Built-In Contract Testing in Component Integration Testing

Hans-Gerhard Gross and Nikolas Mayer

Fraunhofer Institute for Experimental Software Engineering, Sauerwiesen 6, D-67661 Kaiserslautern, Germany

Abstract

Assembling new software systems from prefabricated components is an attractive alternative to traditional software engineering practices which promises to increase reuse and reduce development costs. However, these benefits will only occur if separately developed components can be made to work effectively together with reasonable effort. Lengthy and costly in-situ verification and acceptance testing directly undermines the benefits of independent component fabrication and late system integration. This position paper outlines and introduces an approach for reducing manual system verification effort by equipping components with the ability to check their execution environments at run-time. When deployed in new systems, built-in tester components check the contract-compliance of their server components, including the run-time system, and thus automatically verify their ability to fulfill their own obligations. This comprises functional/behavioural contracts as well as quality-of-service contracts between individual components. Enhancing traditional component-based development methods with built-in contract testing in this way reduces the costs associated with component assembly, and thus makes the "plug-and-play" vision of component-based development closer to practical reality.

1 Introduction

The vision of component-based development is to bring software engineering more in line with other engineering disciplines where assembling new products from standard parts is the norm. This model of software development presents some challenges, however. With traditional development approaches, the bulk of the integration work is performed in the development environment, giving engineers the opportunity to pre-check the compatibility of the various parts of the system, and to ensure that the overall deployed application is working correctly. In contrast, the late integration implied by component assembly means there is often little opportunity to verify the correct operation of applications before deployment time. Although component developers may adopt

rigorous test methodologies, with non-trivial software components it is impossible to be certain that there are no residual defects in the code. Formal proof or 100% test coverage are not viable options in most practical cases. Compilers and configuration tools can help to some extent by verifying the syntactic compatibility of interconnected components, but they cannot check that individual components are functioning correctly (i.e. that they are semantically correct), or that they have been assembled together into meaningful configurations (i.e. systems). As a result, components that may have behaved correctly in the sanitary condition of the development-time testing environment, may not behave so well when deployed in a system where they have to interact, or compete with other (third party) components for resources, for example memory, processor cycles and peripherals.

Realizing the ultimate vision of component-based development is therefore contingent on individual components having the built in ability to check their respective deployment environments. Only then will the true benefits of the "plug and play" vision that is promised by component-based development become a reality. The Built-In Contract testing technology described in this paper directly addresses this need by extending the component model to incorporate in-situ, run-time tests that can be performed without manual intervention. This technology addresses the issues involved in furnishing individual components with the capabilities needed to check their deployment environments at run-time. The enhanced model of component-based development, that incorporates this form of deployment tests, can be characterized by the phrase "plug, test and play".

A prerequisite for the correct functioning of a system containing many components is the correct interaction of individual pairs of components according to the client/server model. Following Meyer [10], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract. This views the relationship between a component and its clients as a formal agreement, expressing each partys rights and obligations. Contracts may be categorized in four levels according to Beugnard [2]. These are *Syntactic Contracts* which are typically taken care of by the component platform or some adaptors, *Behavioural Contracts* that are part of the built-in contract testing approach, *Synchronization Contracts* that are not considered here, and *Quality of Service Contracts* that are also part of the built-in contract testing approach, although here we only consider response time issues.

Section 2 introduces the built-in contract testing paradigm that is initially concentrating on behavioural contracts. This is followed by a section (section 3) on practial applications of the basic built-in contract testing model. Section 4 extends the basic model by introducing contract testing concepts that are aimed at response time verification (i.e. quality-of-service contract). Section 5 summarizes and concludes the paper.

2 Basic Model of Built-In Contract Testing

The objective of built-in contract testing is to check that the environment of a component meets its expectations. The philosophy is that an upfront investment in building test software directly into a component alongside the functional software results in reduced system assembly costs, and thus in an increased return on investment relative to how often the component is reused. Built-in contract checking greatly simplifies the effort involved in reusing a component, because a component can complain if it is mounted into an unsuitable environment. It therefore considerably strengthens the reuse paradigm of component-based software development. An unsuitable environment may be characterized as different functional behaviour from what is expected, or different latencies from what is expected by a component. This corresponds to violations of behavioural, or quality-of-service contracts as discussed in [2] and [4].

When an otherwise fault free component is deployed in a new environment, there are only two basic things that could go wrong during its execution: either the component itself is used incorrectly by others, in that case its providing contract differs from the one of its client, or one or more components that it uses and is depending upon malfunction, in that case its server contracts differ from what the component expects. Both of these scenarios can be characterized in terms of the client/server relationship: the former implies that one or more of a component's clients behave incorrectly, while the latter implies that one or more of a component's servers behave inappropriately. Checking that these errors do not arise, therefore, can be characterized as checking that the contract between components is adhered to. Hence the name contract testing.

While most contemporary component technologies enforce the syntactic conformance of a component to an interface (syntactic contract) through underlying mapping mechanisms, they do nothing to enforce the semantic conformance. Built in tests offer a feasible and practical approach for validating the semantics of components. In general, the server in a built-in contract testing configuration will contain -

- Built-in tests which are exclusively dedicated to checking a server's own deployment environment (its own servers). This comprises normal "explicit" servers as well as server components that are provided through the runtime environment, so-called "implicit" servers. The test cases are carefully designed to comply with trade-off requirements, and they are organized and contained in tester components.
- An introspection interface which provides access to and information about the supported testing interface.
- A built-in contract testing interface. This adds to the component's normal functional interface and serves contract testing purposes. The contract testing interface consists of public methods for state setup and state validation.

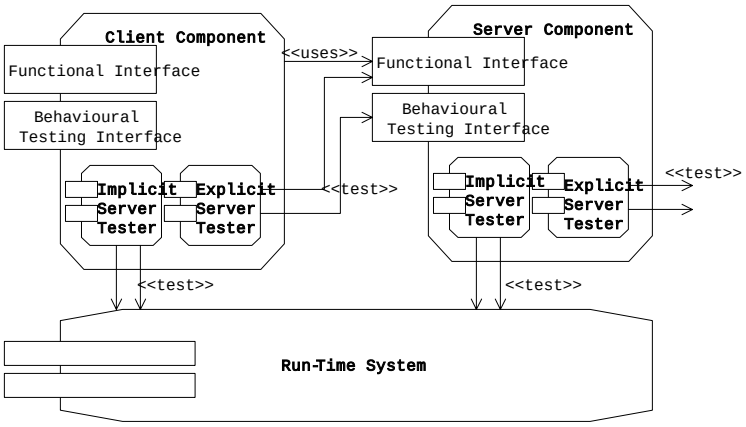

Fig. 1. Model of Components under the built-in contract testing paradigm (behavioural contract).

This enables access to the externally visible logical states of a component (defined in the behavioural model) for contract testing [5].

In general, the client will contain -

- An in-built contract tester for checking the server component. This tester is a separate component and includes the test cases for an associated other server component. The tester accesses the server's normal interface for executing the test cases, and the server's testing interface for state setup and state validation. The execution of the tester represents a full server test with which the client verifies whether the server provides its services correctly. The tests may correspond to functional testing criteria and represent an adequate test-suite for the individual unit. The size of the built-in tester is also subject to efficiency considerations.
- The client does not have to provide a contract testing interfaces in order to apply built-in testing technology to its own servers. However, the contract testing interface may be added so that it serves its clients with the same testing facilities as its server [5].

The basic model of built-in contract testing with built-in tester component and built-in behavioural contract testing interface is depicted in figure 1. The server's testing interface provides access operations that increases a component's observability and controllabilty, and the client's built-in tester component contains test cases according to the client's expectation toward that server that use the server's testing interface in order to verify its functional/behavioural interface. A detailed description of how built-in contract testing interfaces and built-in contract tester components are developed from models on the basis of a mainstream development method, the so-called KobrA Method [1] is provided in [7].

3 Practical Application of the Basic Contract Testing Model

Built-in contract testing has already been applied in several real developments, a number of small-scale case studies and some larger-scale development projects in the scope of the European Union funded IST project Component+ [5]. The following paragraphs briefly describe the process for developing contract testing artefacts. This process is heavily based on the KobrA development method [1] and model-based development principles including the creation of UML diagrams.

Development of the Testing Architecture

In theory, any arbitrary server-clientship relationship in component development and integration may be checked through built-in test software. These relationships are represented through any arbitrary association in a structural diagram, for example UML component, class and object diagram, as well as KobrA composition-, nesting- and creation-tree diagrams. In other words, every nesting association represents client-servership. For example, Figure 2 displays a KobrA containment hierarchy for a simple banking application that shows the individual components and their clientship relations. The decision on which associations in the diagram will be augmented with built-in contract tester components and interfaces depends on the estimation of how likely the individual components will be replaced. Initially, any <<acquires>> association is a possible candidate for built-in contract testing. The diagram displays a distributed system that comprises a local part (bank context) and a remote converter part (context).

The stereotype <<acquires>> represents dynamic associations that may be configured according to the needs of the application (i.e. components may be replaced). These are parts of the overall system that are likely to change over time, and the associations are therefore augmented with built-in contract testers, on the client side, and built-in contract testing interfaces, on the server side of the relationship. These elements represent simple additional software development efforts as specified in Figure 3. Here, the converter's testing interface extends the normal interface of the functional component, so that it becomes a testable component that provides additional access for the bank's built-in test software. This test software with all the individual test cases is contained in the bank's built-in converter tester component.

Development of the Contract Testing Artefacts

Entry criterion for the specification of a testing interface is a full functional specification for each operation of the tested component, for example following the operation specification template of the KobrA Method [1], or the behavioural model. Figure 4 displays the example behavioural model of a

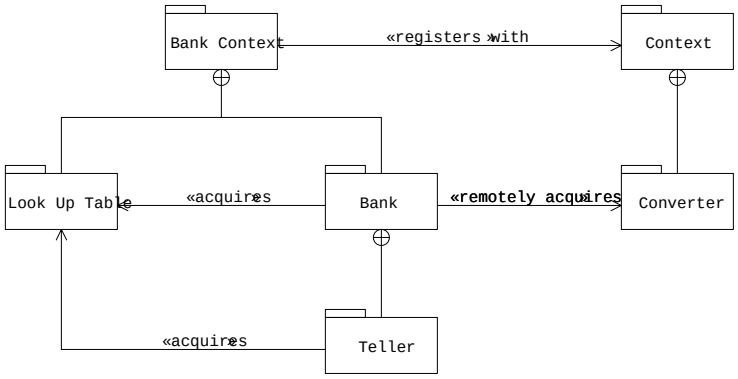

Fig. 2. KobrA-style containment hierarchy (architecture) [1] for a simple distributed banking application.

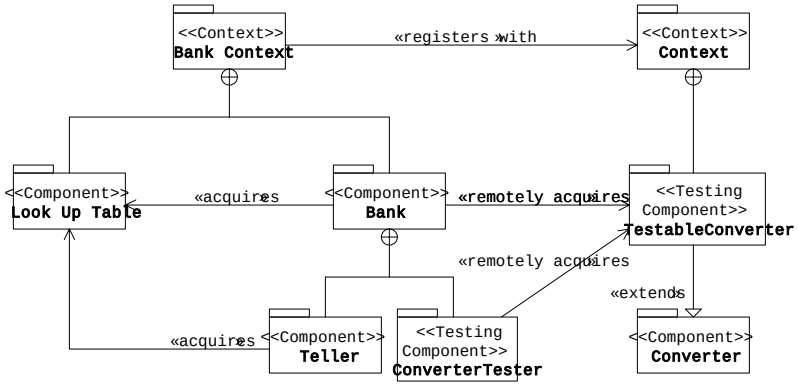

Fig. 3. Containment hierarchy with built-in contract testing.

banking card component. Such a specification comprises sufficient information for development of state setting and state checking operations that augment the functionality of the original server component and essentially represent the testing interface of the component. The structure of the additional testing interface for the banking card component is displayed in Figure 5. The

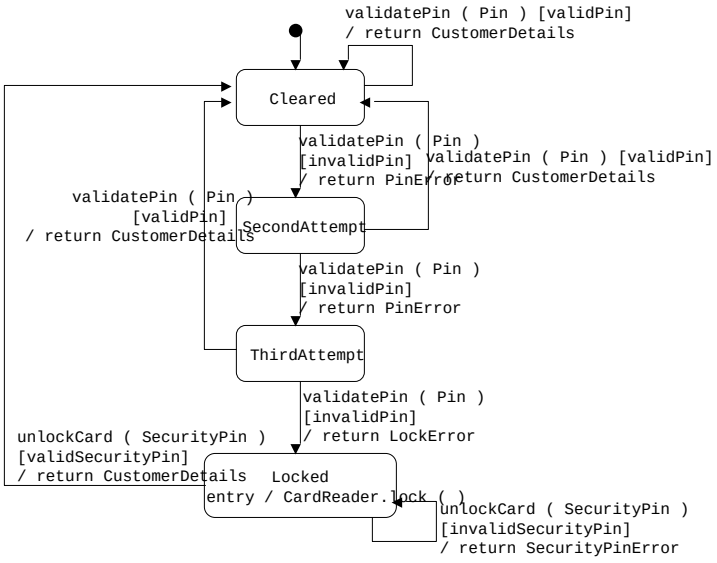

Fig. 4. Behavioural specification of a banking card component.

testing interfaces extends the normal functional interface of the component with state setting and state checking operations according to the component's behavioural model which are used exclusively for testing. Its implementation may be realized in two alternative ways, one that defines an individual state setting and checking operation per state, and one that only provides two parameterized operations plus a external definition of each state. The testing interface enables clients of the component to perform a full state-based test with setting pre- and checking post-conditions.

The tester that contains the test cases and performs the tests on behalf of the client is realized as a component in its own right that the client acquires in the same way as any other server component. It only happens to comprise code that runs a simulation of the transactions that the client typcially performs on the server. The test cases inside the tester component are derived according to typical functional test case generation techniques such as domain analysis and partition testing techniques, state-based testing, or method and message sequence-based testing. Models are also valuable sources for test case generation. For example Table 1 displays test cases on the basis of method sequences according to the behavioural model of the banking card component.

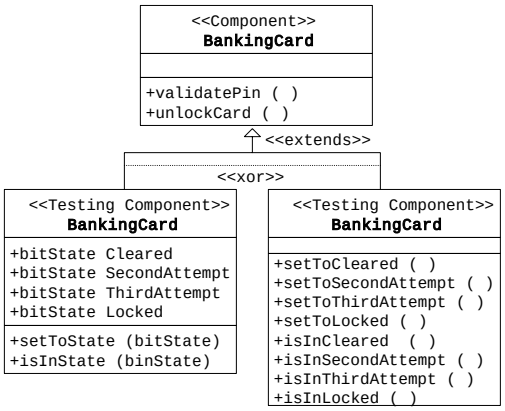

Fig. 5. Structural specification of a banking card component with additional testing interfaces (two alternatives).

No.	Initial State	PreCondition	Event	PostCondition	Final State
1	Cleared	[validPin]	validatePin(Pin)	return CustomerDetails	Cleared
2	Cleared	[invalidPin]	validatePin(Pin)	return PinError	
		[validPin]	validatePin(Pin)	return CustomerDetails	Cleared
3	Cleared	[invalidPin]	validatePin(Pin)	return PinError	
		[invalidPin]	validatePin(Pin)	return PinError	
		[validPin]	validatePin(Pin)	return CustomerDetails	Cleared
...	...	...	...	...	...

Table 1
Test case design based on method sequences according to the behavioural model of the banking card.

4 Extended Model of Built-In Contract Testing

The problem of component integration testing is compounded when non-functional requirements are considered, for example the compliance of the application to a real-time schedule (quality-of-service contract). Such real-time requirements are not only affected by individual objects, but by the entirety of all objects that make up the application. Each individual object or component may have a well-defined timing behaviour in a particular environment, for example the development-time environment. However, this is completely changed if it is plugged to other components that implement functionality of a real-time application at a customer's site. The timing behaviour of each possible combination of a component's feasible usage profiles in respect

to other components on a particular platform must therefore be verified when the compliance to the timing schedule of such a system is validated. Clearly, this cannot be done a priori for each component since the developer of that component can never anticipate its usage in a particular context. Timing verification can only be performed when components are assembled and put together into a new configuration.

The fact that object-oriented entities are inherently encapsulated, and they also represent state machines, creates a fundamental difficulty for the application of testing strategies in general. In order to generate worst-case timing behaviour for some operation of an object, the test software, i.e. an optimisation algorithm (e.g. random testing, or more advanced evolutionary testing [8]), must not only optimise and provide the input parameter values for the operation, but additionally, it must optimise and provide appropriate initial states from which the event will be triggered. The execution time of an operation is defined through the values of the internal state variables plus the input values of the operation. The search algorithm must set the values for the internal state variables from outside the object's encapsulation boundary. This would require all internal state variables to be made publicly available to external clients of the object.

Search based execution-time analysis that is applied to object-oriented, component-based real-time systems is therefore relying on the optimisation of input parameters according to the method invocation history plus a specific architecture. The architecture extends the built-in contract testing technology. This extended model is depicted in figure 6. The tested server component provides another testing interface (quality-of-service testing interface) that comprises timing notification and measurement facilities. This interface is used by the client's timing tester component. This component contains usage information of the server and respective timing requirements for individual transactions. This information comprises sequences of operation calls and their respective input parameter signatures. These are required in order to define the input parameter sequences that are constantly generated through the optimization process during a timing test.

The basic idea for applying this technology is laid out in [8]. Current work in this area is devising a method and a process for develompment and automatic application that will be integrated in our own object-oriented and component-based development method, the KobrA Method [1].

5 Summary and Conclusions

The philosophy behind built-in contract testing is that an upfront investment on verification infrastructure pays off during reuse. This adds considerable value to the reuse paradigm of component-based software development because a component can complain if it is given something unsuitable to interact with, and if it is mounted into an unsuitable environment. This comprises the

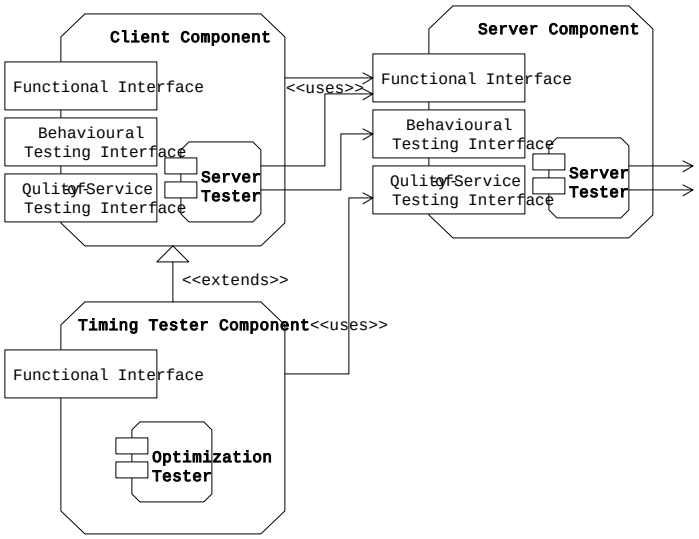

Fig. 6. Extended Model of Components under the built-in contract testing paradigm that comprises optimization-based timing verification (quality-of-service contract).

behavioural contract as well as the quality-of-service contract. The benefit of built-in verification follows the same principles which is common for all reuse methodologies: the additional effort of building the test software directly into the functional software results in an increased return on investment according to how often such a component will be reused.

The idea of building tests into components is not new. The very basic principles of built-in contract testing are based upon the ideas of traditional assertion checking mechanisms. These are operations in the code that are executed in regular intervals and compare current execution conditions of a component with expected execution conditions [3] and raise an exception if both deviate. Wang et. al [12] take these ideas a bit further and adopt a hardware anology in which components have self-test funcationality that can be invoked at run-time to ensure that they have not degraded. However, since software by definition cannot degrade, the portion of a self-test which rechecks already verified code is redundant, and simply consumes time and space. The approach described in this paper concentrates on things that are likeley to change in a new component configuration, that is a component's environment, and more specifically, its associated server components and its run-time platform on which it depends. Jézéquel et al. [9] also advocate the building of test software into components, but their goal is to optimize the development time unit testing of components rather than to support in-situ integration testing at integration and deployment time.

Acknowledgements

This work is partly supported through the European IST Framework Programme EC-IST-1999-20162 Compoent+ project, EC ITEA Framwork Programme under the Empress project acronym, and the German Federal Department of Education and Research under the MDTS project acronym.

References

[1] Atkinson, C., et al., "Component-Based Product-Line Engineering with UML". Addison-Wesley, London, 2001.

[2] Beugnard, A, et al., *Making Components Contract Aware*. Computer, 32(7), July, 1999.

[3] Binder, R., "Testing Object-Oriented Systems - Models, Pattern & Tools". Addison-Wesley, 2000.

[4] Collet, P., *On Contract Monitoring for the Verification of Component-Based Systems*. In 1st OOPSLA WS on Specification and Verification of Component-Based Systems, Tampa Bay, Florida, October 14, 2001.

[5] Component+ Project Technical Report, *Built-in Testing for Component-based Development*. http://www.component-plus.org, 2001.

[6] Component+ Project Technical Report, *D4 - BIT Case Studies*. http://www.component-plus.org, 2002.

[7] Gross, H.-G., Component+ Methodology, *Built-in Contract Testing Method and Process*. IESE Report 037.02/E, Kaiserslautern, 30. Oct. 2002 (`http://www.iese.fhg.de/Publications/Iese_reports/`).

[8] Gross, H.-G., *Search-Based Execution-Time Verification in Object-Oriented and Component-Based Real-Time System Development*. 8th IEEE WS on Object-oriented Real-time Dependable Systems, Guadalajara, Mexico, January 15-17, 2003.

[9] Jézéquel, J.M., Deveaux, D., LeTraon, Y., *Reliable Objects: Lightweight Testing for OO Languages*. IEEE Software, 2001.

[10] Meyer, B., "Object-oriented Software Construction", Prentice Hall, 1997.

[11] Szyperski, C., "Component Software: Beyond Object-Oriented Pro-gramming". Addison-Wesley, 1999.

[12] Wang, Y., King, G., Fayad, M., Patel, D., Court, I., Staples, G., and Ross, M., *On Built-in Tests Reuse in Object-Oriented Framework Design*. ACM Journal on Computing Surveys, Vol. 23, No. 1, March 2000.