Built-in Contract Testing in Component-based Application Engineering

Hans-Gerhard Groß

Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, 67661 Kaiserslautern, Germany
grossh@iese.fhg.de

Abstract. Assembling new software systems from prefabricated components is an attractive alternative to traditional software engineering practices and 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 paper describes 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 testing 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. 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 allow software vendors to avoid the overheads of traditional development methods by assembling new applications from high-quality, prefabricated, reusable parts. Since large parts of an application may therefore be constructed from prefabricated components, it is expected that the overall time and costs involved in application development will be reduced, and the quality of the resulting applications will be improved. This expectation is based on the implicit assumption that the effort involved in integrating components during configuration and deployment is lower than the effort involved in developing and validating applications through traditional techniques. However, this does not take into account the fact that when an otherwise fault-free component is integrated into a system of other components, it may fail to function as expected. Current component technologies can help to verify the syntactic compatibility of interconnected components, but they do little to ensure the semantic compatibility of inter-connected components, so that the individual parts are assembled into meaningful configurations. Software

developers may therefore be forced to perform more integration and acceptance testing in order to attain the same level of confidence in the system's reliability.

The testing approach described in this paper is based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" dynamically at deployment time will fulfil their contract. Although built-in contract testing is primarily intended for validation activities at deployment and configuration-time, the approach also has important implications on the development phases of the overall software life-cycle. Consideration of built-in test artifacts needs to begin early in the design phase as soon as the overall architecture of a system is developed and/or the interfaces of components are specified. Built-in contract testing therefore needs to be integrated with an overall software development methodology. This paper focuses on explaining the basic principles behind built-in contract testing, and how they affect the testing of component applications. Additionally, it demonstrates how built-in contract testing can be integrated with, and made to complement, a mainstream development method that is based on the creation of UML models.

2 Software Components and Component-based Development

This work concentrates on component-based development, so it is important to define the term component for the purpose under consideration: A software component is a unit of composition with explicitly specified provided, required and configuration interfaces, plus quality attributes. This definition is based on the well known definition from ECOOP'96 [1], that defines a component as unit of composition with contractually specified interfaces and context dependencies only, that it can be deployed independently, and that it is subject to composition by third parties. The definition for this work has intentionally a broader scope that is avoiding the terminology *independently deployable*, since this work is not specifically restricted to technologies such as CORBA, .NET or COM. In this respect our definition is closer to Booch's definition who sees a component as a logically, cohesive, loosely coupled module that denotes a single abstraction [2]. From this it becomes apparent that components are basically built upon the same fundamental principles as object technology. The principles of encapsulation, modularity, and unique identities are all basic object-oriented principles that are subsumed by the component paradigm [3]. A Java class may therefore be considered as a component in this respect.

2.1 Development Method

The initial starting point for a software development project is undoubtedly a system or application specification derived and decomposed from the system requirements. Requirements are collected from the customer of the software. They are decomposed in order to remove their genericity in the same way as system

designs are decomposed in order to obtain finer grained parts that are individually controllable. These parts are implemented and later composed into the final product. The decomposition activity is aiming to obtain meaningful, individually coherent parts of the system, the components. It is also referred to as component engineering or component development. The composition activity tries to assemble existing parts that may have been already used in other applications, into a meaningful configuration that reflects the predetermined system requirements. This activity is termed application engineering or application development.

In its purest form, component-based development is only concerned with the second item, representing a bottom-up approach to development. This requires that every single part of the overall application is already available in a component repository in a form that exactly maps to the requirements of that application. Typically, this is not the case, and merely assembling readily available parts into a configuration will quite likely lead to a system that is not conforming to its original requirements. Component-based development is therefore usually a mixture of top-down decomposition and bottom-up composition. In other words, the system is decomposed into finer grained parts, that is subsystems or components, and these are attempted to be mapped to individual prefabricated components. The whole process is iterative and must be followed until all requirements are mapped to corresponding components or until the system is fully decomposed onto the lowest desirable level of abstraction. If suitable third party components are found, they can be composed to make up the system or subsystem under consideration. The outcome of such a development process is usually a heterogeneous assembly made of prefabricated parts plus own implementations. The decomposition process is based on the derivation of component specifications and realizations. The specification of a component comprises every piece of information that is necessary to fully describe what a system part does, and the realization of a component contains full information that is necessary to implement this part. The description of the development artifacts in this paper follows the KobrA development method [3]. This method uses the UML as primary notation. That means most software documents that are created over the course of a development project applying this method are UML models. However, there are other artifacts but KobrA realizes the concepts of the OMG's Model Driven Architectures, so models are the primary development documents.

2.2 Component Specification

A specification is a collection of descriptive documents that collectively define what a component can do. Typically, each individual document represents a distinct view on the subject, and thus only concentrates on a particular aspect of what it can do. A specification contains everything that is necessary in order to fully use the component and understand its behavior. As such, the specification can be seen as defining the provided interface of the component. Therefore, the specification of a component comprises everything that is externally knowable

of its structure (e.g. associated other components, in form of a structural specification, or UML structural diagrams), function (e.g. provided operations, in form of a functional specification), and behavior (e.g. pre- and post-conditions, in form of a behavioral specification, UML behavioral diagrams). These artifacts represent a complete framework for a component specification.

2.3 Component Realization

A realization is a collection of descriptive documents that collectively define how a component is realized. A realization should contain everything that is necessary in order to implement the specification of a component. A higher-level component is typically realized through a collection of lower-level components that are contained within and act as servers to the higher-level component. Additionally, the realization describes the items that are inherent to the implementation of the higher-level component. This is the part of the functionality that will be local to the subject component and not implemented through sub-components. In other words, the realization defines the specification of the sub-components, this is the expected or required interface of the component, plus its own implementation.

3 Built-in Contract Testing

Meyer [4] defines the relationship between a component and its clients as a formal agreement or a contract, expressing each party's rights and obligations in the relationship. This means that individual components define their side of the contract as either offering a service (this is the server in a client-server relationship) or requiring a service (this is the client in a client-server relationship). When ”individually correct” components are assembled and deployed to form a new system or to reconfigure an existing system there are only two things that can go wrong:

1. Explicitly acquired servers or implicitly acquired servers within a component's deployment environment may behave differently to those in its original development environment. Since such servers are either acquired explicitly from external sources, or implicitly provided by the run-time system, they may not abide by their contracts (semantically), even if they conform syntactically to their expected interfaces.
2. Clients of the component may expect a semantically different service to that provided, although they may be ”happy” with the syntactic form of the client-ship.

There are consequently two things, in terms of the generic deployment scenario depicted in Fig. 1, that should ideally be tested to ensure that a component will behave correctly within its deployed environment: The deployed component (in the center of Fig. 1) must verify that it receives the required support from its servers. This includes explicitly acquired servers and implicitly presumed servers

(i.e. the run-time system). Clients of the deployed component must verify that the deployed component implements the services correctly that it is contracted to provide. In order to check these properties dynamically when a system is configured or deployed, test software can be built into the clients alongside their normal functionality. The test software may be a collection of tests organized in a component, and testing is equivalent with executing the built-in test software. The two test requirements identified above indicate that there are basically two places where the additional test software for a given component should be located:

1. In the component itself, to verify that its environment (its servers) behaves according to what the component has been developed to expect, and
2. In the clients of the component to verify that the component implements the semantics of what its clients have been developed to expect.

These two scenarios are illustrated in Fig. 1. The acquires and presumes associations indicate the relations which must be tested to gain full confidence that the integrated component will inter-operate correctly with its environment. While most contemporary component technologies enforce the syntactic conformance of a component to an interface, they do nothing to enforce the semantic conformance. This means that a component claiming to be a stack, for example, will be accepted as a stack so long as it exports methods with the required signatures (e.g. push and pop). However, there is no guarantee that the methods do what they claim to do. Built in tests offer a feasible and practical approach for validating the semantics of component interactions.

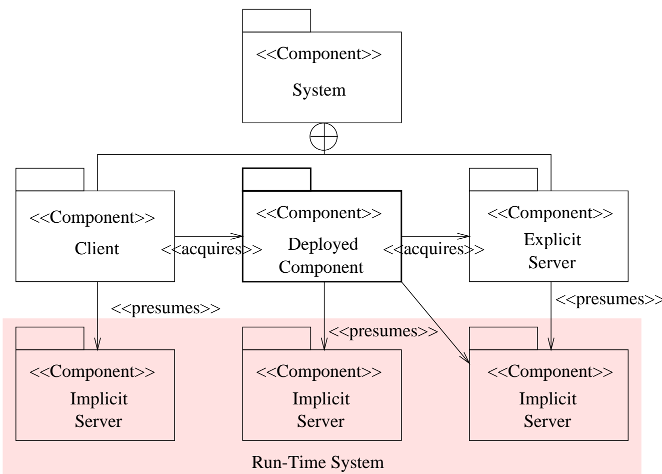

Fig. 1. Deployment of a component into a system illustrated through a KobrA containment tree.

The objective of built-in contract testing is to check that the environment of a component does not deviate from that which it was developed to expect. The philosophy behind built-in contract testing is that an upfront investment in 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 mounted into an unsuitable environment. The benefit of built-in verification follows the principles which are common for all reuse methodologies: the additional effort of building the test software directly into the component alongside the functional software results in an increased return on investment depending on how often such a component is reused. This in turn is determined by how easily the component may be reused. Built-in contract checking greatly simplifies the effort involved in reusing a component.

3.1 Tester Components

Rather than associate a test with a single method, it is more in the spirit of component technology to encapsulate it as a full tester component in its own right. The tester component (the server tester component in the client in Fig. 2) contains tests that check the semantic compliance of the server that the client acquires. The tests inside the client's tester component represent the behavior that the client expects from its acquired server. We call a client that has test software a built-in testing component since it is able to test its associated servers. Tester components that are embedded in testing components provide the optimal level of flexibility with respect to test weight (at both the run-time and development time).

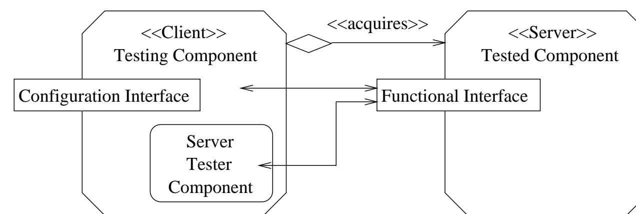

Fig. 2. Testing Component with an associated server tester component.

In general, the testing component (client) will contain one or more built-in contract testers for checking its servers. These testers are separate components that include the test cases for implicitly or explicitly acquired components. Obviously, if the run-time system is invariant, a component will not include any server testers for the underlying platform. A client that acquires a server will typically exhibit a configuration interface through which a particular server component may be set (as in Fig. 2). This can be a method such as *setServer (Component Server)* that assigns a particular server to a component reference in the client. The *setServer* method is typically invoked when the system is configured and

deployed. However, before the assignment is finally established, the client will execute its built-in tester components to perform a full semantic check on this new server. The component passes the reference to its tester, and this executes a simulation on the services of the newly acquired component. It may raise an exception if the test fails, otherwise the client establishes the connection. The tests may be derived through any arbitrary test case generation technique such as requirements-based test generation, equivalence partitioning, or boundary value analysis [5], [6]. According to the applied testing criteria, it may represent an adequate test-suite for the individual unit. In this case, the executed built-in tests represent the client's usage profile that a supplier of the acquired server may never anticipate. The supplier may only perform a traditional unit test according to their own anticipated usage profiles. This is the fundamental difference to more traditional testing that is not built-in. The size of the built-in tester is also subject to efficiency considerations.

3.2 Testable Components

Component-based development is founded on the abstract data type paradigm with the combination and encapsulation of data and functionality. State transition testing is therefore an essential part of functional testing. This means that in order to check whether an operation of a component is working correctly, the component must be brought into an initial state before the test case is executed, and the final state of the component as well as the returned values of the operation must be checked after the test case has been applied (pre- and post-conditions). The basic principles of encapsulation and information hiding dictate that external clients of a component should not see implementation and state variables. The test software outside the encapsulation boundary cannot therefore set or get internal state information. Only a distinct history of operation invocations on the functional interface results in a distinct initial state required for executing a test case. Since the tests are performed in order to verify the correctness of the functional interface it is unwise to use this interface for supporting the testing (i.e. setting and getting state information).

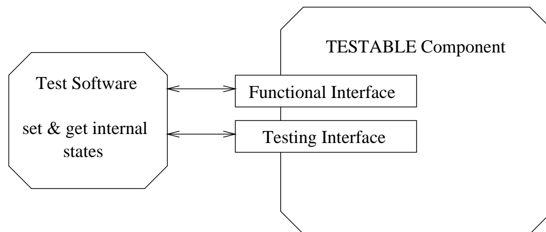

Fig. 3. Testable component with an additional contract testing interface.

A testable component under the built-in contract testing paradigm is a component that can be tested, which means it provides some in-built support for the execution of a test by an external client. The required support is provided in the form of a contract testing interface as displayed in Fig. 3. The basic idea in built-in contract testing is to enhance the normal functional interface of a component with a testing interface that exposes the logical (externally visible) states of the component, and makes them available for setting and checking. These states are part of the component's specification (this is the KobrA specification behavioral model). For example a gear box exhibits externally visible states, five forward settings one reverse and one neutral, that are essential for operating the gear box. A testable component can also contain an introspection interface that provides access to and information about the supported testing interfaces. This may be implemented in form of a Java-like Interface that realizes a testability contract. This is part of the overall methodology and detailed in [7].

A testable component contains a built-in contract testing interface which extends the component's normal functional interface with operations to support contract testing. The most basic contract testing interface consists of state assertions realized as operations that are invoked by a tester to see whether the component is residing in a defined logical state. This is the most fundamental, and well known state checking mechanism that is also outlined in [5], or [6]. This technique is based on additional operations on abstract data types that implement the assertion checking code. For the gear box example, this would map to additional Boolean operations that check whether the box is in one of its logical states that represent the gears; for example *isInGearX ( )*, with X representing each individual gear setting. Functional testing in this case would comprise a test for each possible individual gear transition according to the KobrA specification behavioral model. This corresponds to state transition coverage and consequently to coverage of the behavioral specification.

The disadvantage of having only a state checking mechanism is that for some tests quite a long history of operation invocations may be necessary in order to satisfy the preconditions. This can be circumvented through additional state set up operations that can directly manipulate the state variables according to the logical states, for example *setToGearX ( )*, with X representing each externally defined gear setting. This may be simple and straightforward for many components such as the gear box control system, but for some components substantial re-implementation of the component's functionality may be required. However, specific libraries may alleviate this effort as demonstrated in [7].

For many components, particularly ones that built up complex internal data structures, the effort of designing and implementing state setting operations will be just as intense as for the initial operations whose behavior they will simulate. This is a complexity issue, and it clearly makes no sense to re-implement exactly the same functionality and then use it for state setting purposes. The decisions about the state setting interface must definitely be based upon tradeoff considerations along the lines which strategy is easier to implement: implemen-

tation of the state setting operation plus less complex test case implementations, or implementation of only state checking operations with the cost of having to implement larger or more complex client tests. This clearly underpins the importance of the state checking operations that should always be provided as a minimal requirement by a component that implements built-in contract testing technology. These decisions must be wearily measured, and they are subject of further investigations. The Java standard library provides many good examples of how much introspection individual classes may provide and how simply these operations may be designed [9].

3.3 Quality assurance of the additional built-in test artifacts

State checking and setting operations of the testing interface enable access to the internal state variables of a component for contract testing by breaking the encapsulation boundary in a well-defined manner, while leaving the actual tests outside the encapsulation boundary. The clients of the component use the contract testing interface in order to verify whether the component abides by its contract. In this way, each individual client can apply its own test suite according to the client's intended use, or usage profile of its server. However, an important issue especially when in-built testing is considered, is that the test software may also be faulty.

Since the additional testing interface is an integral part of the component and thus of the overall development effort, in fact it may be seen as implementing normal functionality, the provider of that component should apply the same thorough quality assurance criteria as used for the rest of the component. The KobrA method recommends the creation of a Quality Assurance Plan that defines the quality criteria for the entire project and also individual components [3]. KobrA's primary quality assurance technique for individual units is inspections of the underlying models and specification artifacts, since this type of defect detection may be performed as early as possible in the development life cycle. Additionally it promotes the application of unit tests that represent the vendor's view on the functionality of the component. This in fact is not testing the testing environment, but rather it may be seen as checking the normal functionality of a component.

The tester components that are built into clients are typically also subject to a thorough inspection process that is based upon and guided through the clients' individual specifications. That means each test must be checked for correct pre- and post-conditions and input parameters according to the models and criteria from which a test is derived. This is performed through perspective-based inspections of the requirements and specification documents as detailed in [8]. A test of the tester component in the conventional way clearly makes no sense.

4 Modeling and Design of Testing Artifacts

The advantage of a model driven architecture is that a system or component may be described in an abstract form without having to decide how the component

will be implemented in a particular language. The models basically represent a complete specification of what a component will be implementing (e.g. KobrA Specification), and how (logically) it will be realized (e.g. KobrA Realization). The models also determine the items which must be checked when the product is finally realized. A test model can therefore be derived directly from the functional model of the component's architecture (model-based testing). A component and the testing of that component can consequently be developed in parallel by using exactly the same principles and processes.

4.1 Design and Development of a Testing Interface

The testing interface is responsible for setting preconditions and checking the post conditions for a test. Pre- and post-conditions for an event are defined in the functional model and in the behavioral model. Setting initial states and checking final states can be performed through auxiliary operations in a component's testing interface, i.e.

```
void setToState (_BIT_State definedState);
bool isInState (_BIT_State definedState);
```

These two testing interface operations represent an alternative to the previously introduced state checking and setting operations. Their advantage is that all contract testing interfaces will have the same signatures, only the logical states and their realizations are variable [7]. The testing operations *setToState()* and *isInState()* bring the component into a well defined state, and check whether the component is residing in a well defined state according to the specification. The testing interface is constant for any arbitrary component as part of a testability contract whereas the state model and hence the states are different according to the functionality of the component. Each state represents a number of invariants that the component must abide to at a given point in time.

4.2 Design and Development of Tests and Tester Components

Since built in tests, by definition, are embedded within a component at run-time, their size and speed is an important concern. If a test is too small it may not provide the required degree of confidence in the correctness of the tested server component. On the other hand, if a test is too large, it may unduly increase the size or decrease the execution speed of the test at run-time. The optimal test represents a balance between these two requirements, and it is highly dependent on the context in which the test is applied. The magnitude of a test is characterized by its weight. At one logical extreme, the heaviest form of test is one that performs a complete functional test covering all possible I/O and state options. At the other logical extreme, the lightest possible form of test is a test that does nothing. In between these two extremes, a full spectrum of test weights is conceivable.

The context sensitivity of the optimal weight of a test creates a dilemma for the reusability of components with built in tests. To make BIT components reusable, therefore, it is necessary to use some form of mechanism available for handling variability. Most variation techniques are applied at development time (e.g. inheritance, extension points, templates and generation). At run-time, selection among the alternatives built into the executable component is achieved by the provision of a configuration interface. Thus, reusable built-in test components will incorporate one or more configuration methods to select the weight of the built in tests at run-time. Such a configuration interface can be used to attach and detach different contract tester components dynamically depending on speed and size considerations of the deployed system (Fig. 4).

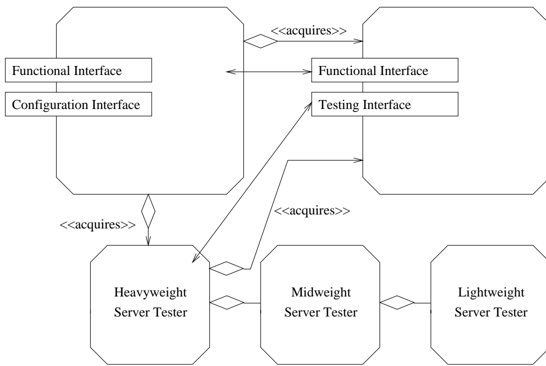

Fig. 4. Configuration for adjusting the test weight in testing components.

5 Development of Testable and Testing Components

The basis of a functional contract test suite that will be used for deployment testing is the state transition table derived from the KobrA specification state model. This defines a minimal set of tests for functional testing. Each line in the transition table represents a test for each state transition from the externally visible state model. This test set represents full transition coverage of the model. Table 1 shows the state transition table for the previously used gear box control interface. The table contains every possible externally visible state transition of the considered component. Its abstract state model is defined through the states *Neutral*, *Gear1* to *Gear5* and *Reverse* representing each feasible gear box setting respectively.

The state transition table contains sufficient information for deriving the testing interface for a testable gear box control component. Fig. 5 displays a UML-like representation (in form of a structural model [10]) of a testable gear box realized in the two previously mentioned alternative ways. In this case, the

Table 1. State transition table of a gear box control unit.

No	Initial State	Pre-Condition	Event	Final State
1	Neutral	momentum < reverseMomentum	toReverse ()	Reverse
2	Reverse		toNeutral ()	Neutral
3	Neutral	momentum < gear1Momentum	toGear1 ()	Gear1
4	Gear1		toNeutral ()	Neutral
5	Neutral	momentum < gear2Momentum	toGear2 ()	Gear2
6	Gear2		toNeutral ()	Neutral
7	Neutral	momentum < gear3Momentum	toGear3 ()	Gear3
8	Gear3		toNeutral	Neutral
9	Neutral	momentum < gear4Momentum	toGear4 ()	Gear4
10	Gear4		toNeural ()	Neutral
11	Neutral		toGear5 ()	Gear5
12	Gear5		toNeutral ()	Neutral

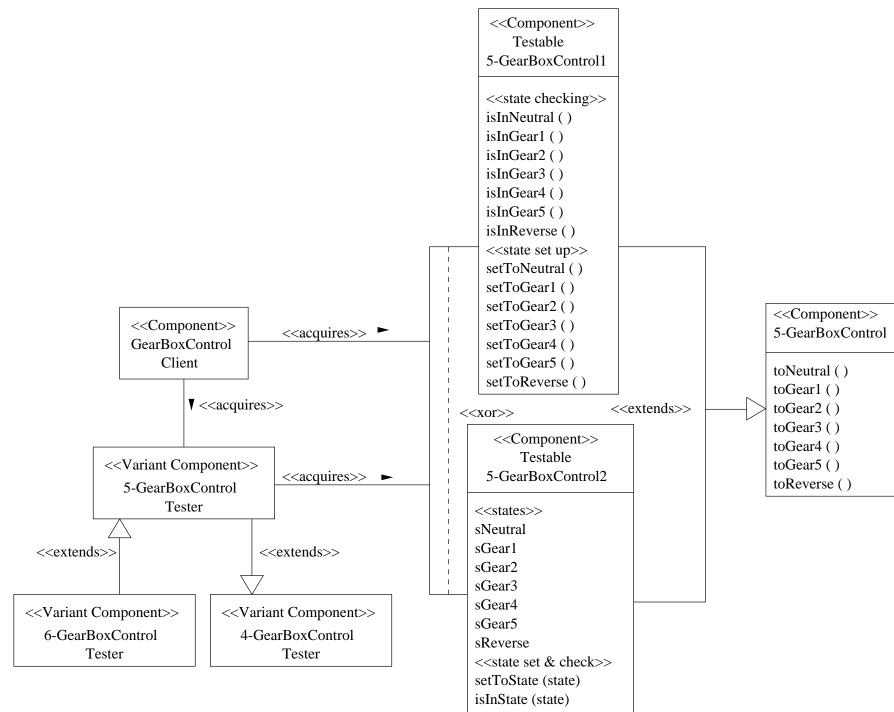

Fig. 5. Contract testing artifacts for a gear box control unit. TestableGearBoxControl1/2 represent alternative realizations of the contract testing state set up and checking mechanisms.

testing interface extends the normal functional component with state testing infrastructure. Each defined state (Gear1 to 5, Neutral, and Reverse) is mapped to functionality that can bring the component into this state or answer whether the component is residing in a particular state. A contract tester component for specification-based testing (*GearBoxControlTester*) can be derived from the state table by using the corresponding interface operations and the test cases (Event in Table 1). The contract tester component can be seen as a particular test driver for a particular test criterion. In this case the test criterion is transition coverage and consequently coverage of the behavioral specification. This corresponds to the functional tests that a client of the gear box control component may perform when an application is assembled and put together. Another tester component may contain additional test cases for other testing criteria (e.g. equivalence partitioning). This may be generated in the same way as for the functional model.

Contract testing is particularly aimed at deployment time testing in application engineering. If during reconfiguration of the application, a new implementation of the *testableGearBoxControl* unit replaces the current one, its client's *setServer* operation invokes the built-in *GearBoxControlTester* automatically and verifies the semantic compliance of the new server to the existing client-ship contract. This mechanism realizes automatic integration testing. The fact that the testers are organized in components facilitates their reuse for different client-ship contracts. If a 6-gear system is required, the original tester can simply be extended or replaced to accommodate the additional tests. If only a 4-gear system is developed, the number of tests in the original tester can simply be reduced through the extension mechanisms displayed in Fig. 5).

6 Conclusions

This paper has described an approach for enriching components with the capability to verify their run-time integrity, in-situ, by means of built-in tests. The idea of building test into components is not new. However, previous approaches such as [11] have adopted a hardware analogy in which components have self test functionality that can be invoked at run-time to ensure that they have not degraded. 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 resources. The approach described in this paper augments the earlier work by focusing built-in test software on the aspects of a component's capabilities which are sensitive to change at run-time, and this the environment that represents the services used by the component. This new emphasis is characterized as *built-in contract testing*.

Built-in contract testing provides an architecture and a methodology that is particularly well suited for highly dynamic and distributed systems, such as Internet applications, and systems with dynamic reconfiguration. Since these are the applications primarily targeted by modern component technologies, built-in contract testing represents a natural extension to component-based software

engineering practices. The work is currently oriented towards reconfigurable information systems for which the overhead of run-time tests does not particularly affect efficiency considerations. Extending the technology to real-time and embedded systems still presents some challenges for future research in terms of how much in-built testing such systems may bear.

We believe this methodology represents a contribution towards the practical applicability of component technology and component-based development practice. We have consequently integrated built-in contract testing with a general-purpose model-driven approach to component-based development, known as the KobrA method [3] that promotes the early design of tests along with functional artifacts. By extending the component paradigm with effective in-situ verification techniques and processes, built-in test technology brings the vision of component-based development one step closer to realization.

References

1. Szyperski, C., "Component Software: Beyond Object-Oriented Programming". Addison-Wesley, 1999.
2. Booch, G. "Software Components with Ada. Structures, Tools and Subsystems", 1987.
3. Atkinson, C., et al. "Component-based Product Line Engineering with UML", Addison-Wesley, 2001.
4. Meyer, B., "Object-oriented software construction", Prentice Hall, 1997.
5. Pressman, S., "Software Engineering: A practioner's approach", McGraw-Hill, 1997.
6. Somerville, I., "Software Engineering", Addison-Wesley, 1995.
7. Component+ Project Report D.3, http://www.component-plus.org, September 2001.
8. Laitenberger, O., "Cost-effective Detection of Software Defects through Perspective-based Inspections", Fraunhofer IRB Verlag, 2000.
9. Java Standard Library API Specification at http://java.sun.com/apis.html.
10. OMG Unified Modeling Language Specification, Object Management Group, 2000.
11. 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 Com-puting Surveys, Vol. 23, No. 1, March 2000.