

Built-in Contract Testing in Model-driven, Component-Based Development

Colin Atkinson¹ and Hans-Gerhard Groß²

¹ University of Kaiserslautern, AG Component Engineering
Kaiserslautern, Germany
atkinson@informatik.uni-kl.de

² Fraunhofer Institute for Experimental Software Engineering
67661 Kaiserslautern, Germany
grossh@iese.fhg.de

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 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 test (BIT) 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 are therefore created by reuse, 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 at deployment time is lower than the effort involved in developing and validating applications through traditional techniques. However, current component technologies do little to ensure that this is indeed the case, and that an application assembled from independently developed components will function correctly in the deployment environment. 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.

When using traditional development approaches, the bulk of the unit integration work can be performed in the development environment, affording engineers the opportunity to 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. For non-trivial software components it is impossible to be certain that there are no residual defects in the code since formal proof or 100% test coverage are not viable options in most practical cases, and even if a component itself is fault free it may fail in a deployed environment when it has to rely on the services of other components or has to compete with other (third party) components for resources such as memory, processor cycles and peripherals. Compilers and configuration tools can help to verify 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). In short, although traditional development time V&V techniques can help assure the quality of individual components, they can do nothing to assure the quality of systems assembled from them.

This paper describes an approach, developed within the EU project Component+, which addresses this problem by building tests into components so that they have their own "built-in" capability to check their environment at deployment time. The approach therefore distributes the verification effort, traditionally applied exclusively at the system level, amongst the components making up a system. In effect, it serves to make components more robust, or trustworthy, by giving them the capability to raise a flag if they are deployed in an unsuitable environment. System integrators can thus gain a given level of confidence in the quality of an application with much less validation effort at the system level.

The correct functioning of a system of components at run time is contingent on the correct interaction of individual pairs of components according to the client/server model. Component-based development can be viewed as an extension of the object paradigm in which, following Meyer [3], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract which views the relationship between a component and its clients as a formal agreement, expressing each party's rights and obligations. Testing the correct functioning of individual client/server interactions against the specified contract therefore goes along way towards verifying that a system of components as a whole will behave correctly. The described approach is therefore based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" at deployment time will fulfil their contract.

Although built-in tests actually execute at run-time once a component has been deployed, their impact is much wider. Consideration of built-in tests 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 within the overall software development methodology. In this paper we show how built in contract testing can be integrated with, and made to complement, a mainstream model-driven approach to component based development. The underlying 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. 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 is reused.

In the following section, we first introduce the basic principles of component based and model driven development upon which the approach is based. Section three then describes the core concepts of built-in contract testing and the various forms of components which it involves. section 4 describes how contract testing can be integrated naturally with the features of a mainstream model-driven approach to component-based development known as Kobra. Finally, section 5 illustrates how specific built-in test components can be developed using this methodology.

2 Component-based and Model-driven Development

Built in contract testing is intended to provide a seamless extension to the regular component-based and model-driven development paradigms. To avoid dependencies on the idiosyncrasies of a particular physical component technology, and to allow the approach to be used in conjunction with any model-driven architecture in the early phases of development, our approach adapts a general model-driven view of components as explained below. The component model is based on that of the Kobra development method [1], which in turn is adapted from Szyperski [7]:

"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

This captures the key ideas that a component is a self-contained piece of software, with well-defined interaction points and modes, and is intended to be combined with other components into larger software entities. The same core ideas are also contained in the UML's definition [4]. We therefore also adopt this as the working definition of components for built-in contract testing. The basic rule that governs the use of the UML in a component-based architecture is that individual diagrams should focus on the description of the properties of individual components. The use of a suite of UML diagrams to model a single complex component is depicted in Figure 1

Specification

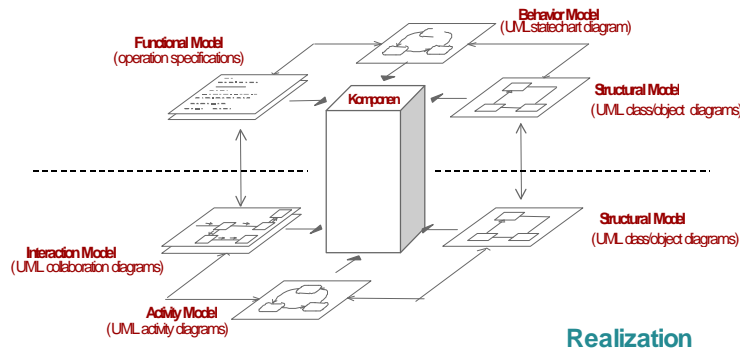


Figure 1: Kobra component model suite.

The specification diagrams collectively define the externally visible properties of the component, and thus in a general sense can be viewed as representing its interface. The diagrams of the structural model describe the types which the component manipulates, the roles with which it interacts and the list of services and attributes which it exports. The diagrams in the functional model provide a declarative description of each of the services or operations supported by the component in terms of pre and post conditions. Finally, the diagrams of the behavioural model describe the externally visible (i.e. logical) states exhibited by the component. These are externally visible internal states that the user of a component needs to know in order to use it.

The realization diagrams collectively define how the component realizes its specifications in terms of interactions with other components and objects. This can include externally acquired, server components, or subcomponents which the component creates and manages itself. The realization diagrams collectively describe the architecture and/or design of the component. The structural diagram is a refinement of the specification structural diagram which includes the additional types and roles needed to realize the component. The interaction diagrams document how each operation of the component is realized in terms of interactions with other components and objects. Finally, the activity diagrams document the algorithm by which each operation is realized.

Every component executes in a certain environment or context that dictates which services it is expected to provide to its clients, and which support it receives in return from its servers. The description of this context is contained within the collection of models generated by a model-driven development method such as the Kobra method. This collection of diagrams represents the basis from which testing artifacts are derived and generated for the built-in contract testing approach.

3 Built-in Contract Testing

Meyer [3] 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 Figure 2, that should ideally be tested to ensure that a component will behave correctly within its deployed environment:

The deployed component (with the thickened border in Figure 2) 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 component implements the services correctly that it is contracted to provide. In other words, clients of the deployed component must ensure that it is meeting its contract. In order to check these properties dynamically when a system is configured or deployed, test software can be built into the components alongside the normal functionality. The test software may be a collection of tests which are 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 Figure 2. The *acquires* and *presumes* associations indicate the relations which must be tested to gain full confidence that the integrated component will inter-operate correctly. If a component C is a client of another component S, the purpose of the test software built into C is to check S by invoking its methods and verify that they behave individually and collectively as expected. Since the tests built into a client check the behavior of a server, they essentially check the semantic compliance of the server to the clientship contract. 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. 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 components.

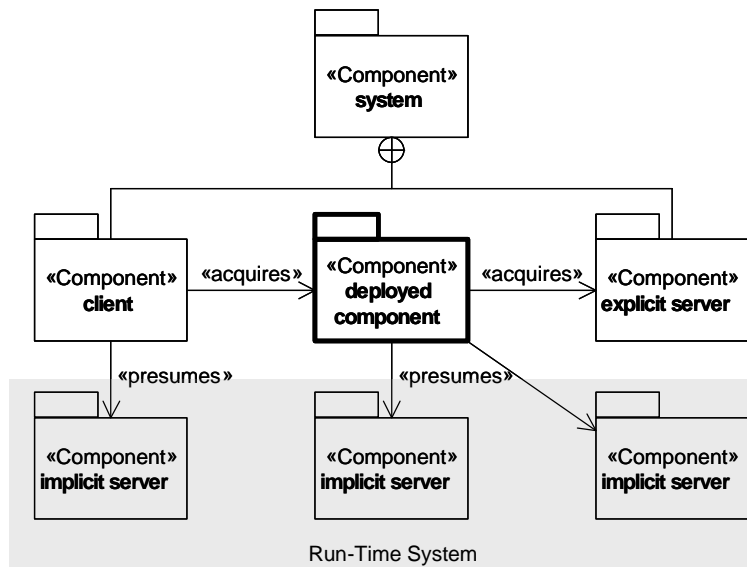


Figure 2: Deployment of a component into a system illustrated through a KobrA containment tree.

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 to 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 in the client in Figure 3) contains tests that check the semantic compliance of the server that the client acquires. The tests inside the client's tester component represent the behaviour that the client expects from its acquired server. We call a client that has test software 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).

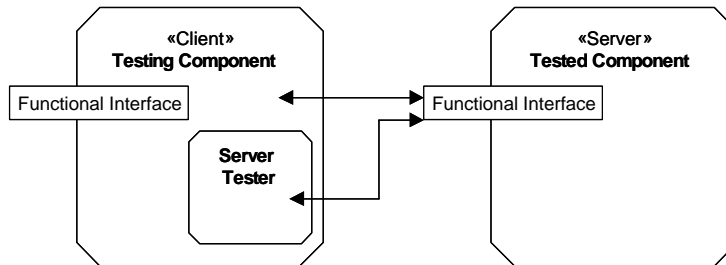


Figure 3: Testing Component with an associated server tester component.

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. This can be a method such as *setServer(Component Server)* that assigns a particular component 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 established, the client will execute its built-in tester components to perform a full semantic check on this new server. The tester executes a simulation on the services of the newly acquired component and may raise an exception if the test fails. The tests may be derived through any arbitrary test case generation technique such as requirements-based, equivalence partitioning, boundary value analysis, or structural testing criteria [5]. According to the applied testing criteria, it may represent an adequate test-suite for the individual unit. 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 component 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). A test case for an abstract data type is therefore always consisting of three parts: ensure pre-condition, execute event, compare post condition with expected post condition. This presents a fundamental dilemma, however. 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).

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 (Figure 4).

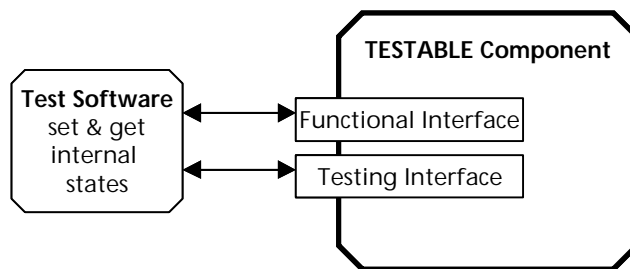


Figure 4: Testable component with an additional contract testing interface.

The basic idea in built-in contract testing is to enhance the normal functional interface of a component with a testing interface which exposes the logical states of the component, and makes them available for setting and checking. The logical states of a component are its externally visible states that the user of the component must know in order to use it properly. These states are part of the component's specification (this is the Kobra specification behavioural model), and they must not be confused with the state model of a component's internal realization. In other words, they are logical states associated with the data type abstraction rather than with a specific implementation. 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. Nevertheless, these externally visible states are entirely different from the internal states of the gear box such as the setting of individual cog wheels, shift rods, shafts, and satellite drives according to the external visible state, that is the gear setting.

A testable component can also contain an introspection interface which provides access to and information about the supported testing interfaces. This may be implemented in form of a Java Interface that realizes a BIT testability contract. This is part of the overall BIT methodology and detailed in [2].

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 explained in [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. State transition testing in this case would comprise a test for each possible individual gear transition according to the KobrA specification behavioural model. If this interface on assertion-checking-basis is used, the initial state of a test's pre condition can be set through the normal functional interface (that is the one being tested), and the correctness of that initial state can be verified through the assertion operations. If the assertion fails, the whole test fails, otherwise the test event is invoked and the final state assessed through the assertion operation.

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 pre-conditions. 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 reimplementations of the component's functionality may be required. However, specific BIT libraries may alleviate this effort as demonstrated in [2].

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 (contract test) [3]. In this way, each individual client can apply its own test suite according to the client's intended use, or usage profile of the component.

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). This concerns the functionality of the component. The models also determine all 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. 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, which are typically distinct initial states for particular tests, and checking the post conditions, which are typically distinct final states plus some output after test execution. Pre- and post conditions for an event are defined in the functional model and in the behavioural 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 realization are variable. One goal of the Component+ project is the provision of a generic library (currently implemented in Java) that supports the definition of the contract testing interface artifacts, including the functional state model [2].

The testing operations *setToState()* and *isInState()* bring the component into a well defined state, and check whether the component is in a well defined state according to the externally visible state model. This model is part of the model driven architecture. 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 state-based 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.

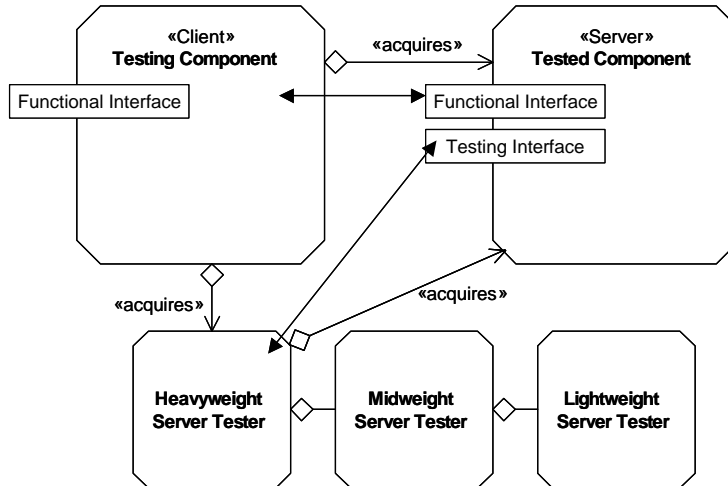


Figure 5: Configuration of the test weight in testing components.

The context sensitivity of the optimal weight of a test creates a dilemma for the reusability of components with built in tests. Since the optimal test is highly dependent on the context in which the component will be used, tests constitute a point of high variation. 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. This may be implemented in the same way as for normal functional variation where a client acquires a component plus a suitable tester for that component. 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. For example, run-time contract tests within an embedded real-time system will have to consider speed and memory limitations, so that a configured contract test will only acquire small and efficient tester components, whereas the dynamic test configuration for highly dependent systems is likely to acquire heavier and more thorough tester components (Figure 5).

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 state transition-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 state model is defined through the states *Neutral*, *Gear1 to Gear5* and *Reverse* representing each feasible gear box setting respectively.

	initial state	pre-conditions	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		toNeutral ()	Neutral
11	Neutral	[momentum < Gear5Momentum]	toGear5 ()	Gear5
12	Gear5		toNeutral ()	Neutral

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

The state transition table contains sufficient information for deriving the testing interface for a testable gear box control component. This comprises the states that are defined through the state model plus the state setting and getting operations. Figure 6 displays a UML-like representation of a testable gear box realized in the two previously mentioned alternative ways. In this case, the 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 (GearBoxControl Tester, Figure 6) can be derived from the state table by using the corresponding interface operations and the test cases (events in table 1).

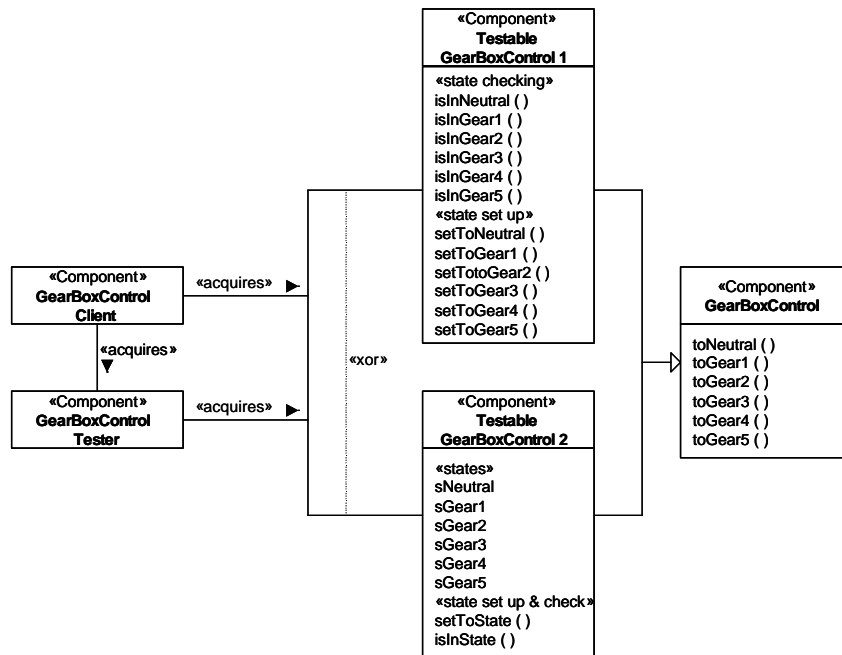


Figure 6: Contract testing artifacts for a gear box control unit. TestableGearBoxControl1/2 represent alternative realizations of the contract testing state set up and checking mechanisms.

The contract tester component can be seen as a particular test driver for a particular test criterion. In this case the test criterion is state transition coverage. 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 clientship contract. This mechanism realizes automatic integration testing.

The fact that the testers are organized in components facilitates their reuse for different clientship contracts. If a 6-gear system is required, the original tester can simply be extended 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.

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 [8] 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 - namely the environment which provides 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 [1], 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.

7 Acknowledgements

This work is partly supported through the German federal department of education and research under the MDTS project acronym, and through the European IST programme framework under the Component+ project (IST-1999-20162).

References

1. Atkinson, C., et al. Component-based Product Line Engineering with UML, Addison-Wesley, 2001.
2. Component+ Project Report D.3, <http://www.component-plus.org>, September 2001.
3. Meyer, B., Object-oriented software construction, Prentice Hall, 1997
4. OMG Unified Modeling Language Specification, Object Management Group, 2000.
5. Pressman, S., Software Engineering: A practitioner's approach, McGraw-Hill, 1997.
6. Somerville, I., Software Engineering, Addison-Wesley, 1995.
7. Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1999.
8. 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.