined as test oracles in component implementation; otherwise the results have to be manually verified. This technique relies on the component provider defining an objective component testing process.

3.2.5. Built-in testing for component behaviour prediction

Barbier and Belloir [48] proposed BIT for component behaviour prediction and for the monitoring of real-time and safety-critical systems. Their approach is applicable only to components that have the reflection mechanism enabled. Since the component interface details are extracted dynamically, the accessed information is always consistent with the current component version. This access of component behaviour during execution is termed Quality of Service (QoS) testing [48]. QoS testing identifies the residual errors in a component, as well as largely identifying the errors that occur due to errors in the component’s integration within the system.
3.2.6. Component+ built-in testing

Momotko and Zalewska [49] proposed a framework based on Component and BIT (C+ BIT) technology for testing component interactions with the environment at run-time, and also provided a preliminary evaluation of the framework. The C+ BIT technology [49] proposes two types of testing techniques for software components; i.e. contract testing and QoS testing. Contract testing aims at providing a BIT mechanism to ensure that the component is providing services to the environment and the environment/system is also behaving according to the contract. QoS testing not only ensures the contract testing of component and system, but it also verifies continuously that the component provides correct functionality at run-time, and identifies the points at which possible defects might occur.

The C+ BIT architecture [49] has four types of components: BIT components, testers, handlers and system constructors. This architecture is generally applicable to any CB system. The BIT component is itself the component that can obtain from or provide information to other parts of the architecture. The tester component gathers information from the BIT component as well as other components in the system so as to execute contract-based and QoS testing. Handlers are responsible for keeping the test information up-to-date; since it is a generic architecture, the precise implementations of handlers are not defined. The system constructor is in charge of interconnecting all other components in the architecture. The typical and mandatory interfaces of the architecture components are also specified.

As a result of experimentation, Momotko and Zalewska infer that the C+ BIT technology does not increase the size of component source code. The technique is also ideal for the QoS testing of components with a complex algorithm, and furthermore, these tests can be reused. C+ BIT technology is appropriate for in-house components, but for the typical case of COTS components, the technique has some additional requirements, which include either the component users building in their own tester component or the component developer enhancing the component with open source test cases and test data that can be further improved by the component user. The first approach requires some testing knowledge at the component user end, while the second approach relies on the developer to provide open source testing code for the component. The additional benefits that C+ BIT provides over other BIT techniques is that this framework handles component heterogeneity, and also reduces the overheads of building the test cases within the component by providing a separate component to handle the testing issues of the software component. However, the technology still requires further experimentation.

3.2.7. Built-in testing approach review

In general, the main advantage of the BIT approach is that it enhances component testability and allows for easy maintenance. However, it does suffer from some drawbacks. The first, which is technical in nature, is that the memory required at run-time to instantiate the objects from a class can become huge and is mainly dominated by the need to allocate space for the testing methods. These testing methods, obviously, are completely useless in normal mode. The second and more conceptual problem concerns the meaningfulness for the component user of the developer’s defined test cases. This approach depends on the component provider to generate test cases for the software component. Even without considering malicious developers, only those aspects of a component that are enabled by the component developer can be tested.
BIT does improve component testability, but it is limited up to a certain level, as it only allows the developer to provide static component test coverage criteria. As advocated by different authors [50], it is instead important that the user develops their own test suites so as to ascertain whether a candidate component is ‘compliant’ with the requirements of the component under consideration.

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 such as OO components (Java or C++), but rather would be uniformly applicable to all component types, regardless of their implementations. The definition of standard procedures for stub and driver generation can further enhance BIT generality, and also facilitate the execution of component integration testing.

3.3. Testable architecture approaches

This approach can be seen as a special case of BIT, and in fact shares the same aims. However, unlike BIT, testable architecture avoids the problem of a huge amount of memory being 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 execute the test cases easily. The test information is appended by the developer in the form of specifications, instead of enclosing them in the component itself.

3.3.1. Component interaction graph

An effort by Wu et al. [39] defines a test model called the component interaction graph (CIG) for integration testing. As a first step, test elements are identified to build a 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 the component source code, if it is accessible to the user. If the component implementation is not visible, then mechanisms are required to extract component test elements indirectly from the component design information. Wu et al. [39] 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 to the component. Otherwise, a set of events can be generated exhaustively for testing purposes. When in possession of the interfaces and events, an algorithm is designed for extracting context dependences. For content dependences in the CIG, the component developer is required to append the design information, such as a class diagram. A CIG model defines the test coverage criteria needed to reveal interaction faults by exhaustively testing all test elements, as mentioned in Section 2.5.1. The results of a case study also reveal that the test elements are useful in deriving test cases for integration testing of OO components. However, the technique suffers a drawback, as it would allow for access to component source code through reverse engineering from design information. A CIG model requires the developer to append design information, such as a sequence diagram, for easy test derivation. This information may allow the component user to perform the reverse engineering task and obtain access to source code, thus affecting the implementation transparency, which is an important characteristic of a software component.
3.3.2. **Introspection through component test interfaces**

Gao et al. [51] require that each component implements a specific interface for testing purposes. This interface has the explicit goal of augmenting component testability. In this manner, the developer can subsequently provide the component user with test cases coded in terms of clients that use the testing interface. By envisaging the presence in the test-oriented interface of methods that use introspection mechanisms, which are generally provided by component standard models, the same power of a BIT 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 the component developer.

3.3.3. **Contract-based built-in testing architecture**

Another interesting approach, likewise relying on the definition of a particular framework for component testing, has been proposed by Atkinson and Gross [52]. The framework is not intended for the execution of generic test cases, but rather focuses on providing the component user 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 [53], a modelling 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 solely for the interfaces specified by the user. This architecture thus facilitates the component user in establishing confidence in component services. It also increases component reuse on the developer’s side, as the test specifications generated for one user may be reused when the same component is delivered to another user.

The implementation of this method is achieved through the use of modelling tools. Any component 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 a component. Hence, the technique does not present a generalized approach that is equally applicable for heterogeneous component environments.

3.3.4. **Integration testing framework**

Jabeen and Rehman [54] proposed an OO component testing framework, which uses three discrete descriptors which are prepared, respectively, by the component provider, component user and a TPT, to provide an effective mechanism for communicating test information. A generalized XML (eXtensible Markup Language) format is used for the descriptors to establish uniform information flow. Since XML is a standard format, it can be applicable to all component environments. The component developer is required to provide additional information such as pre- and post-conditions for each interface, so that component testing can be performed with no source code access. Thus, the developer prepares a component descriptor in XML format and attaches it to the component for the TPT. The component user is also required to specify the anticipated component requirements in another descriptor, i.e. a component requirement descriptor (CRD). Using the information in the component descriptor and CRD, finally, the TPT generates the test information in another descriptor.
The role of the TPT in the framework not only supports impartial testing, but also ensures the component user obtains the correct and required functionality from the software component. In this way, each participant in the framework has responsibility for generating respective descriptors to facilitate component integration testing.

### 3.3.5. Testable architecture approach review

The testable architecture approach is slightly different from BIT and also solves some of the problems inherent in the BIT approach. First, the memory consumption problem is resolved, as the test specifications are built separately from the component source code. Second, a few testable architecture approaches allow the component 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, thereby simplifying the component integration process.

This approach enhances component testability, yet poses considerable challenges. Contract-based testable architectures, as discussed in Section 3.3.3, are tool dependent, and so may not be applicable to components not running the modelling tool. This limits the architecture only for those components that support the modelling tool. Similarly, the approach cannot be generally applicable to all component types. Extra effort is demanded from the developer, largely during maintenance time, and most of the techniques in this approach are no longer valid for heterogeneous components.

The testable architecture must support the test case derivation process according to user requirements, but at the same time must preserve component implementation transparency, which is an important characteristic of software components. Component reverse engineering, through component design information, allows for component source code generation. In the same way, the Unified Modelling Language (UML) dynamic diagrams in the CIG technique (Section 3.3.1) may allow the component user to generate the source code from the component’s design. The addition of such component design might affect implementation transparency.

### 3.4. Metadata-based approaches

Among the several problems hindering the CB testing process, the main issue is the lack of adequate information regarding the acquired component. A great deal of research work has attempted to solve this problem by attaching additional information to the component, whilst not making available the source code. All forms of additional information appended to the software component, either by the developer, the user or a TPT, so as to facilitate software testing can be regarded as forms of metadata. In particular, to improve the integration testing process on the part of the component user, metadata is defined as the information added to the component in order to increase the component user’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 [55], and so on. The CIG, discussed in Section 3.3.1, for example, can itself be seen as a form of metadata. Metadata-based component integration testing techniques, as discussed in the following subsections, can be compared by an analysis of the typology of metadata appended to 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.

3.4.1. *White-box components analysis*

Orso *et al.* [12] highlight the need for added component information for testing and maintenance purposes. In their view, component implementation information is extracted and used as metadata, given that the developer is required to enable run-time checking mechanisms. Three types of summary information are provided in this approach.

- **Program slicing** involves associating slices of the component source with a set of variables. Program slicing is carried out so as to develop an understanding of a software component. Program statements are divided into chunks of code (i.e. sub-domains) according to variable use, and summary information is bound to each chunk. A slicing algorithm [56] is used for backward traversal of the program in order 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 that statement. Errors that may occur in the component due to the external environment are attached to those statements, and can be reported by exception generation. At a detailed level, summary information can be provided, with each exception containing information for sets of variables and a state defined for the values of variables. This control-flow analysis information is defined for the abstract component model, but needs to be redefined according to the requirements of a particular model.

- **Data-flow analysis** at the lowest level provides the definition and use only of input variables, while at the highest level this information can contain the definition and use of each variable in the component, as defined in the program slicing phase. For achieving higher levels 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 a program are tested, thereby accomplishing a 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 initial work for recovering any missing information, and can be extended to supply comprehensive component testing information.

A format similar to the Multi-purpose Internet Mail Extension (MIME) 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 a complete definition of component behaviour.

Likewise, the proposal of Stafford and Wolf [57], who envisage the provision of pathways expressing the potential an input may have to affect a particular output, can be considered a particular instance of the metadata approach. Whaley *et al.* [58] propose supplying models that express acceptable method call sequences. In this manner, the users can evaluate their use of the component and check whether legal calls are indeed permitted.
3.4.2. UML test model

Given its widespread adoption, UML-based metadata obviously become an attractive means for the integration testing of CB software in industry. Wu et al. [59] propose an approach for metadata in the form of UML models, which target the addition of specific test elements for software components on the part of the providers. They define a set of adequacy testing criteria in which 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 context-dependence relationships (see Section 2.5.2). These relationships are modelled conveniently and directly in UML interaction diagrams. The execution of each possible sequence through these diagrams provides the desired validation, which can be enhanced still further by covering the possible sequences in the associated statecharts. Diversely, the content-dependence relationships are not modelled directly in any UML diagram. However, the same information can be extracted indirectly from collaboration and statechart diagrams. Collaboration diagrams, which involve entity classes, can show the content-dependence relationship through update and retrieve messages. An update message only modifies a value (in an entity class). This is modelled 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 modelled by a pair of in- and out-flowing messages. If the execution of an interface $I_1$ involves an update message that has a corresponding retrieve message in interface $I_2$, then $I_2$ is content-dependent on $I_1$. As with collaboration diagrams, the use of statecharts 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, given 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 the test elements of a software component. Automatic test case generation from UML models allows software component users to generate test cases from the UML artefacts attached by the component developers. On the other hand, the proposed test model assumes that one interface corresponds to one operation of a software component, but more generally an interface may abstract the occurrence of multiple component operations, whereas the invocation of one interface in a 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. [59] advocate providing UML diagrams with the component only for 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.

An integration testing technique using FSMs is defined by Gallagher et al. [60]. This work develops an inter-class testing technique that uses a data-flow-based approach and relies on UML diagrams. A software component is basically modelled in the form of a state machine and data flow is used to identify the definition-use paths that subsequently form the input for testing software components. They aim at solving the issues in OO component integration testing. However, as with other model-based approaches, implementation transparency is affected.

Redolfi et al. [61] defined a reference model in the form of a class diagram. They took into account characteristics and properties of components existing in the literature for defining this reference model.
The model does not directly expose implementation details. However, an implementation specification in the class diagram permits the accessing of component behaviour, as with the component source code, thus affecting the implementation transparency feature of the component. The model is mainly used in defining a component repository, so that a component can easily be accessed, based on the requirements of component functionality and behaviour. Reuse, an important benefit of software components, is achieved through the documentation of component properties. By enhancing component understandability, this reference model can also be classified as a form of metadata facilitating component testing.

3.4.3. Checking Object Constraint Language constraints

Another approach to metadata-based testing is that proposed by Brucker and Wolff [62]. This technique, which has been inspired by the design-by-contract principle [16] and based upon it, attempts to generate components that have the capability of self-testing. This is achieved by building some testing code into the component that will execute and test the component. The self-testing code is in the form of pre- and post-conditions for 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 the Object Constraint Language (OCL). Once the components are built, these constraints have to be added. This is achieved automatically through the instrumentation tools provided. Similarly, the execution of this code calls for the execution and checking of the constraints that are still specified in the OCL format, even in the code. Again, tool support for OCL constraint checking exists. In addition to providing BIT support in components, Brucker and Wolff also define some design patterns for distributed components and support them with a prototype tool.

3.4.4. Automated black-box testing framework

Belli and Budnik [63] propose a framework for the automation of user-oriented component testing. Component developers, in general, do not provide any domain knowledge, or component implementation knowledge. This framework thus requires the augmentation of a component with a model to automate black-box testing of the component using conventional test tools. Since this technique relies on the augmentation of the component with specifications in the form of a component model, it can be classified as a metadata approach. Another model called the ‘behavioural model’ is generated using the graphical user interface (GUI) objects in XML format. The component model is also transformed into an intermediate XML format. Thus the two models are used for generating test scripts. Both the behavioural model that is based on the GUI interaction of a component, as well as the component model that is a UML statechart, are then used for test generation. This framework uses existing methodologies for the modelling of interactive systems and for test generation from the UML models [64]. The framework mainly considers black-box component unit testing. Thus, it might reduce the implementation transparency of a component, since UML models clearly provide information about the component internals. The component models, however, must be generated every time the component is modified. When the component changes, the previously generated models become useless, since the models are not updated as a consequence of modification in the component. Further research is required to identify automatic updating procedures for these models, so that consistency and correctness can be maintained.
3.4.5. Metadata approach review

In common with the testable architecture approach, the metadata approach tends to enhance component testability. In this approach, the component user is provided with some (limited) component details so that the user is allowed to generate test cases from the added metadata. Strictly speaking, the addition of information per se is not a test technique. However, component metadata clearly assists in software engineering tasks such as testing and maintenance by providing analysis and testing information. From the added information, for instance, test coverage criteria can be defined according to user requirements. Nevertheless, there exists a need to define standardized methods to query and generate the metadata dynamically, given that, for large software components, the design information or metadata becomes too complex to manage. The techniques in the 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 on the effectiveness of these testing approaches. For instance, Cechich and Polo [65] propose using ‘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, as is the case with the testable architecture approach. The design information appended as metadata, particularly UML dynamic structure diagrams (e.g. interaction diagrams) in fact might reduce implementation transparency, for example by allowing component reverse engineering. However, component metadata should not affect the implementation transparency feature of a software component. Another advantage of the approach is that the techniques can be further explored so that test information, as with test data and test cases, can be automatically generated from the metadata. This can further ease the integration testing process at the user end.

3.5. Certification strategy approaches

The widespread use of software components can only be obtained by building a high level of confidence in the software component. However, the user of a component is generally suspicious about the information and proof of quality provided by the component developer. Indeed, the component developer cannot be trusted 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, in order to provide a more effective method of increasing a user’s trust in a particular component, different forms of component certification have been proposed [66]. These forms mainly rely on the idea that each component undergoes a standard certification for the development process, as well as for its features and architecture. In general, the component certification process is classified into the following types, according to who takes on the role of certifier:

- third-party component certification;
- component user certification;
- component developer certification.
3.5.1. Third-party component certification

Third-party certification is an effective way to build component trust, because it seeks to preserve objectivity in component testing [33,67,68]. 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.

An initial proposal by Voas [66] called for 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 tests carried out and the environments exploited. This can be more beneficial for large software systems, and particularly for safety-critical systems.

Councill [33] states that third-party testing can provide impartial component testing to component users, and it may be performed either voluntarily, or as required contractually or enforced by legal permission. The results or findings of third-party testing are readily available to any organization that might request them.

Ma et al. [68] have proposed a framework for the 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 the metadata are not identified, although it is assumed they generally include information in line with the general philosophy of component metadata, i.e. analysis and test support information. The framework defines a three-step process.

- The TPT provides guidelines and supporting tools to the component developer.
- The component producer generates a test package using these guidelines. The test package consists of information for deploying and testing the component, as well as to audit the test suite of the component in the form of metadata.
- The TPT checks the conformance of the test package with its guidelines, executes the test package, and generates a test report.

An evaluation of such a third-party testing framework by Ma et al. [68] demonstrated that the TPT encountered some problems while executing the test suite provided by the developer. The objective of the third-party testing framework was for the functional testing of a software component to be performed by a third party possessing some prior knowledge of the component. However, the generated test package proved insufficient for the testers to thoroughly understand and test component functionality. The problems that manifested themselves during evaluation reinforce the need for the producer to attach comprehensive metadata, but the framework lacks an explicit definition and a formalized notation for metadata representation. Some discrepancies also occurred between part of the testing information metadata and the software component. In addition, the component developer could not produce correct test oracles, as the developing team lacked the required testing skills, in spite of guidelines being given to the developers for the generation of the test package. These problems arose due to reliance on the component producer for the generation of the test package, whereas, normally, development teams lack standard testing potential. Notwithstanding, a 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 CB applications.
Table II. Objectives of testing in the certification process.

<table>
<thead>
<tr>
<th>Testing for certification process</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-box component testing</td>
<td>To check component compliance with the specification, and to check component quality</td>
</tr>
<tr>
<td>Operational system testing</td>
<td>To determine the system’s reliability with an operational component</td>
</tr>
<tr>
<td>System-level fault injection</td>
<td>To determine the system’s reliability while generating failures in the component</td>
</tr>
</tbody>
</table>

3.5.2. Component user certification

The inherent difficulties in establishing these third-party agencies suggested that, as an 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 a particular software component, a ‘user-based software certification process’ [69] could be established.

The component users would be able to check that a selected component performs according to the specified functionality for a system by referring to component user certification. It is also an element of integration testing. Voas [66,69] defines a component user certification process as incorporating black-box component testing, operational system testing and system-level fault injection. Table II provides the objective of each of the testing techniques which are used in the certification process.

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 the component user then operational system testing of the component is performed. In order to assess reliability, faults are randomly generated and, instead of testing the component with correct inputs, the system is tested with erroneous outputs from the component. The erroneous outputs generated and passed to the system allow the user to determine the undesirable inputs to the system. Those outputs of a component which can adversely affect the system are revealed. 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, the component buyer defines oracles for testing the quality of a 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 reason, the component developer may append additional specifications to the component, which can assist in finding such errors by the application developer.

3.5.3. Component developer certification

A different approach to component certification has been proposed by Morris et al. [67]. This approach originates in the remark that using the services of a certification agency could prove particularly expensive for a small software company. To overcome this problem, certification on the part of the software’s developer has been proposed, i.e. the component developer certifies that a component’s functionality complies with the test specifications provided with the component, by testing the
component’s behaviour and attaching testing information (metadata) to the component. It is therefore envisaged that the developer should generate test cases and testing information, perform component testing and append the testing cases and test results with the component, as proof of having tested the software component. This approach relies on the release, together with the component, of test case sets specified in a formal notation using XML [70]. This format should guarantee a better understanding, on the component user’s part, of the test cases that have been carried out by the developer on the component.

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

The test specifications [67] consist of a set of test cases generated for each interface. A test set consists of multiple TestGroups, which further consist of operations. Each operation is a collection of method calls. Thus, to perform operation testing, each method call in the operation is tested. To be precise, the test case for each operation requires running through a set of method calls given in the operation. For OO components, there is a specification of the invariant element, which must hold true from object construction, and through all operations in an object’s lifecycle until object clean-up.

Another means by which the developer can increase trust in the mission-critical components is by adding contracts [16,17] to a software component. The component developer may add the contracts at four different levels [17], which range from non-negotiable to negotiable contracts. The levels include basic contracts, behavioural contracts, synchronization contracts and QoS contracts. This technique attempts to verify the reliability of a component prior to its incorporation in the system. The contracts may allow the component user to verify and validate component interfaces as defined by the developer in the contract specification. In this way, defining contracts in software components provides a workable solution for enhancing component reliability. However, contracts at a deeper level may affect the implementation transparency of software components.

3.5.4. 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 certifier, and hence they present different pros and cons in the testing process.

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

For component user certification, 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 for black-box testing of a software component, without, however, making use of white-box component testing.

Component developer certification can be very useful at deployment time for users, and the added test-related information could also be termed as component metadata, given that it facilitates user understanding of component functionality. However, as it is the component developer who performs component testing, no account can be taken of any context-dependence information while
generating test cases and test data for integration testing. Testing performed by the developer fulfils the requirements of component unit testing, but cannot be utilized fully by the component user for integration testing. A test specification is generated by the component developer and statically kept with the software component. For integration testing, it is assumed that the component user will repeat the test cases, and perform integration testing. This certification has reduced support for impartial component testing. All test specifications are attached and validated by the component developer, with the result that there is total reliance on the developer for unbiased component testing.

3.6. User’s specification-based testing approach

To a differing extent, all of the above approaches rely on some element of cooperation and goodwill on the part of the component developer, that some specified procedure will be followed during the production of the component (in particular, that some test cases will be executed and documented), or that some required information or property about the component behaviour and/or structure will be provided. However, this cannot be assumed to be the general case, as often components are delivered with very little supplementary information. At this point, the sole means in the hands of the user to increase trust in the component’s behaviour remains the carrying out of test cases that have been defined on the basis of the specifications for the component in question. This philosophy is at the basis of the user’s specification approach. The use of test cases developed on the basis of the component user’s specification in the target environment is useful in any case, but especially so when only black-box test techniques can be used. However, the application of this kind of approach requires the development of suitable tools and methodologies for test case reuse and derivation. In addition, the development of a new means for the derivation of relevant test cases from the specifications is of primary importance.

3.6.1. Interface probing

Interface probing [71] identifies a quite intuitive and ad hoc approach to component testing. Put very simply, the component user derives a set of test cases, executes the component in accordance with them and analyses the outputs produced. The goal is to understand a component’s properties, functionality and possible limitations. It is proposed more as an approach to component understanding [72], rather than as an approach to integration testing. However, it can also be included in the set of user’s specification-based testing approaches, in that it is, in effect, the user that designs the exploratory test cases, based on their ‘mental model’ [72] and expectations of the component when it is first used. Korel [71] distinguishes among test cases aimed at finding an input on which a desired property is exhibited, test cases aimed at detecting whether there exists any input on which a required property is violated and test cases aimed at identifying component pre-conditions. Korel then proposes an approach for partially automating interface probing, by means of an 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 in question in terms of the component inputs, and second defines the scope of the search (which consists of the range of input parameters to be explored) as well as the goal for which the search is being 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, such 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. Nevertheless, a great effort on the part of the component user is still required for specifying the needed inputs to the search engines and subsequently to infer the component’s properties.

3.6.2. Component deployment testing

Having observed that, in practice, COTS components may still today be delivered with scarce, if any, useful information for testing purposes, Polini and Bertolino [73] proposed an integration testing framework for easing the execution of test cases derived from user architectural specifications, making the least restrictive assumptions on how the component is developed or packaged.

The component user performs an analysis of component requirements before actually deploying the component. Through this analysis, the user identifies a virtual component, which partially simulates envisaged 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 [73].

**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 the component user, which are expressed syntactically in a Spy class containing method signatures. One Spy class may abstract the functionality of multiple objects of a real component. Test cases are coded and stored for the Spy class by using the well known JUnit tool [74], which is an open source framework for conducting and managing test cases. The tool-based approach eases generation of test cases for a modified Spy class.

**Step 2: Disparities in real and virtual components are settled.** The component user selects a set of available components which match user requirements. Each real component is plugged into the framework and its outputs and features at run-time are validated with the virtual component. For each component to be tested, the Spy class is modified according to the component undergoing testing. In this way multiple components can be evaluated for performance in the system. An XML adapter, and casting classes are used for executing the real component by invocations on the virtual component. The methods of the virtual component are triggered in the real component through the casting class. This requires the extra overhead of a method invocation transfer mechanism simulated by the component user. The casting classes attempt to resolve the naming and syntactic conflicts but this mechanism has not been tested on a non-trivial example.

**Step 3: Test cases are executed on the 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 a real component. An XML parser obtains the real component method signatures from the XML adapter and passes them on to the driver class, which is another element in the deployment framework, in order to simulate the corresponding test cases of the virtual component. To execute test cases, the tests are initialized (as generated in the first step), and the methods in the real component are provoked by the `execMethod()` in the driver class, which resolves the syntactic deviations and executes test cases using the XML parser.
3.6.3. Behaviour capture and test framework

Mariani and Pezzè [75] emphasize the value of gathering the integration test results of a software component in different deployment environments. Once a software component is developed, it can be reused many times. In the same manner, the component behaviour in one system can be captured, and subsequently monitored or analysed while reusing the component. Instead of performing exhaustive testing of a component, the saving and monitoring of test results in one environment can be useful for evaluating component behaviour in another environment. The Behaviour Capture and Test (BCT) [75] framework uses an object-flattening technique to monitor component behaviour, while invariants are used for extracting component design information. Two types of invariants include the ‘interaction invariant’, and the ‘input/output (I/O) invariants’. When the component is deployed for the first time in the deployment environment, the data collection phase starts [75]. In this phase, the events generated from the component are sent to an Invariant Distiller, which prepares the invariants (I/O or interaction) and saves the operation of every service in the form of a Finite State Automata (FSA). The next phase is the monitoring phase, which is achieved through invariant checking, in which the monitors are inserted to check the invariants at run-time. If there is no invariant similar to the existing invariants, the invariant is inserted as an FSA, i.e. the data collection phase may overlap with the monitoring phase. Otherwise, the invariant is checked and any violation is signaled at run-time. This technique is similar to the retrospection mechanism [76], where the component is made so as to keep a retrospector that holds the static and dynamic test history of the software component. Through retrospection, the information that has been built into the software component is also useful in executing component integration testing. Since the retrospection mechanism [76] enables the component to maintain a test history, it facilitates the provider in maintaining a record of continuous component testing of different systems. It also helps the component user to understand the software component from its test history. However, one limitation of retrospection seems to be that when the component is modified, the test history of a component may become ineffective for the next version of the component.

In the BCT framework, the inconsistency of the component with the test information is managed by means of an automated analysis process. BCT is defined and explained with preliminary experimental results that show the applicability of the approach for testing software component integrations in the system, through reusing component behaviour captured in the component. The drawback of the technique is that it requires the component to have run-time checking mechanisms enabled for the dynamic monitoring of component behaviour [75].

3.6.4. A framework for object-oriented component testing

Buy et al. [77] have proposed a framework for testing OO components. This framework relies on generating message sequences for the component undergoing testing and for the integration of the component in the system. Data-flow analysis is achieved in a similar mechanism, as defined by Harrold and Rothermel [78]. A class control-flow graph (CCFG) is generated for the component, which further consists of a control-flow graph (CFG) for each method in the component, so as to carry out data-flow analysis. Definition-use pairs are identified or derived from the 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 definition-use pairs and the information via symbolic execution are used for the generation of message sequences for the class...
through automated deduction. The message sequences for a 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. The component testing framework presents an essential solution for the 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 such as variable names, i.e. a ‘dry run’ is accomplished by symbolic variables, rather than through 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 limitations in this testing approach for OO components is that it does not take into account source code unavailability, heterogeneity or the significant characteristics of software components.

3.6.5. User’s specification-based testing approach review

In the user’s specification-based testing approach, different techniques have been presented, which variously attempt to conduct component testing by exploiting the test specification at the user end.

The interface probing technique is largely based on the component user’s understanding of the component’s behaviour. 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 has been previously stated, a large number of test cases might be required, and some functional guidance in the exploration of component behaviour might also be desirable.

Various frameworks have been proposed to support the component user in testing and evaluating multiple components. However, the testing of multiple software components in the system can aggravate cost issues and require extra effort for the diligent testing of each software component, so as not to affect the implementation complexity of the software system. Besides, for component testing, the developer must enable the run-time checking mechanisms in the component (e.g. the reflection Application Programming Interface (API) in Java), but this may not always be possible, due to the programming language of the component, or due to the developer’s copyright issues. Hence, such frameworks become technology dependent. For instance, component deployment testing can only be easily used for components developed using the Java language, while the framework of Buy et al. [77] does not handle heterogeneous components and is restricted only to OO software components.

4. SUMMING UP

In this section, a synthesis is made of the topics and approaches discussed. Some tables are used to provide a quick overview of the main characteristics of each presented approach. Finally, an attempt is made to present a critical discussion of guidelines for choosing among the many approaches.

4.1. A comparison of approaches

In this survey, various component integration testing techniques have been discussed in accordance with a preliminary coarse classification approach. The classification proposed was useful for viewing
Table III. Comparison attributes for comparing integration testing techniques.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Comparison attribute</th>
<th>Possible values of attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDeriveBy</td>
<td>Who defines the test cases?</td>
<td>Component developer (CD), Component user (CU), Third-party tester (TPT)</td>
</tr>
<tr>
<td>TExecBy</td>
<td>Who executes the test cases?</td>
<td>CD, CU, TPT</td>
</tr>
<tr>
<td>Metadata</td>
<td>Additional information required for test derivation?</td>
<td>YES (which and how), NO</td>
</tr>
<tr>
<td>MetaStruct</td>
<td>Additional structure required for test execution?</td>
<td>YES (which and how), NO</td>
</tr>
<tr>
<td>TSpecEasy</td>
<td>Test specifications are easily accessible</td>
<td>YES, NO</td>
</tr>
<tr>
<td>TSpecComp</td>
<td>Test specifications increase complexity</td>
<td>YES, NO</td>
</tr>
<tr>
<td>HH</td>
<td>Handles heterogeneous components</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>AM</td>
<td>Allows for easy maintenance</td>
<td>YES, NO</td>
</tr>
<tr>
<td>FCSyntax</td>
<td>Formal constructs for test specification syntax</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>FCSemantics</td>
<td>Formal constructs for test specification semantics</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>TCAuto</td>
<td>Supports test case automation</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>TDAuto</td>
<td>Supports test data automation</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>TOAuto</td>
<td>Supports test oracles automation</td>
<td>YES, YES-P (partial), NO</td>
</tr>
<tr>
<td>ToolTC/TD</td>
<td>Tool support for test case or test data generation</td>
<td>YES (test case (TC)/test data (TD)), NO</td>
</tr>
</tbody>
</table>

Similarities and relations between the component integration testing techniques as discussed. However, that classification cannot be considered exclusive in practice.

Based on the discussion, more detailed comparison attributes can now be established, as defined in Table III. The first column assigns an acronym to each identified attribute; the second column defines the attributes of interest in terms of a self-explanatory sentence; finally, the third column provides the possible values for qualifying the corresponding attribute.

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 subsequent column, refers to the acronyms of the comparison attributes as defined in Table III. Obviously, the comparison attributes can be usefully applied for evaluating any other component integration testing technique, even those which have not been included here.

4.2. Other component-testing surveys

The topic of CB testing has previously been the subject of other related surveys. For the sake of completeness, some recent efforts are summarized here. Beydeda and Gruhn [79] identify the problems which occur due to a lack of information in CB systems, and present an overview of approaches that exist to solve these problems. According to Beydeda and Gruhn [79], the approaches that solve the missing information problem either address the cause or the effect of a lack of information. Based on this classification, some methodologies add information to the component to reduce the cause of missing information, for example the component metadata approach, the retro-components approach, the reflective wrapper approach [80] and the Component Test Bench (CTB) [81] approach.
<table>
<thead>
<tr>
<th>Testing technique</th>
<th>Category</th>
<th>TDeriveBy</th>
<th>TExecBy</th>
<th>Metadata</th>
<th>MetaStruct</th>
<th>TSpecEasy</th>
<th>TSpecComp</th>
<th>HH</th>
<th>AM</th>
<th>FCSyntax</th>
<th>FCSemantics</th>
<th>TCAuto</th>
<th>TDAuto</th>
<th>TDTool</th>
<th>TC/TD</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT wrappers</td>
<td>BIT approach</td>
<td>CD</td>
<td>CU</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>BIT in maintenance mode operation</td>
<td>BIT approach</td>
<td>CD</td>
<td>CD/CU</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Self-testable software component</td>
<td>BIT approach</td>
<td>CD</td>
<td>CD/CU</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>STECC strategy</td>
<td>BIT approach</td>
<td>CD</td>
<td>CU</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Component behaviour prediction and monitoring</td>
<td>BIT approach</td>
<td>CD</td>
<td>CD/CU</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES-P</td>
<td>YES-P</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Component+ BIT</td>
<td>BIT approach</td>
<td>CD</td>
<td>CD/CU</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES-P</td>
<td>NO</td>
<td>NO</td>
<td>YES-P</td>
<td>YES-P</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Component interaction graph</td>
<td>Testable architecture approach</td>
<td>CU</td>
<td>CU</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Introspection through component test interfaces</td>
<td>Testable architecture approach</td>
<td>CU</td>
<td>CU</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Contract-based BIT</td>
<td>Testable architecture approach</td>
<td>CU</td>
<td>CU</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Integration testing framework</td>
<td>Testable architecture approach</td>
<td>TPT</td>
<td>TPT/CU</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Metadata for component testing</td>
<td>Metadata approach</td>
<td>CU</td>
<td>CU</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>UML test model, checking OCL constraints</td>
<td>Metadata approach</td>
<td>CU</td>
<td>CU</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>A framework for automated black-box testing</td>
<td>Metadata approach</td>
<td>CU</td>
<td>CU/CU</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Third-party component certification</td>
<td>Certification strategy approach</td>
<td>TPT</td>
<td>TPT/CU</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Component user certification</td>
<td>Certification strategy approach</td>
<td>CU</td>
<td>CU</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Component developer certification</td>
<td>Certification strategy approach</td>
<td>CD</td>
<td>CD/CU</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Interface probing</td>
<td>User’s specification-based testing approach</td>
<td>CU</td>
<td>CU</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES-P</td>
<td>YES-P</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Component deployment testing</td>
<td>User’s specification-based testing approach</td>
<td>CU</td>
<td>CU</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>A framework for software component testing</td>
<td>User’s specification-based testing approach</td>
<td>CD/CU</td>
<td>CU</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table IV. Component integration testing: a comparison of approaches.
On the other hand, some approaches require that components are implemented in such a manner so as to reduce the effects of missing information, for example BIT and the testable beans approach [51].

Vincenzi et al. [82] present an overview of problems encountered in testing CB software with a special focus on automation. They consider the two perspectives in CBD (i.e. developer and user), and the specific properties of OO components, i.e. encapsulation, inheritance and polymorphism, which further aggravate the component testing process for the user. For example, the Component Test Bench (CTB) [81] is a component verification tool that attaches a test specification in XML format to the component. The component user in CTB thus performs conformance testing of the component using XML specifications. A number of tools exist, for example JTest, Glass Jar Toolkit (GJTK) [82] but they work either for component unit or component conformance testing. Vincenzi et al. [82] have undertaken work on the implementation of a Java bytecode understanding and testing tool, i.e. JaBUTi. This tool attempts to conduct white-box testing using the Java bytecode of a component whilst certain test criteria are also applied. It presents a viable approach, but its use is limited only to components built using Java technologies and is not generally applicable.

4.3. Component integration testing approaches: which one to choose?

If the readers of this survey were looking for a definitive answer to the question of which is the best approach for testing their components, they would be dismayed by this article, since no such formula is found here. The fact is that there is no definitive answer to the question and, as often happens, no killer solution for testing CB software has emerged yet, and probably will not appear in the future. So the selection of one of the discussed approaches rather than another depends on many different factors and assumptions, such as the availability of specific information enabling one approach or the other, or the required quality of the software under development. Achieving high levels of reliability, for instance, will oblige the developer not to put all their confidence on tests defined by the component developer.

It is up to the market to take the final decision on which approaches will win the challenge of finding an agreement between the component developers and users. On their side, the users of the component would like to have details of the component that they buy. This means that they would certainly prefer components to come equipped with clear and testable models, with testable architecture and, possibly, encapsulating predefined tests. Obviously, adding this artefact to the component will cause a rise in its cost, thereby convincing some users to reduce their expectations and accept buying cheaper and less well-documented components at a lower price. Component developers will behave duplicitously; they would like to sell components without any extra information for at least two reasons. Of primary importance is reducing to a minimum the extra work necessary in order to deliver a component. Second, in general, they do not like to disclose details of the actual implementation of the component. However, reducing the accompanying information might convince fewer customers to buy a product, and this could lead some producers to increase the information and artefacts included with a component. It is easy to foresee that the balance between these different interests will also depend on the application domain. Probably cheaper and simpler components will be sold as they are, whilst more complex and critical components will be expensive and highly documented.

However, although certainly important, market factors are not the only feature that needs to be considered; or at least, they might also be considered as the effects and not only the causes of other motivations. In particular, when a component needs to be introduced into a system which demands high reliability, the final users will certainly need to test the component for themselves.
For this reason they would ask the component developer to produce models of the provided components or introduce probes that will enable them to better evaluate the adequacy of the component when deployed in the final environment, rather than trusting some predefined test cases, over which the customer has no control.

The approaches discussed in this survey can be classified into two groups according to who should develop the final tests that will be executed. To the first group belong those approaches that envisage definition of test suites by the component developer, which will be included with the component. In this group, all the BIT and testable architecture approaches can be put. The second group comprises those approaches that envisage the definition of the test suites by the component user. It is probably worthwhile discussing these two different ‘philosophies’ further, highlighting their respective possible advantages and drawbacks.

Placing the responsibility for deriving test cases on the shoulders of the component developer has clear advantages for the component user. This approach does not require, in fact, any particular testing skills on the part of the component user. The component can be bought, and the test suite launched in the final environment. Therefore, this approach seems particularly convenient for those users that do not have an adequate testing team and/or skills, and who, in any case, cannot gain any particular advantage from the availability of any ‘testing enabling information’ for the component, such as models. At the same time, the delivering of predefined test cases has some drawbacks even when ‘normal’ reliability is required. The problems become evident, by considering that the input domain of a component is, in general, infinite. So the developer has to take the decision on which test to include in the set of accompanying tests, and which to leave out. Obviously, these assumptions do not always coincide with the requirements of all possible deployment domains. Consider, for instance, a simple component that manipulates optical signals. Envisaging a general deployment of the component, the developer could decide to provide tests cases that verify the behaviour of the developed component when interacting with components, providing with equal probability a certain frequency for the signal. So test cases will be distributed in a uniform way, leaving some non-tested gaps. A user, instead, might be interested in using the component in a specific context, in which the incoming signals are only included within a specific and small range of the spectrum, because it will only be interacting with systems producing this kind of frequency. Therefore, in this case, the test cases delivered by the developer will not be sufficient to adequately test the component in the final environment. This is, indeed, the same old tale already told by Weyuker [41], and corresponds to the notion of CB adequacy formalized by Rosenblum [83].

The impression regarding these kinds of approaches is that they are most effective when precise reuse/deployment domains emerge. In this situation, in fact, it becomes more probable that the test cases defined by the component developer actually match those required by the component user.

The approaches outlined in the second group have, as yet, not been investigated sufficiently by the research community. The metadata approach can be included in this group (even though strictly speaking it is not a testing technique per se, but rather an enabling technique for such approaches). Research should establish what information can reasonably be provided by the developer and usefully applied by the component user to derive test cases. At the same time, more work on the development of tools for the automatic derivation of test cases from the provided information should be carried out. The approaches included in the second group are certainly more appealing for a system that requires high reliability and for which it is necessary to have a precise idea of what has been tested and to what extent. At the same time, the approaches in the second group require people with strong testing skills, who are able to derive, set and execute test cases on complex software systems. So it seems they could have a major impact on the development of more complex, reliable and expensive systems.
Finally, the certification strategy that does not completely fit in any of the groups mentioned
above, is that of envisaging the performance of the testing activity by a third party. This approach
could probably prove useful in complementing the first group of approaches. Its application could be
feasible in enlarging the number of domains in which the component can be deployed with increased
guarantees, especially if, for instance, certification agencies independently derive test cases for the
different domains and publish them for public use. However, the setting up of such an organization is
extremely expensive, and it is not clear if there will be room for such an approach.

5. CONCLUSIONS

While generally acknowledged as quite an attractive paradigm for increasing reuse and reducing
time-to-market, CBD also gives rise to a new critical problem in the area of software production,
generally referred to as the component trust problem [84]. This problem underlines the component
user’s exigency to gain confidence in 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
components delivered without the source code. However, as is also the case with components reused
internally in an organization, the difficulties of communication between teams and the lack of clear
documentation can produce, to some extent, similar effects.

This survey paper has 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 CBD has already been highlighted 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 its possible
application domains or what components will interact with the produced component, some kind of
testing against the component user’s specifications will always be deemed necessary [41]. In this
sense, it is also illusory to hope that the reuse of components drastically diminishes the need for
testing [34,41].

For the application of appropriate testing techniques, the packaging of suitable metadata along
with the component implementation would be advisable. Testing techniques have thus been proposed
which exploit metadata included with the component. Nevertheless, the existing metadata approaches
do not completely fulfil the requirements for component test information. In fact, the inclusion of
metadata must be accomplished in such a way as not to affect component implementation transparency.
Future research in this field requires the development of mechanisms to ensure that proper information
is available to the component users at integration testing time. It is also necessary to produce definitions
of automated processes so as to append metadata to the component before its final shipment, as well as
processes which display the metadata to the user without exposing implementation details.

As a final remark, it may be useful to remind the reader that ‘no single technique can produce
completely trusted components’ [84]. This paper has focused on component integration testing
approaches, but in practice an appropriate combination of approaches should be applied, both to
component construction, such as the previously mentioned design-by-contract [3,16] approach or
formal specification techniques, as well as 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 also
serves as a useful piece of input in addressing the burning problem of component trust.
ACKNOWLEDGEMENTS

Antonia Bertolino and Andrea Polini wish to acknowledge support by the Marie Curie Network TAROT (MRTN-CT-2004-505121), and by the European project PLASTIC (FP6 STREP 26955).

Condolences

Professor Jaffar-ur Rehman will never see the publication of our joint paper. Very sadly he died, along with his wife and four daughters, all below the age of seven, in the tragic earthquake of 8th October 2005 in Pakistan. Professor Jaffar-ur Rehman gained his PhD in Computer Science at the University of Pennsylvania, U.S.A. in 1997. In spite of a promising career overseas, he decided to go back to his beloved Pakistan, and there promote higher education and scientific research. He had been appointed Dean of the Department of Computer Science and Engineering at the Muhammad Ali Jinnah University, Islamabad, and devoted his energy to educating numerous Masters and PhD students. It was a pleasure and an honour to work with him, and his memory will accompany us for many years to come. His untimely departure has left a void in the software engineering community.

REFERENCES


DOI: 10.1002/stvr