Electronic Notes in Theoretical Computer Science 82 No. 7 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 11 pages

Integration of "Components" to Test Software Components

Antonia Bertolino [1,2], Eda Marchetti [1,3], Andrea Polini [1,4]

Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR Pisa, Italy

Abstract

We present an ongoing research project aimed at developing a framework for component-based testing, in which we re-use and suitably combine some existing tools: the system architecture and the components are specified by the UML, and specifically the recently proposed UML Components methodology; the test cases are derived by applying the Cow_Suite, an environment for UML-based testing, previously conceived for the integration testing of OO systems; and the tests are codified and executed within the CDT, a framework under development, allowing for the decoupling between the abstract specification of tests, which is made against an architectural model, and their concrete execution, which needs to take into account the component implementations.

1 Introduction

Component-based (CB) development is today one of the focal trends in software production. Although in the last years the first commercial frameworks, e.g., .NET or J2EE, have appeared, research in this area is far from having been completed, and many topics, such as component specification, development tools, or performance predictability, are still open.

Our research aims at building a general framework for enabling the validation of component-based (CB) systems by testing them against the system architectural specifications. Our study encompasses both the functional and non-functional qualities of a system, and this paper focuses on methodologies and tools for the functional testing tasks [5].

[1] This research is carried on within Pisatel Lab under a cooperation agreement with Ericsson Lab Italy, Rome.

[2] Email: bertolino@iei.pi.cnr.it
[3] Email: e.marchetti@iei.pi.cnr.it
[4] Email: a.polini@iei.pi.cnr.it

[5] A companion paper [3] discusses the related approach for performance-related parameters.

Our goal is specifically to lay out a test environment in which the system developer can: (i) derive the test cases and (ii) codify and execute them. To do this, we are taking a pragmatic approach: we try to combine, of course by doing the necessary adaptations and extensions, two tools we have developed in previous projects: the Cow_Suite for test derivation, and the CDT for test execution within a CB development process. We illustrate here the respective features of these two tools, and then discuss how and why we believe that by combining them we can obtain a general framework for CB testing.

The Cow_Suite [2] is a tool originally developed in the context of OO integration testing. It is based on the widespread UML modeling notation. Although UML was not conceived having a CB paradigm in mind, it is very flexible and provides suitable mechanisms for extensions. In fact, some works already start investigating appropriate UML applications for specifying components and component assemblies. We follow the recently proposed UML Components methodology [8]. Hence, we need to re-shape the input models to Cow_Suite to allow for the analysis of specifications according to the UML Components, and to revise the procedure followed by Cow_Suite for test case generation.

The CDT [4] is a Java-based framework we have recently developed in order to facilitate component deployment testing by CB system developers: the framework is conceived so that the system developer can early codify the test cases (derived from the system architecture), and later (re)execute them each time a component instance is plugged into the system. The advantage of CDT is that the wrapping required for launching the tests on the component implementation is automatically performed by exploiting the reflection mechanism of the Java language.

While the principles of integration have been settled, the combination of Cow_Suite and CDT is currently under implementation. Therefore we cannot yet provide here a definitive picture, or empirical results. This is meant rather as a working paper to present the two tools, and the main directives we intend to follow for their combination, adopting the UML Components methodology as the input modeling notation. In this sense, our research approach is itself component-based, and an overview is provided in Figure 1.

In the next section we introduce the component testing problem. Then, in Section 3, we present in detail the existing tools that we are going to integrate. In Section 4 an overview of the proposed approach is provided. Conclusions are in Section 5.

2 Component Testing

A central challenge in CB practice is the development of a new process that effectively addresses the peculiar features of this new methodology. In CB production a system will result from the synthesis of the work of many, mainly uncoordinated, stakeholders (the system assembler and the various component developers). As a consequence a new problem arises, generally referred to as the *component trust* problem. This is relative to the fact that the system assemblers have to make a "vote of confidence" each time they use a component produced externally; the problem is exacerbated by the fact that components are generally shipped as black boxes. What is needed are means allowing the system developers to evaluate the component against their specific needs and within their application environment. There is no univocal solution, but several methodologies can be used, such as Design-by-Contract, formal methods, and testing.

In particular, testing externally produced components is a quite delicate and difficult task. Suitable techniques that increase the testing and analysis capability of the system developer (i.e., the component user) must be developed. Several approaches have been proposed, which we classify as:

- *the metadata approach*: it is suggested that the component developer includes with the component some information specifically aiming at increasing the customer capability for analysis and test derivation [14]. For instance some UML diagrams could be useful.
- *the certification strategy approach*: the establishment of independent certification agencies is advocated, with the specific duty of evaluating the components for the perspective users [16].
- *the built-in test approach*: the component is shipped instrumented with test cases, directly coded within the component itself. The customer can then re-execute them with the component plugged in the target environment [17].
- *the testable architecture approach*: this can be seen as an improvement of the previous case, as it limits the component size growth implied by the built-in tests by providing a specific component as a testing interface [10].
- *the customer specification based approach*: this approach foresees that the system developer derives the test cases against the system specifications using a virtual definition of the component, and then uses them to evaluate the candidate components [4].

We take here the last approach; further details will be presented in the following.

3 Existing tools

Our goal is to develop a general framework suitable for test derivation and execution in a CB environment. As said, we intend to build this framework by integrating some available tools and methodologies, as sketched in Fig. 1:

- UML Components [7] provides a modelling notation and a process for specifying CB systems;
- Cow_Suite [2] is a UML-based test environment, originally developed in the area of OO testing;

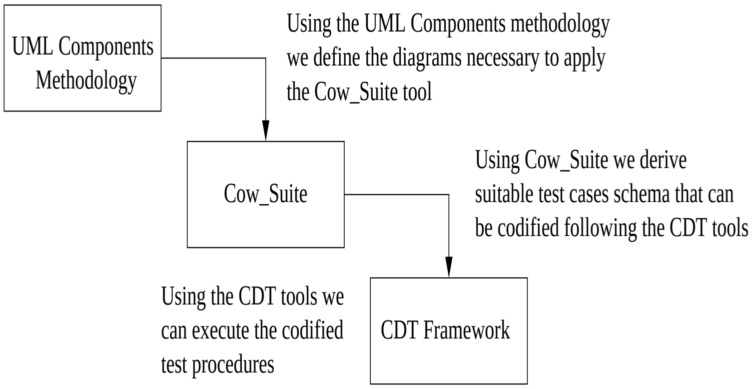

Fig. 1. Overview of the testing framework

- the CDT framework [4] (including the WCT component [5]) supports the system developer during the various steps of the validation phase of a component based system.

3.1 UML components

While UML is today the *de facto* standard notation for the analysis and design of Object Oriented systems, its application to the specification of CB systems is just starting. We adopt the methodology proposed by Cheesman and Daniels [7], called the UML Components, which focuses both on the representation of the components and on the process development applicable for this purpose. We report briefly the mainly details of the specification process adopted in [7], divided in interacting workflows as suggested by RUP [11].
The tasks of the requirement workflow are the business concept model and the use case model. The former is a conceptual model, which specifies the key concepts, their relations and a common vocabulary useful for avoiding misunderstanding and ambiguities. It is represented by a class model, but the classes involved, as well as their associations, are only conceptual and not related with the specification. The use case model represents instead the interaction of the system with the external users. It is represented by a Use Case Diagram, in which each Use Case is related to a different requirement. The system behavior and main exceptions are represented for each Use Case in the associated scenario, following the textual structure of the Cockburn's Use Cases [8].
The specification workflow is subdivided into three phases: (i) the identification of the components: starting from the requirements, an initial system architecture is produced; (ii) the interactions among the components, which identify the system operations and responsibilities; (iii) the specification of the components, which specifies the operations and interfaces of the components themselves. A business model, represented by a class diagram, is used for modeling the business information. The involved classes are defined at the specification level, with no relation to a specific language. The notation used

for the component interfaces differs from that defined in the standard UML, in which the interfaces represent implementation constructs typical of the OO languages and that do not require attributes or associations. In the UML Components, an interface specification consists in: the type, the information model (the attributes, the interface roles in the association and their types), the specification of the operation (prototypes, pre- and post-conditions), and the invariants. All this information is grouped together in a package representing an interface specification, which can also import information from other packages.
In UML Components, even the concept of a component is quite different than in the standard UML, because it is completely independent from the implementation. To differentiate the specification of a component from its implementation or the installed component, a new stereotype <<`comp spec`>> is introduced which has a set of interface types. The ways in which the components interact via the interfaces are finally described using collaboration or sequence diagrams.
Considering the provisioning workflow, it is aimed at ensuring that the released software is consistent with the given specification of the components. For this purpose the components can be implemented, bought, readapted or derived from the integration of existing software.
Finally the integration workflow connects together the various components, the user interface, the application logic and the existing software to obtain an efficient application.

3.2 Cow_Suite

Cow_Suite [2] is a methodology for the planning and generation of UML-based test suites, since the early stages of system analysis and modeling. Cow_Suite stands for *COW*test plu*S UIT* Environment, and as the name implies it combines two original components:

1. Cowtest (Cost Weighted Test Strategy) is a strategy for test prioritization and selection;
2. UIT (*U*se *I*nteraction *T*est) is a method to derive the test cases from the UML diagrams.

These two components work in combination, as Cowtest helps decide which and how many test cases should be planned from within the universe of test cases that UIT could derive for the system under consideration.

3.2.1 Cowtest

This strategy is based on the analysis of the UML design, and in particular on the Use Case (UC) Diagrams, Sequence Diagrams (SDs) and Collaboration Diagrams (CDs). Starting from the main Use Case Diagram onwards, Cowtest considers each developed diagram and, by using their mutual relationships, organizes the model elements into a defined structure. In particular the Actors, UCs, SDs, CDs are organized in an oriented graph called the Main Graph, and the packages and their components in another one called the Design Graph. Both of them are then explored by using a modified version of the Depth-First Search algorithm [9] for producing a forest of several Main Trees and Design Trees which constitute the basic hierarchical structures of the Cow_Suite approach. Each tree level evidences a different degree of detail of the system functionalities and represents for us a specific integration stage. The nodes of the derived trees are annotated with a value, called the weight, belonging to the [0,1] interval and representing its relative "importance" with respect to the other nodes at the same level: the more critical a node the higher its weight. Different criteria can be adopted to define what "importance" means for test purposes, e.g., the component complexity, or the usage frequencies (such as in reliability testing [12]). Finally, Cowtest calculates the final weight of each node relative to a selected integration stage, i.e. the product of all the nodes weights on the complete path from the root to this node. These final weights are used for choosing amongst the tests to execute, in two different manners:

1. by fixing the number of test cases: then Cowtest selects the most suitable distribution of the test cases among the functionalities on the basis of the leaves weights.
2. by fixing a functional test coverage (e.g. 80%) as an exit criterion for testing. In this case Cowtest can drive test case selection, by highlighting the most critical system functionalities and properly distributing the test cases.

3.2.2 UIT

Largely inspired by the well-known Category Partition method [13], UIT was originally conceived [1] for UML-based integration testing of the interactions among the objects, or objects groups, involved in a SD. Within the Cow_Suite approach, a simplified version UIT_sd is employed, by which test derivation is applied once for each SD as a whole and not by separately considering the objects involved. UIT_sd automatically constructs the Test Procedures using the information retrieved from the UML diagrams. A Test Procedure instantiates a test case, and consists of a sequence of messages, and of the associated parameters. UIT_sd is an incremental test methodology; it can be used at diverse levels of design refinement, with a direct correspondence between the level of detail of the scenarios descriptions and the expressiveness of the Test Procedures derived. All the SDs relative to a selected integration stage constitute the basis for the UIT_sd method. For each selected SD, the algorithm for Test Procedures generation is the following:

1. *Define Messages_Sequences*: Observing the temporal order of the messages along the vertical dimension of the SD, a Messages_Sequence is defined considering each message with no predecessor association, plus, if any, all the messages belonging to its nested activation bounded from

the *focus of control* [15] region.

2. *Analyze possible subcases*: the messages involved in a derived Messages_Sequence may contain some feasibility conditions (e.g., if/else conditions), formally expressed using the OCL notation [15]. If these feasibility conditions exist, a Messages_Sequence is divided in subcases, corresponding to the possible choices.
3. *Identify Settings Categories*: the Settings Categories are the values or data structures that can influence the execution of a Messages_Sequence.
4. *Determine Choices*: for each Message choices represent the list of specific situations or relevant cases in which the messages can occur; for the Settings Categories, they are the set or range of input data that parameters or data structures can assume.
5. *Determine Constraints among choices*: to avoid meaningless or even contradictory values of choices inside a Messages_Sequence, constraints among choices are introduced.
6. *Derive Test Procedures*: for every possible combination of choices, for each category and message involved in a Messages_Sequence a Test Procedure is automatically generated.

3.3 CDT

This section shortly summarizes our ideas in [4,5]. CDT (*C*omponent *D*eployment *T*esting) was motivated by the objective of allowing a system constructor to early codify the integration test cases of a CB system, even before a searched component is identified. In our approach, we foresee that the system developer establishes the architecture of the system in terms of virtual components with precise interfaces. This specification is used by: a searching team to look for suitable components, and a testing team to develop the test cases. In this view, we distinguish two levels of component deployment testing:

1. a single virtual component (which may be obtained by one or more real components) is tested in isolation by the system developer;
2. an integrated set of virtual components (a subsystem) is tested in the final application environment.

To use CDT in either case, suitable test cases must be derived for each component and/or subsystem that we intend to test. As soon as a real component/subsystem is identified, it can be "attached" to a CDT by providing some information necessary for customizing the test cases that were derived for the virtual components.
It is important that the test cases are stored in homogeneous groups, so that they can be selectively reloaded. To do this we have reused the structure of *JUnit* [18], a well-known Java testing tool, originally developed in the field of eXtreme Programming (XP).

CDT can be structured in three groups of elements:

- *testing classes*: the elements in this group concern the specification of the test cases and the interfaces of the virtual components. Elements within a same class refer to the testing of the same component/subsystem.
- *the package* ***it.cnr.testing***: this package contains all the CDT logic that does not depend on the specific context. By means of this package, the invocations made by the test cases to a virtual component can be redirected towards the corresponding instances of the real components.
- *the adapter elements*: this group includes the elements that adapt a virtual component to the instances of the real components implementing it. In the current implementation, the adaptation is made via a XML file compiled with the information needed after the real component is available.

To facilitate the assembly of a component, or of a subsystem, we have also developed the WCT (*W*rapper for *C*omponent *T*esting) [5]. WCT yields a structure similar to CDT, but its aim is to dynamically redirect the invocations made by the component, when it needs a service, and to dynamically collect information on the execution, as, for instance, the services requested in consequence of a service supply.

4 The proposed approach

In this section we show how we foresee to integrate the two "components" of CDT and Cow_Suite, in order to obtain a component testing environment. We discriminate between two different levels of test cases.

4.1 Test of the single virtual component

At this stage each component is tested alone, by means of suitable stubs when necessary. We use, for each component, its UML specification to derive the set of test procedures that will be used to verify the conformance of its instance. Specifically we analyze every collaboration and/or SD in which an interface, belonging to the tested component, is involved. We use the method invocations to derive test procedures following the UIT methodology as described in [1]. The final weights derived by Cowtest for each SD, as described in subsection 3.2.1, are now used for associating an importance factor to each method. This value will be used to distribute the test cases among the methods of the same component. In particular the importance factor is obtained following the steps below:

1. for each SD in which an interface appears, we distribute the weight of the SD on the invoked methods belonging to the considered interface;
2. for each method we sum all the values obtained in the previous step;
3. for each interface we normalize to 1 the sum of the values associated to its methods.

The definition of a test procedure is made using the CDT framework, that permits to codify test cases without references to any particular real implementation. In fact the invocations of the test cases refer to the interfaces of the virtual component, as defined in the specification workflow, that will be implemented using yet unknown components. Moreover, by assembling prefabricated components to form a virtual component, it is likely that the real implementation could supply more functionalities than those required. However, it is worth noting that the established set of test cases only stresses the functionalities defined in the UML specification.

In order to provide the choices useful for the definition of the test procedures, belonging to the same test case, we intend to analyze the contract associated to each method. To this purpose particular attention is dedicated to the preconditions which can specify parameter intervals or values useful for test cases generation.

4.2 Test of a group of integrated virtual components

The UML Component provide a description of the architecure incrementally obtained by the refinement of the Use Case diagrams. In particular, as illustrated in subsection 3.1, in each Use Case the complete list of the exception conditions (conditions that cause a different workflow from the standard established by the Use Case) are described. In order to apply the Cow_Suite methodology we use this information to derive a high-level SD, for each Use Case, in which the different paths are expressed using *if* conditions.
We can distiguish two different cases:

1. The Use Case does not have exceptions: applying the Cow_Suite methodology a set of test cases is derived from each SD or CD belonging to the sub-tree rooted in the considered Use Case. For each test case the invocation of the first method in its message-sequence implies the execution of the same, unique, path as specified in the SD. Every real implementation that does not reflect the behaviour subsumed by the test case, can be therefore refused. For deriving the test procedures we revise the UIT methodology, adapting it to a CB context. Specifically the choices cannot anymore refer to internal features, for instance states of the components, but only on those supplied by the interfaces. Similarly to the previous subsection we can analyze the contracts of the components implied in the SD, to derive parameters for the test procedures;
2. The Use Case contains exceptions: the Cow_Suite methodology, for each SD as described in subsection 3.2.2, supplies the complete list of the possible paths obtained combining the *if* conditions. In particular each of them corresponds to one exception expressed inside the Use Case diagram. The condition is the direct consequence of a return value of a method invocation included in the path. Therefore, differently from the previous case the invocation of a method could imply more than one

path. It is a tester task to establish the parameters for the test procedures in order to cover all the paths. To this purpose a controller of the test behaviour can be implemented to force a path, as outlined in [5].

In both cases it can be useful, for the tester team, to recover the test cases developed during the verification of the single components in order to obtain parameters for the test procedures generation. The diagrams used to derive test procedures can be fruitfully used, also as a guideline for the integration workflow. In this manner we obtain a functionally driven workflow, rather than a structural one (as for instance it would be using a class diagram).
In order to execute the test cases on the implementation of the virtual components we use the framework provided by CDT to execute the test cases. To do that, each test procedure is codified in a test written in Java and stored following the JUnit framework. Then when a suitable implementation of the subsystem can be realized we can use the CDT tools to execute the related test cases. Moreover by means of the WCT we can control if the pre- and post-conditions of each method invocation in the path are violated. To this purpose we have integrated the free tool iContract [19] in the framework. Therefore if a contract violation is detected, during the testing phase, we stop the test and immediately identify the wrong interaction between the components (similarly to what is done in [6] for OO development).

5 Conclusions

As evident, this is a paper presenting ongoing work: we have settled the ground for the development of an integrated testing framework for CB systems by selecting the necessary ingredients: the UML Components methodology is the adopted modelling notation; the Cow_Suite environment will be expanded so to allow for the derivation of the test cases from the UML specifications; and the CDT framework, in turn integrated with WCT,JUnit and iContract, will serve as the test driver. Several technical challenges underly this project and they are far from having been solved. We have identified some key issues: for instance, with regard to Cow_Suite, how can we construct meaningful test cases for components whose behavior and interaction are dynamically determined? With regard to CDT, how can we force the execution of the test sequences identified by Cow_Suite on a black-box component? For these and many related questions, discussion at TACoS will certainly provide very useful insights.

References

[1] Basanieri, F., Bertolino, A., *A Practical Approach to UML-based Derivation of Integration Tests*, Proceedings of QWE2000, Bruxelles, November 20-24, 2000.

[2] Basanieri, F., Bertolino, A., Marchetti, E., *The Cow_Suite Approach to Planning*

and Deriving Test Suites in UML Projects Proc. 5th International Conference on the Unified Modeling Language - the Language and its applications, Dresden, Germany, September 30 - October 4, 2002.

[3] Bertolino, A., Mirandola, R., *Modeling and Analysis of Non-functional Properties in Component-based Systems*, in these Proceedings.

[4] Bertolino, A., Polini, A., *A Framework for Component Deployment Testing*, to appear in the Proceedings of ICSE 2003, Portland, USA, May 3-10, 2003.

[5] Bertolino, A., Polini, A., *WCT: a Wrapper for Component Testing*, Proceedings of Fidji'2002, Luxembourg, November 28-29, 2002, to appear as LNCS.

[6] Briand, L.C., Labiche, Y., Sun, H., *Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code*, Proc. of ISSTA 2002, Roma, Italy, July 22-24, 2002, pp. 70-80.

[7] Cheesman, J., Daniels, J., "UML Components - a Simple Process for Specifying Component-Based Software", Addison-Wesley, 2000.

[8] Cockburn, A., "Writing Effective Use Cases", Addison-Wesley, 2001

[9] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., "Introduction to Algorithms", 2nd Ed., The MIT Press and McGraw-Hill, 2001.

[10] 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.

[11] Kruchten, P., "The Rational Unified Process - An Introduction", Addison-Wesley, 1999.

[12] Musa, J.D., Iannino, A., and Okumoto, K., "Software Reliability - Measurement, Prediction, Application", McGraw-Hill, New York, 1987.

[13] Offutt, J., Abdurazik, A., *Using UML Collaboration Diagrams for Static Checking and Test Generation*, Proceedings of UML 2000, University of York, UK, October 2-6, 2000.

[14] 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.

[15] UML Documentation version 1.3 Web Site. On-line at: http://www.rational.com/uml/resources/documentation/index.jsp

[16] Voas, J., *Developing a Usage-Based Software Certification Process*, IEEE Computer, August 2000, pp. 32-37.

[17] Wang, Y., King, G., Wickburg, H., *A Method for Built-in Tests in Component-based Software Maintenance*, in Proc. of the 3^{rd} ECSMR, Amsterdam, March 03-05, 1999, pp. 186-189.

[18] On-line at: http://www.junit.org

[19] iContract downloadable from: http://www.reliable-system.com