WCT: a Wrapper for Component Testing

Antonia Bertolino, Andrea Polini[*]

ISTI-CNR, Area della Ricerca CNR di Pisa, Via Moruzzi, 1
56124 Pisa, Italy
{bertolino, a.polini}@iei.pi.cnr.it

Abstract. Within component based (CB) software development, testing becomes a crucial activity for interoperability validation, but enabling test execution over an externally acquired component is difficult and labor-intensive. To address such problem, we propose the WCT component, a generic test wrapper that dynamically adapts the component interfaces, making the test execution independent from any specific component implementation. The wrapper does not require that the component implements any specific interface for testing purposes, it permits to easily reconfigure the subsystem under test after the introduction/removal/substitution of a component, and helps optimize test reuse, by keeping trace of the test cases that directly affect the wrapped component.

1 Introduction

Component based (CB) methodologies are today largely used in all the classical engineering branches. Their adoption is mainly motivated by the need to get more predictable timing and costs of the development phase. Although the introduction of a CB paradigm also in the software engineering branch has been advocated for long time [1], it is only in the last years that we can observe significant advances towards the real applicability of this methodology to software production. Proof of this progress is the advent of the first successful component models such as COM+/.Net, EJB, JavaBeans, CCM. However, in spite of these advances, we can certainly say that CB production is not Software Engineering state of practice yet. What is still lacking for the real take-up of the CB paradigm is a major revision of the software process, to address the peculiarity of a CB production. In [2] a list of important challenges in the CBSE is discussed.

A first and basic difference between the traditional production methodology and a CB one is in the non-deterministic distribution, in time and in space, of the CB development process. In fact in CB production the “pieces” that will constitute the final assembled system can be acquired from many other organizations, that do not necessarily communicate or synchronize with each other. Moreover the acquired elements are not in general developed as a consequence of a specific requirements specification, instead are retrieved from the market as pre-built elements.

In this scenario we can distinguish, at least, two different stakeholders. The first is

[*] Andrea Polini’s PhD grant is supported by Ericsson Lab Italy in the framework of the Pisatel initiative (http://www.iei.pi.cnr.it/ERI).

the **component developer**, who is engaged in the construction of the components that will be released to third parties. The second kind of stakeholder is represented by the **system constructor**; this is himself/herself a software developer, who builds a system by assembling together some components, either internally developed or externally acquired. Certainly the CB paradigm mainly affects the process of the system constructor. In [3] we have outlined a possible iterative process that tries to address the new requirements of a CB production; another process is shown in [4]. Within the process of the system constructor, we can note the presence of two new related phases, referred to as the ***searching phase*** and the ***selection phase***.

Aim of the searching phase is to find one or more components that can be assembled in the system to address some specific functional requirements. This search is not an easy job and requires specific tools; in particular, one active research direction tries to identify what kind of information (and how) can be attached to the component by the component developer, so to automate, as much as possible, the task.

Aim of the selection phase is to choose the component, among those identified by the previous phase, which is the most suitable for the system under construction. It is obvious that in the general case the result of the searching phase cannot be a single component for which the full conformity with the searched one can be guaranteed; therefore a validation phase is necessary to evaluate the components and to select the most promising one.

In practice these two phases are particularly hard, since it is generally the rule to provide the component without the source code (black-box); moreover, the attached documentation is often incomplete or only limited to the explanation of how to use the provided API.

A direct consequence of this lack of information is generally referred to as the “component trust problem”, to indicate that the system constructor, who acquires a component from the component developer, needs some means to get a better understanding of what the component does and how it behaves. In this context, our perspective is in studying **how to use testing techniques for the validation and selection of the components**, so contributing to the mitigation of the component trust problem. In particular we are studying how the testing activities and tools must be reviewed and augmented to take in account the peculiarities of a CB development.

So far we have used the term “component” in a complete general form. A reference definition for the concept of component is still debated and in the literature there is not yet a universal agreement. An often reported definition is that provided in [5]: “*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*”. With respect to this definition we take a more simplified view, and, as in [6], we identify a component with a system or subsystem developed by one organization, deployed by one or more different organizations, and possibly provided without the source code. According to this definition, we will consider also a class or a set of classes as a particular example of a component.

Regarding the technical aspects of the methodology, we require that the used component model foresees basic introspection mechanisms, because we need to retrieve at run-time information mostly referring to the component interfaces. As a consequence, we have adopted the Java language.

2 Related work

As recognized in [7], several techniques can be usefully combined to reduce the component trust problem (formal specification, Design by Contract, testing and others), and further research is still necessary. Different approaches are under study to address the problem; in particular some authors suggest to add supplementary information to the component, in the form of metadata, with the objective to increase the analysis capability of the system constructor [6][8]. Another approach proposes to establish independent agencies that act as software certification laboratories. The main duty of this agencies would be the certification of the components (as done in many other engineering disciplines) so to increase guarantee of adequate behavior [9].

Regarding testing, in our knowledge there is not much work addressing the problem: we list three different proposals. A first approach proposes to embed test cases in the component itself (Built-In Tests) in the form of methods externally visible [10]. A disadvantage of this approach is the size growth of components that have to contain also code specific for the testing phase. To overcome this problem, another approach introduced the concept of a testable architecture. This architecture foresees that the components implement a particular interface for testing purposes, that permits to execute pre-built tests without the necessity to include them in the component code [11]. Finally in another approach [12], the authors propose to provide the component user with the test cases that the component has undergone in the form of a XML file. In this case the authors also provide, attached to the component, a tool for the re-execution of the test cases.

It is worth noting that, differently from the listed approaches, we do not impose that a component implements any particular interface, so to make this task less costly and more generally applicable. Moreover our aim is to provide the component user with tools to simplify the execution of test cases developed on the base of the specification for the searched components, permitting at the same time the re-execution of the test cases provided by the component developer.

3 The testing phase in CB development

It is our belief that a revision of the development process is a necessary precondition for the effective success of a CB production, and this revision must also concern in particular the testing stage. In [3] we have outlined a possible direction for the revision of the testing process. In particular we have highlighted that the traditional three phases of the testing process (unit, integration and system test) can be revisited in terms of three corresponding new phases, respectively referred to as ***component***, ***deployment*** and ***system*** test.

Briefly, in the component test phase, the component is tested in the component developer environment, to ascertain the conformance of the component to the developer specifications. However, as many authors recognized [13], the tests performed by the developer are clearly inadequate to guarantee the dependable use of the component in the final application environment (that of the system constructor). The testing should be repeated also in the final environment, both for the component as a single element and when integrated with the other interacting components. Therefore the deployment test phase performed by the component user results composed of two sub-phases: in the first phase a selected component is evaluated directly invoking the provided API; in the second phase, the component is inserted in a subsystem and then integration-tested as an element of the subsystem. The last stage of system test does not show notable differences with respect to the traditional one.

From the above discussion, the system constructor process must involve at least two kinds of independents teams. The first is the searching team, who on the base of precise specifications looks for components that, in their understanding, correspond to the searched ones. The second team instead, on the base of the specification, develops test cases that can be used to ascertain the conformance of the found selected components to the searched ones.

Probably the major costs induced in CB development can be ascribed to the two illustrated phases, and then any little improvement, thanks to suitable techniques and tools, can bring great benefits. In particular it is worth noting that a lot of the work made by the two kinds of teams can be in principle carried on in parallel, and therefore it is important to adopt a methodology that permits the effective exploitation of this possibility.

4 WCT, a Wrapper for Testing Purposes

In this section we introduce the structure of the WCT, a wrapper to be integrated in a CB testing platform, to permit the easy reconfiguration of the subsystem under test when components are introduced or removed. A WCT basic feature is flexibility, in that we do not require that the component to be wrapped within a WCT component implements any particular interface.

As we have said, the testing stage is probably the most expensive phase in component based production. Hence, means for reducing its cost and “gaining time” are extremely important. We can identify two major costs in the testing stage. The first can be identified in the effort to set up the system into a configuration suitable for executing the tests, the second is the controlled execution of the test cases on the subsystem.

To reduce these two sources of costs we have thought to intervene at the level of the component to be integrated, providing a wrapper model that can be used to bind the component, thus saving time and effort in two important respects:

1. it is easier to set up the test configuration, e.g., to reconfigure the component/subsystem under test adding components, in the place of stubs, or to substitute components with new candidates
2. it is easier to identify the “indispensable” test case to be rerun when a candidate component is substituted with a more promising one.

In the following of this section we present in detail the proposed approach.

4.1 WCT: the Constituent Elements

The WCT wrapper has a fixed structure that is independent from the connected components. It can be removed, e.g., for performance purposes, when a final configuration is identified, to be substituted with a static wrapper: clearly the test wrapper can be very useful to develop the permanent wrapper, because it can be taken as a reference model, like a prototype.

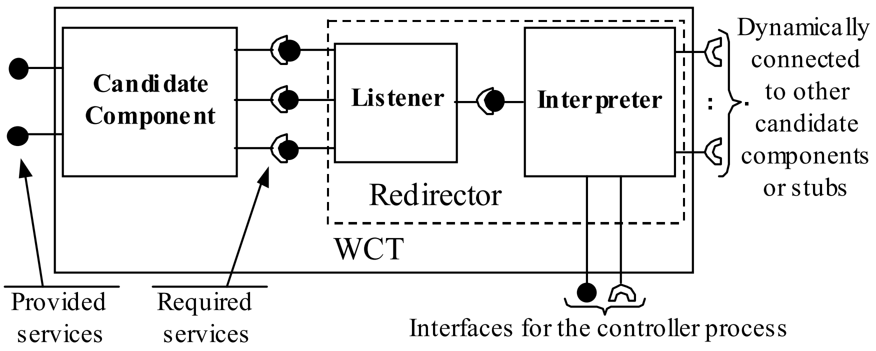
Candidate Component
Listener
Interpreter
Redirector
WCT
Dynamically connected to other candidate components or stubs
Provided services
Required services
Interfaces for the controller process

Fig. 1. Schema of the WCT illustrating the constituent elements and the various connections.

The approach forecasts to connect all the required interfaces of a candidate component to a "standard" component named the ***Redirector***. As we will see this consists of two parts: one depending on the candidate component, and the second fixed for each instance of the Redirector. As illustrated in Figure 1, the composition of a candidate component and of the Redirector forms a new component, named ***WCT***, that provides to the external world the same services of the candidate component but does not statically requires any particular service. Only at run-time, to realize the subsystem under test, the binding between the various components is established as the result of the appropriate interpretation of a XML document. Therefore a subsystem under test, at each time, results from the composition of more WCTs. The configuration of the subsystem can be modified also at run-time, e.g., by inserting a candidate component in the place of a stub or by substituting a component with a more promising one.

To clarify the approach and how it works, it can be useful to describe in detail the structure of the WCT and in particular of the Redirector. The duty of the Redirector is exactly to *redirect* (then its name) at run-time the invocations made by the candidate component towards other candidate components (or otherwise stubs) that opportunely provide the required services. To put in place the redirection, it uses three main elements (see also Fig. 1):

1. the Listener class
2. the Interpreter class
3. a XML file named "XMLAdapter"

In the following, we describe in detail these elements. We refer to the ***associate component*** to indicate the component that is contained in the same WCT of the Redirector, and to the ***attached components*** to indicate the components that provide the required services to the associate component.

The Listener Class. Main target of the class Listener is to isolate the associate component from the attached components. In other words, this class act as a proxy, postponing to a subsequent moment the instantiation of one or more real attached components, which will be able to manage the invocations made by the associate component. To do this, the Listener class has to implement all the interfaces that the associate component requires, but instead of giving, for each method, a real implementation, it delegates this task to the Interpreter class (as we will see the latter, in turn, does not implement the method, but "knows who" can opportunely serve the invocation). The implementation of the class Listener depends from the specific candidate component, having to implement the specific interfaces, but the implementation can be totally automated with the support of suitable tools[1]. In fact, the scheme of the methods is completely fixed, and for each method the only duty is to redirect the invocation to the Interpreter class packaging the parameter in a vector of objects. In the following piece of Java code, we show a scheme for the Listener class that implements two interfaces A and B, each of which requires the implementation of one method, respectively named "a" and "b".

```
public class Listener implements A,B{
  private Interpreter interpreter;
  public Listener(Interpreter interpreter) {
    this.interpreter = interpreter;}
  public t1 a(tA1 p1,…,tAn pn) {
    Object[] parameters = new Object[] {p1,…,pn}[2];
    return (t1)interpreter.execMethod("a",parameters);}
  public t2 b(tB1 p1,…,tBn pm) {
    Object[] parameters = new Object[] {p1,…,pm};
    return (t2)interpreter.execMethod("b",parameters);}
}
```

The Interpreter Class. If the duty of the Listener classes is to "deceive" the candidate component simulating the presence of the attached components, the main duty of the Interpreter class is to redirect, at run-time, the invocations towards an implementation that can really give suitable answers. The redirection is based on the information retrieved from the XMLAdapter, that contains the rules based on which the redirection will be based (in the next section we explain in major detail how to draw up this file).

To identify the method, or methods, that must be invoked, as a consequence of a request made by the associate component, the instance of the Interpreter class uses the introspection mechanisms provided by the component model, trying to retrieve information from the attached components. In particular to apply the model we exploit the Java introspection mechanism which permits to retrieve information from all the public methods and to invoke, on the base of these information, a selected method. The use of reflection permits the easy reconfiguration of the system, allowing for the introduction of new candidate components taking the place of a stub or substituting a candidate component with a more promising one.

The class Interpreter presents two main public methods. The first method is invoked by the controller of the testing process that provides the name of the XMLAdapter. Obtained the name, the method reacts by parsing the file (to do this the

[1] It is worth noting that in the specific case of a clash in one or more of the method names enclosed in the required interfaces, it will be necessary to use more then one Listener class.

[2] In Java we need also to use a wrapper type when the parameter is of a basic type.

interpreter contains an instance of a suitable XML parser) and storing the retrieved information in appropriate data structures. Through the invocation of this method, the controller of the testing process can reconfigure the subsystem under test.

The second method, instead, is invoked by the associated listener and is appointed to redirect the invocations, made by the associate component, to the opportune method (methods) of the attached components. To perform this task the method uses the information stored in the data structures by the XML parser, and uses the reflection mechanisms on the attached components.

We think that this method, that has the control over the methods invocations of the associate component, can be augmented to perform other useful tasks. A first task that comes to mind is the recording of method invocations to keep trace of each test case execution. This tracing facility may result particularly useful when we consider the replacing of a stub with a real component or the substitution of a candidate component with a more promising one. In fact, having recorded the test cases that stimulate a particular method, in the case of a replacement we can re-execute only the test sequences exercising the methods affected by the substitution.

The XMLAdapter file. Aim of the XMLAdapter is to provide a means by which the searching teams can explicitly formulate the correspondences between a client component (the component that needs a service) and a server component (the component that provides the services). Several levels of mismatches can exist between a searched component and a found one. These are immediate consequences of the selection process. In fact, we suppose that the selection of a candidate component has to follow some "semantic" principles, in the sense that the choice is mainly based on the understanding of what a component does, understanding that the searching teams must derive from the documentation associated to the component. This "choice" obviously implies the necessity of suitable wrappers to actually permit the "syntactic" interaction among components. Our approach is to codify the rules that establish the correspondence in the XMLAdapter, a XML file with definite tags, that can be parsed by the Interpreter to redirect the invocations made by the associated component. We have identified several levels of mismatch between the client and the server components, that can be overcome with the use of the XMLAdapter:

1. differences in the methods names and signatures:
   a. the methods have different names
   b. the methods have the same number and types of parameters, but they are declared in different order
   c. the parameters have different types, but we can make them compatible, through suitable transformations. It can be also necessary to set some default parameters.
2. one method in the client component corresponds to the execution of more than one method, in one or more server components.

Regarding the structure of the XMLAdapter, it can be divided in two parts. The first part specifies the component instances (that can also be remote) that must be used and manages the invocations of the associated component. In the second part, for each invocation of the associated component, the corresponding sequence of methods and transformations, that must be invoked in the attached components, are specified.

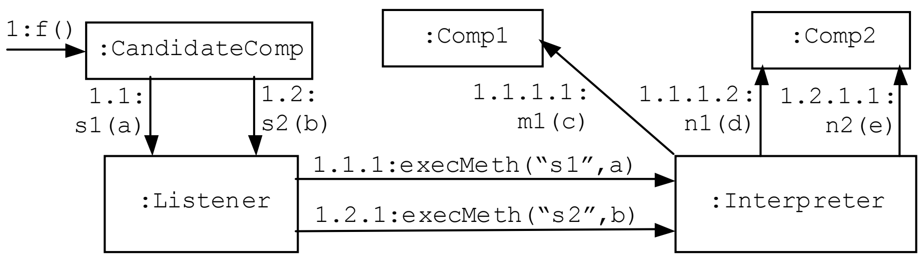

Fig. 2. Collaboration diagram that illustrates the interaction between the WCT elements and two attached components. In particular to provide service `f` the associated component requires two services, `s1` and `s2`. To provide this services the Interpreter instance opportunely transforms the invocations of `s1` and `s2` into sequences of invocations of the services provided by the two attached components `Comp1` and `Comp2`.

5 Discussion and future work

In this section we explain how the WCT component can be used in CB development. In particular we outline the scheme of a general platform for component deployment testing, within which the WCT component can be employed for assembling and testing the components.

The general structure of the platform is shown in Figure 3. The main target of this platform is to provide the capability to exploit the potential parallelism in the activities of the searching and testing teams. In fact, following the approach proposed in [14], we suppose that the test cases are defined by the system constructor, before the component is acquired, on the basis of a virtual component specification. The structure of this virtual component can be derived from the system specifications, which are also used as a reference by the searching teams.

For the purposes of presentation, we have spoken so far of the acquisition of a single component. Indeed, when we have to assembly a “piece of software” within a CB system, in our approach there is no difference between a “monolithic” component or instead a “composite” component, made by more opportunely connected components. Since a subsystem can be externally viewed as a single component, the testing teams can use the same platform to codify the test cases for exercising either a component, or a composite subsystem (such as the gray box in Figure 3). To do that the subsystem under test is viewed itself as a component with a specified virtual interface that expresses the functionality that a real instance of the subsystem has to provide. On the basis of the virtual interface the test cases are codified and stored in a suitable repository to be later used when the subsystem will be instantiated. It follows that the derivation of the test cases and the searching activity can largely proceed in parallel, since the codification of the test cases in this way is not dependent from any particular implementation.

As we can see, the platform shown in Figure 3 is composed of three main elements. The first is the Tester Component, widely described in [14], that permit the application of the test cases established by the testing teams on the subsystem. The second element of the approach is the Controller, which is a distributed interface that permit to control the testing process, in particular giving to the testing teams a means to add developed test cases, and to the searching teams a means to modify the subsystem structure. The use of the WCT components, as constituent elements of a subsystem under test (that constitute the third element in the figure), is particularly useful at this stage. In fact, each time a searching team identifies a potential candidate component, to be able to insert the latter in the subsystem it is only necessary to modify the XMLAdapter associated with those components that need to invoke the new inserted one. However, when the introduction/removal/substitution of a component has also effect on the external interface of the subsystem, it is necessary that the set of test cases used by the Tester Component is accordingly modified.

The WCT can also be fruitfully employed to reduce the number of test cases to be re-executed when a new component is introduced. To do that, the Interpreter in the WCT can keep trace of the methods that are invoked by the associated component during the execution of a test case, and communicate them to the Controller. This information can be opportunely stored by the Controller and then used to establish the set of regression test cases, when a new component is inserted and an invocation is redirected. Concluding, in this paper we have briefly revisited the testing process in CB development and we have highlighted how the testing activity can result particularly useful in the component selection phase. We have then presented the notion of a test wrapper that can be usefully employed by a system constructor to test sets of integrated components (subsystem) within his/her environment.

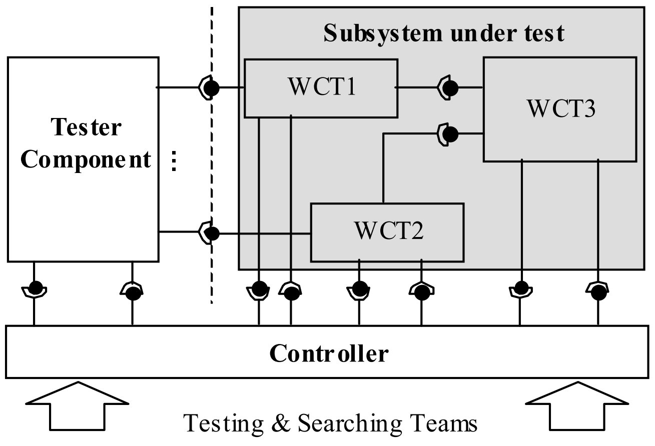

Fig. 3. The whole logical structure of a possible subsystem under test composed of three components. The figure shows the Tester Component and the Controller, that constitutes the interface towards the Testing and Searching teams.

Favorable features of the proposed approach are that: it does not require that the component implements any specific interface for testing purpose; it permits an easy and fast reconfiguration of a subsystem after the introduction/removal/substitution of a component within the subsystem; and finally it is possible to introduce in the WCT useful features for regression testing purposes: in particular, we have outlined how it is possible to reduce the number of test cases to re-execute at each reconfiguration.

The WCT wrapper is part of a long term research project addressing CB testing. Our aim is to employ the WCT component within a more general platform for CB testing currently under development.

In the next future we will work at the implementation of the mentioned testing platform. In particular we intend to formalize as much as possible the drawing up of the XMLAdapter by means of suitable graphical interfaces that partially automate the process. It is also our objective to reuse as much as possible existing tools, as for instance Junit [15], a framework developed for the early testing of OO code but that can be partially revisited in the CB testing field. Lastly, we also plan to validate the approach within industrial CB production using real case studies.

6 References

1. McIllroy, D.: Mass Produced Software Components. In P. Naur and B. Randall Eds, Software Eng.: Report on a Conf. by the NATO Science Committee, pp 138-155, Brussels, 1968.
2. Crnkovic, I.: Component-based Software Engineering – New Challenges in Software Development. Software Focus, John Wiley & Sons Eds, December2001
3. Bertolino, A., Polini, A.: Re-thinking the Development Process of Component-Based Software. ECBS02 Workshop on CBSE, Composing Systems From Components, April 10-11, 2002, Lund, Sweden.
4. Crnkovic, I.: Component-based Software Engineering – New Paradigm of Software Development. Invited talk & Invited report, Proc. MIPRO 2001, Opatija, Croatia , May 2001.
5. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.
6. Orso, A., Harrold, M.J., Rosenblum, D.: Component Metadata for Software Engineering Tasks. In W. Emmerich and S. Tai Eds. EDO2000, *LNCS 1999*, pp.129-144.
7. The Trusted Component Initiative: http://trusted-components.org, Access date 2002-09-21
8. Stafford, J.A., Wolf, A.L.: Annotating Components to Support Component-Based Static Analyses of Software Systems. In Proceedings of the Grace Hopper Celeb. of Women in Computing 2001.
9. Voas, J.: Developing a Usage-Based Software Certification Process. *IEEE Computer*, August 2000, pp. 32-37.
10. Wang, Y., King, G., Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance. In Proceedings of the 3rd ECSMR, 1999.
11. Gao, J., Gupta, K., Gupta, S., Shim, S.: On Building Testable Software Components. In J. Dean, and A. Gravel Eds, Proceedings of ICCBSS 2002, LNCS 2255, pp.108-121.
12. Morris, J., Lee, G., Parker, K., Bundell, G.A., Lam, C.P.: Software Component Certification. *IEEE Computer*, September 2001, pp.30-36.
13. Weyuker, E.: Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, Sept./Oct. 1998, pp. 54-59.
14. Bertolino, A., Polini, A.: A Framework for Component Deployment Testing. Proceedings of the ACM/IEEE International Conference on Software Engineering ICSE 2003 (to appear), Portland, USA, May 3-10, 2003
15. JUnit: http://www.junit.org.