Component Integration through Built-in Contract Testing

Hans-Gerhard Gross[1], Colin Atkinson[2], and Franck Barbier[3]

[1] Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
grossh@iese.fhg.de
[2] University of Mannheim, D-68131, Mannheim, Germany
atkinson@pi1.informatik.uni-mannheim.de
[3] LIUPPA, Université de Pau et des Pays l' Adour, F-64013 Pau cedex, France
Franck.Barbier@univ-pau.fr

Abstract. This chapter describes a technology and methodology referred to as built-in contract testing that checks the pairwise interactions of components in component-based software construction at integration and deployment time. Such pairwise interactions are also referred to as contracts. Built-in contract testing is based on building test functionality into components, in particular tester components on the client side and testing interfaces on the server side of a pairwise contract. Since building test software into components has implications for the overall component-based development process, the technology is integrated with and made to supplement the entire development cycle starting from requirements specification activities and modeling. The chapter outlines typical specification concepts that are important for built-in contract testing, provides a guide on how to devise built-in contract testing artifacts on the basis of models, and discusses issued involved in using this approach with contemporary component technologies.

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 already existing 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 at deployment time 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. This is because the other components to which it has been connected are intended for a different purpose, have a different usage profile, or are themselves faulty. Current component technologies can help to verify the syntactic

A. Cechich et al. (Eds.): Component-Based Software Quality, LNCS 2693, pp. 159–183, 2003.

compatibility of interconnected components (i.e. that they use and provide the right signatures), but they do little to ensure that applications function correctly when they are assembled from independently developed parts. In other words, they do nothing to check 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. In short, although traditional development time verification and validation techniques can help assure the quality of individual components, they can do little to assure the quality of applications that are assembled from them at deployment time.

1.1 Contracts in Component-Based Development

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 [12], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract. This characterizes 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.

1.2 Contract-Based Integration Testing

The testing approach described in this chapter 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" dynamically at deployment time will fulfill 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 other development phases of the overall software lifecycle. 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. In this chapter we explain the basic principles behind built-in contract testing, and how they affect component-based development principles. Additionally we show how it can be integrated with model-based development.

Since we are talking about components and component-based development, it is important that we initially define the term component and its usage throughout this chapter. We choose to define a software component as a unit of composition with explicitly specified provided, required and configuration interfaces, plus quality attributes [8]. This definition is based on the well known definition of the 1996 European Conference on Object-Oriented Programming [16], that

defines a component as a unit of composition with contractually specified interfaces and context dependencies only that can be deployed independently and is subject to composition by third parties. We have intentionally chosen a broader definition that avoids the terminology *independently deployable*, since we are not specifically restricting ourselves to contemporary component technologies such as CORBA, .NET or EJB/J2EE. In this respect we are closer to Booch's definition which sees a component as a logically, cohesive, loosely coupled module that denotes a single abstraction [7]. 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].

1.3 Structure of This Chapter

Since built-in contract testing is primarily dependent upon specification documents and models we initially introduce (Section 2) typical specification concepts that are required for its application. These represent the basis from which all built-in contract testing concepts are derived and specified. Section 3 introduces functional test generation techniques that may be used for test case design and discusses how the test cases may be derived from the specification documents. These test cases make up the tester components that are integral part of built-in contract testing. The following Section (Section 4) describes the concepts of the technology in detail, the tester components that comprise the tests and are built into the client, and the testing interface that is built into the server of a client/server relationship. Additionally, the Section provides a guide for the development of these artifacts from models. Section 5 discusses the implications of using built-in contract testing with typical component concepts such as reuse, commercial third-party components, and Web-services, and Section 6 summarizes and concludes this chapter.

2 Component Specification

The initial starting point for a software development project is typically a system or application specification derived and decomposed from the system requirements. Requirements and specifications are also the primary source for acceptance and integration testing. 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 aims 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 already existing parts into a meaningful configuration that reflects the predetermined system requirements. In its purest form, component-based development is only concerned with the second item,

representing a bottom-up approach to system construction. 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 does not conform to its original requirements. Component-based development is therefore usually a mixture of top-down decomposition and bottom-up composition activities.

A specification is a set of descriptive documents that collectively define what a component can do. Typically, each individual document represents a distinct view on the subject component, and thus only concentrates on a particular aspect. Whichever notation is used for the documents, a specification should contain everything that is necessary in order to fully use the component and understand its behavior, for composition with other components. As such, the specification can be seen as defining the provided interface of the component. It therefore describes everything that is externally knowable about a component's structure (e.g. associated components) in the form of a structural model, functionality (e.g. operations) in the form of pre- and post conditions, and behavior (e.g. states and state transitions) in the form of a behavioral specification. Additionally, a specification may contain non-functional requirements which represent the quality attributes stated in the component definition. They are part of the quality assurance plan of the overall development project or the specific component. A complete set of documentation for the component is also desirable, and a decision model that captures the built-in variabilities that the component may provide. These variabilities are supported through configuration interfaces.

2.1 Structural Specification

The structural specification defines operations and attributes of the considered subject component, the components that are associated with the subject (e.g. its clients and servers), and constraints on these associations. This is important for defining the different views that clients of component instances can have of the subject. Essentially, this maps to the prospective configurations of the subject, and thus its provided configuration interfaces. A structural specification is not traditionally used in software projects, but the advent of model driven development approaches has increased its importance as a specification artifact. In the form of a UML class or object model, the structural specification provides a powerful way of defining the nature of the classes and relationships by which a component interacts with its environment. It is also used to describe any structure that may be visible at the subject's interface [3].

To illustrate the concepts described in this chapter we will use the well known example of an Automated Teller Machine (ATM). Fig. 1 depicts the structural model of an ATM component (the subject) as a UML class diagram. The structural model only depicts the direct environment of the subject. These are the components with which the ATM interacts.

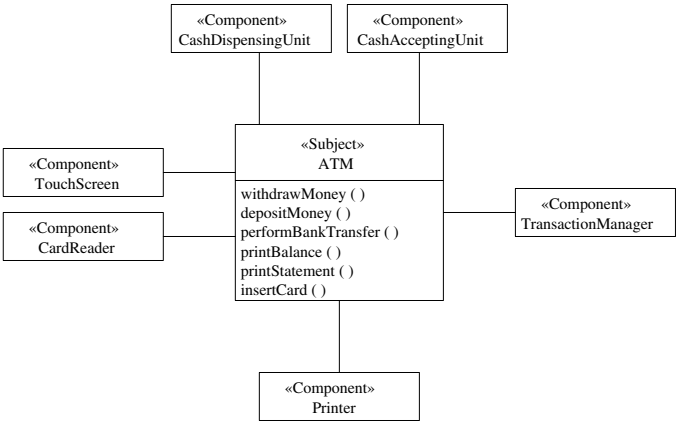

Fig. 1. UML-style structural specification of an ATM application

2.2 Functional Specification

The purpose of the functional specification is to describe the externally visible effects of the operations supplied by the component - that is, its provided interface. Example operation specifications for a *BankingCard* component in the context of the *CardReader* component are depicted in Tables 1 and 2. These provide a full functional specification of the banking card. The respective structural and behavioral models are depicted in Fig. 2.

The most important elements of a functional specification are the *Assumes* and *Result* clauses which represent the pre- and post-conditions for the operation. These are also essential for testing. The *Assumes* clause defines what must be true for the operation to guarantee correct expected execution, and the *Result* clause describes what is expected to become true as a result of the operation if it executes correctly. It is possible to execute an operation if its *Assumes* clause is false, but then the effects of the operation are not certain to satisfy the post-condition (compare with *design by contract* [12]). The basic goal of the *Result* clause is the provision of a declarative description of the operation in terms of its effects. This means it describes what the operation does, and not how. Pre- and post-conditions typically comprise constraints on the provided inputs, the provided outputs, the state before the operation invocation (initial state), and the state after the operation invocation (final state) [3, 11].

2.3 Behavioral Specification

The object paradigm encapsulates data and functionality in one single entity, the object. This leads to the notion of states, and the transitions between states, that typically occur in objects when they are operational. The component paradigm subsumes these. Components may have states too. If a component does not have states, it is referred to as functional object or functional component, meaning

Table 1. Example operation spec. for the banking card component, event *validatePIN*

Name	**validatePIN**
Description	Validates a given Pin and, on success, returns the stored customerDetails. After three unsuccessful invocations (invalid Pin) the card is locked.
Constraints	cardLockRequest from card locks the cardReader (card is not returned to customer)
Receives	Pin: Integer.
Returns	On success: customer details. On failure: invalid Pin error.
Sends	None.
Reads	None.
Changes	None.
Rules	Unless card is locked: return customer details. After third unsuccessful invocation [invalid Pin AND card not locked]: lock the card. After second unsuccessful invocation [invalid Pin AND card not locked]: allow one last unsuccessful attempt. After first unsuccessful invocation [invalid Pin AND card not locked]: allow two more unsuccessful attempts. After no unsuccessful invocations [invalid Pin AND card not locked]: allow three more unsuccessful attempts. One successful invocation clears the card from previous unsuccessful invocations.
Assumes	card not locked AND Number of unsuccessful attempts < 3
Result	(card locked AND Number of unsuccessful attempts $= 3$) XOR (card not locked AND Number of unsuccessful attempts < 3)

Table 2. Example operation spec. for the banking card component, event *unlockCard*

Name	**unlockCard**
Description	Unlocks a previously locked card, so that it may be used again.
Constraints	Only locked cards can be unlocked.
Receives	SecurityPin: Integer.
Returns	On success [valid SecurityPin]: CustomerDetails stored on the card.
Sends	On failure [invalid SecurityPin]: Security Pin Error.
Reads	None.
Changes	None.
Rules	On success [valid SecurityPin]: set card to cleared.
Assumes	Card locked AND (valid SecurityPin OR invalid SecurityPin).
Result	(Card cleared AND valid SecurityPin) XOR invalid SecurityPin.

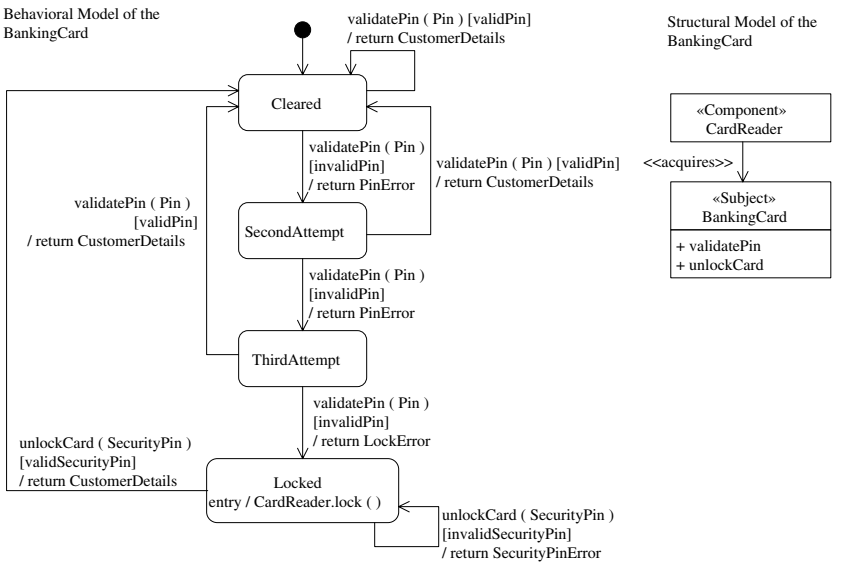

Fig. 2. UML-style behavioral specification and structural specification of an ATM banking card component

it has no internal attributes that are exhibited through its provided interface. In other words, a pure functional component does not exhibit externally visible states and transitions. It may, however, have internal states that are not externally visible.

The purpose of the behavioral specification (or the behavioral model) is to show how the component behaves in response to external stimuli [3] and changes its externally visible states. It concentrates on the *Assumes* and *Result* clauses of the functional specification that define the pre- and post-conditions of the operations (Tables 1 and 2). If the component has externally visible states, and most components do, the behavioral model succinctly expresses much of the complexity that is collectively contained in the pre- and post-conditions of the component's operations.

The behavioral model describes the instances of a component in terms of their observable states, and how these states change as a result of external stimuli that affect the component instance [11, 6]. A state is a particular configuration of the data values of an object's internal attributes. A state itself is not visible. What is visible or externally observable is a difference in behavior of the component from one state to another when stimuli are sent. In other words, if the same message is sent to a component instance twice, the instance may behave differently, depending on its original state before the message is received. A transition, or change from one state into another is triggered by an event, which is typically a message arrival. A guard is a condition that must be true before a transition can be made. Guards are used for separating transitions to various states that are based on the same event [11]. A behavioral specification may be represented

Table 3. Behavioral specification according to the model in Fig. 2.

Initial State	Pre Condition	Event	PostCondition	Final State
Cleared	[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
Cleared	[Invalid Pin]	validatePin (Pin)	PinError returned	Second Attempt
Second Attempt	[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
Second Attempt	[Invalid Pin]	validatePin (Pin)	PinError returned	Third Attempt
Third Attempt	[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
Third Attempt	[Invalid Pin]	validatePin (Pin)	LockError returned	Locked
Locked	[Valid SecurityPin]	unlockCard (SecurityPin)	CustomerDetails returned	Cleared
Locked	[Invalid SecurityPin]	unlockCard (SecurityPin)	SecurityPinError	Locked

through one or more UML state diagrams or state tables. Fig. 2 displays the behavioral model (plus the structural model) for the *BankingCard* component with two public operations, and Table 3 displays the corresponding state table.

2.4 Component Realization

The specification does not provide sufficient information to implement a component. Before that is possible it is necessary to define how a component will realize its services. The component realization is a set of descriptive documents that collectively define how a component realizes its specification. A higher-level component is typically realized through, and composed of, a combination of lower-level components that are contained within, and act as servers to, the higher-level component. Once it has been defined which parts of a logical higher level component will be implemented through which sub-components, the implementation of the higher level component can be started.

A component 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 considered component and not implemented through sub-components. These items correspond to the component's private design that the user of the component does not see. The user or the system integrator only cares about a component's provided and required interfaces, because these define the context into which the component will be integrated.

3 Specification- and Model-Based Component Testing

The previous Section described the initial starting point for using and applying built-in contract testing in component development - namely a sound specification, ideally in the form of documents that directly support the generation of the built-in testing architecture and the built-in test suites. In this Section we discuss how the specification can be used for the generation of test artifacts for integration testing.

Since component specifications are the primary source of information for the development of built-in contract test architectures and test suites, built-in contract testing is primarily concerned with functional testing techniques that view an integrated component as a black box. However, black and white-box testing cannot be strictly separated in component testing. Component engineering takes a fractal-like view of software systems in which components are made of other components that in turn are made of other components in a recursive manner. The terminology of black and white boxes has only a meaning for the level of abstraction that we are looking at. A white box test for a super-ordinate component maps to a black box test for a sub-ordinate component and so on. Thus, testing in the traditional code-based-testing sense has no meaning in component integration testing since we are only concerned with the testing of interfaces.

3.1 Functional Testing

Functional testing techniques completely ignore the internal mechanisms of a system or a component (its internal implementation) and focus solely on the outcome generated in response to selected inputs and execution conditions [10].

Domain Analysis and Partitioning Testing. Typical representatives of functional testing techniques are domain analysis testing and partitioning techniques. A domain is defined as a subset of the input space that somehow affects the processing of the tested component. Domains are determined through boundary inequalities, algebraic expressions that define which locations in the input space belong to the domain of interest [5]. Domain analysis is used for and sometimes also referred to as partitioning testing. Many functional test case generation techniques are based upon partition testing. Equivalence partitioning is a strategy that divides the set of all possible inputs into equivalence classes. The equivalence relation defines the properties for which input sets are belonging to the same partition, for example equivalent behavior (state-transitions). Proportional equivalence partitioning, for example, allocates test cases according to the probability of their occurrence in each sub-domain.

State-Based Testing. State-based testing concentrates on checking the correct implementation of the component's state model. Test case design is based on the individual states and the transitions between these states. In object-oriented or component-based testing effectively any type of testing is state-based as soon as

the object or component exhibits states, even if the tests are not obtained from the state model. In this case, there is no test case without the notion of a state or a state-transition. In other words, pre- and post-conditions of every single test case must consider states and behavior. State-based testing comprises a number of test coverage strategies: Piecewise coverage concentrates on exercising distinct specification pieces, for example coverage of all states, all events, or all actions. This technique is not directly related to the structure of the underlying state machine that implements the behavior, so it is only accidentally effective at finding behavioral faults [6]. Transition coverage is achieved through a test suite if every specified transition in the state model is exercised at least once. As a consequence, this covers all states, all events and all actions. Transition coverage may be improved if every specified transition sequence is exercised at least once [6]. This is also a method sequence based testing technique. Round-trip path coverage is defined through the coverage of at least every defined sequence of specified transitions that begin and end in the same state. The shortest round-trip path is a transition that loops back on the same state. A test suite that achieves full round-trip path coverage will reveal all incorrect or missing event/action pairs [6].

Method-Sequence-Based Testing. This test case generation technique concentrates on the correct implementation of a component's combinations, or sequences of provided operations. Test case design is based on the behavioral model, such as a UML state chart diagram. Here, the paths through the state model are checked. This may also include multiple invocations of the same operation. Method sequences are typical representatives of usage profiles, that is a profile of how a client uses the services of a component. Table 4 shows an excerpt of a typical test suite based on method sequences for the *BankingCard* component.

3.2 Model-Based Testing

Model-based testing concentrates on how tests may be derived from graphical specification notations such as the UML. Such techniques are traditionally used in the development of safety critical and real-time systems (e.g. Petri Nets), and more recently it concentrates upon approaches for how to derive test information from individual UML models.

Models represent a solid foundation for test case generation that is primarily based on the specification, and are therefore mainly functional. Models use powerful (semi-) formal abstract notations in order to express requirements specifications. Having good requirements is crucial not only for the development of a system but additionally for the development of its testing infrastructure. If requirements are additionally testable they are the perfect source for instant test scenario generation.

Class and Package Diagrams. Class diagrams represent structure, that is associations between entities plus externally visible attributes and operations

of classes (or components). They are a valuable source for testing. Specification class diagrams (server) represent the interfaces that individual components export to their clients and therefore show which operations need to be tested, which operations support the testing and which external states are important for a unit. These can directly guide the construction of tester components for a server component. Realization class diagrams (client) represent the operations of the servers that a client is associated with. They only contain externally visible server operations and attributes that a client is actually using. It means such a diagram restricts the operational profile of a client in terms of operations. This helps to determine the range of operations that a tester component must consider. Class diagrams may be used to generate test cases according to boundary conditions and component interaction criteria [6]. Package diagrams represent a similar source as class diagrams although on a coarser grained level of abstraction. In built-in contract testing such component diagrams (component trees) are used to indicate variability in an application and therefore mark the associations between components that need to be augmented with built-in contract testing artifacts.

State Diagrams. State diagrams are a valuable source for testing in many ways. This is also demonstrated in Table 4. State diagrams are the primary source for test case generation in built-in contract testing for development of tester components as well as testing interfaces. State diagrams concentrate on the dynamics of components in terms of externally visible states and transitions between the states. State chart diagrams may also be used to generate test cases according to class hierarchy and collaboration testing criteria [6].

Collaboration Diagrams. While state diagrams concentrate on the behavior of individual objects, UML collaboration diagrams represent the behavioral interactions between objects. They describe how the functions of a component are spread over multiple collaborating entities (i.e. sub-components) and how they interact in order to fulfill higher-level requirements. Collaboration diagrams represent two views on an entity, a structural view, and a behavioral view. Additionally, they pose constraints on a system. Since collaboration diagrams realize a complete path for a higher-level use case they may be used to define complete message sequence paths according to the use case [1].

Use Cases, Operational Profiles, and Scenarios. Many organizations define use cases as their primary requirements specifications, for example [13]. Additionally, they use operational profiles in order to determine occurrences and probabilities of system usage. Use case models thereby map to operations in an operational profile. Another application of use cases is the generation of state chart diagrams [14] from use-case driven requirements engineering, or the generation of collaboration diagrams. Use cases may be used to generate test cases according to combinational function and category partitioning criteria [6]. Scenarios are used to describe the functionality and behavior of a software system

Table 4. Test case design based on method sequences according to the behavioral model of the banking card

#	Initial State	Pre Condition	Event	PostCondition	Final State
1	Cleared	[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
2	Cleared	[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
3	Cleared	[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Valid Pin]	validatePin (Pin)	CustomerDetails returned	Cleared
4	Cleared	[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Invalid Pin]	validatePin (Pin)	PinError returned	
		[Invalid SecurityPin]	unlockCard (SecurityPin)	Invalid SecurityPin returned	
		[Valid SecurityPin]	unlockCard (SecurityPin)	CustomerDetails returned	Cleared
5	...	...	...	...	...

from the user's perspective in the same way as use cases. Scenarios essentially represent abstract tests for the developed system that can be easily derived by following a simple process. This is laid out in the SCENT Method [15].

4 Specification of the Contract Testing Artifacts

The previous Section described which specification documents may be used to generate test data for the test suites of built-in contract testing. This Section concentrates on the description of the contract testing architecture and explains the nature of the contract testing interface on the server side, and the contract tester component on the client side of a component relationship. The test suites are contained within the tester components.

Meyer [12] defines the relationship between an object 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). Built-in contract testing focuses on verifying these pairwise client/server interactions between two components when an application is assembled. This is typically performed at deployment time when the application is configured for the first time, or later during the execution of the system when a re-configuration is performed.

The previous sections have laid out the foundations for the development of the built-in contract testing artifacts. They can be seen as an entry criterion for using built-in contract testing. This is a sound development process that is ideally based on models, though other notations may be acceptable as long as they provide similar contents, plus a testable requirements specification (that is part of the method) from which the tests may be derived. Additionally, we need to define the test target in terms of a quality assurance plan [3] that determines the testing techniques for deriving the individual test cases. This may be a selection of the test case generation techniques that we have introduced in the previous Section (Section 3).

The two primary built-in contract testing artifacts are the *server tester component* that is built into the client of a component in order to test the server when it is plugged to the client, and the *testing interface* that is built into (extends) the normal interface of the server and provides introspection mechanisms for the testing by the client.

4.1 Built-in Server Tester Components

Configuration involves the creation of individual pairwise client/server relations between the components in a system. This is usually done by an outside "third party", which we refer to as the context of the components (Fig. 3). This creates the instances of the client and the server, and passes the reference of the server to the client (i.e. thereby establishing the clientship connection between them). This act of configuring clients and servers may be represented through a special association that is indicated through an ≪acquires≫ stereotype as illustrated in Fig. 2 or in Fig. 3. The context that establishes this connection may be the container in a contemporary component technology, or it may simply be the parent object.

In order to fulfill its obligations towards its own clients, a client component (e.g. *CardReader* in Fig. 3) that acquires a new server (e.g. *BankingCard* in Fig. 3) must verify the server's semantic compliance to its clientship contract. In other words, the client must check that the server provides the semantic service that the client has been developed to expect. The client is therefore augmented with in-built test software in the form of a server tester component (e.g. *BankingCardTester* in Fig. 3), and this is executed when the client is configured to use the server. In order to achieve this, the client will pass the server's reference to its own in-built server tester component. If the test fails, the tester component may raise a contract testing exception and point the application programmer or system integrator to the location of the failure.

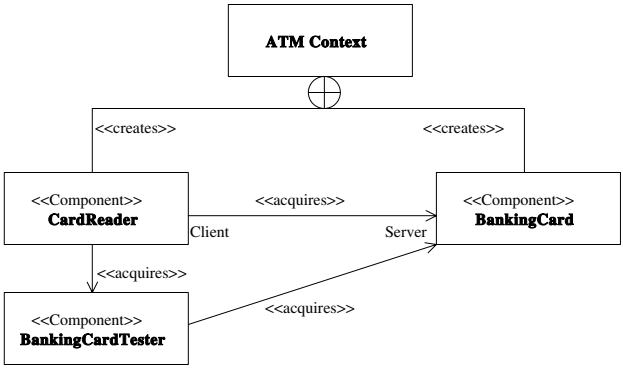

Fig. 3. Structural model of built-in contract testing *server tester component* for the *BankingCard/CardReader* example

4.2 Built-in Contract Testing Interfaces

The object-oriented and component-based development paradigms build on the principles of abstract data types which advocate the combination of data and functionality into a single entity. State transition testing is therefore an essential part of component verification. In order to check whether a component's operations are working correctly it is not sufficient simply to compare their returned values with the expected values. The compliance of the component's externally visible states and transitions to the expected states and transitions according to the specification state model must also be checked. These externally visible states are part of a component's contract that a user of the component must know in order to use it properly. However, because these externally visible states of a component are embodied in its internal state attributes, there is a fundamental dilemma.

The basic principles of encapsulation and information hiding dictate that external clients of a component should not see the internal implementation and internal state information. The external test software of a component (i.e. the built-in contract tester component) therefore cannot get or set any internal state information. The user of a correct component simply assumes that a distinct operation invocation will result in a distinct externally visible state of the component. However, the component does not usually make this state information visible in any way. This means that expected state transitions as defined in the specification state model cannot normally be tested directly.

The contract testing paradigm is therefore based on the principle that components should ideally expose externally visible (i.e. logical) state information by extending the normal functional server as displayed in Fig. 4. In other words, a component should ideally not only expose its externally visible signatures, but additionally it should openly provide the model of its externally visible behavior. A testing interface therefore provides additional operations that read from and write to internal state attributes that collectively determine the states

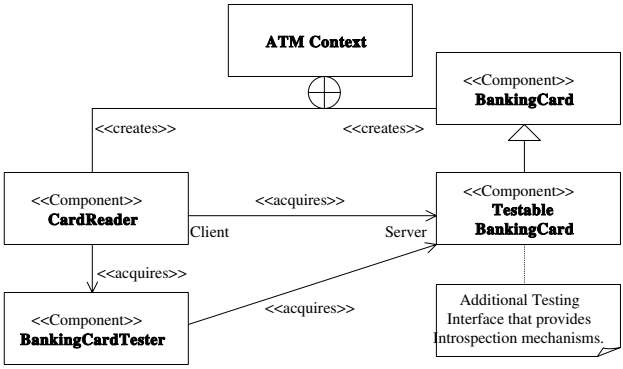

Fig. 4. Structural model of a *testing interface* or *testable component* for the ATM example

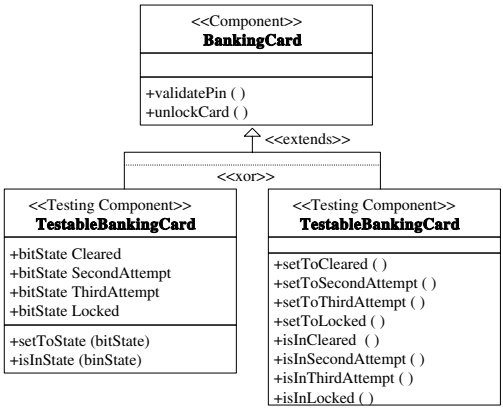

Fig. 5. Class diagram of a testable *BankingCard* component with testing interface operation that reflect the behavioral model. Two alternative implementations are feasible

of a component's behavioral model. A component that supports its own testing by external clients through an additional testing interface in this way is called *testable component*. Fig. 5 displays two alternative implementations for a *TestableBankingCard* component that exposes a testing interface according to its behavioral model depicted in Fig. 2. The first implementation type defines a number of public state variables that represent the states of the behavioral model and act as input parameter for the *setToState* and *IsInState* testing interface operations. The first operation sets the component to one of the defined states and the second one checks whether the component is in a particular state. The second implementation type defines an individual *setToState* and *isInState* operation per state in the behavioral model.

4.3 Component Associations in Built-in Contract Testing

The previous sub-sections have introduced the two primary artifacts that must be generated for built-in contract testing: tester component and testing interface. These represent additional functionality whose focus is testing. The first one extends the client component and contains the actual test cases that check the client's deployment environment. The second extends the interface of the server in order to make the server more testable.

If a server does not provide a testing interface (e.g. a COTS component) it does not mean that contract testing may not be used. It is simply limited with regard to testing controllability and observability. The test cases in the client must be designed differently as well according to the missing testing interface.

The Client and the Client's Tester Component. In the client role, a component may own and contain its tester component. This means that the test cases, typically organized as components in their own right, are permanently encapsulated and built into the client. This is the simplest form of built-in contract testing, and it provides no direct run-time configurability with respect to the type and amount of testing the client component will perform when it is connected to its server components. This association can be expressed through the UML composition relationship.

A more flexible way of built-in contract testing is realized through a loosely associated tester component that may be acquired by the testing client in the same way it acquires any other external resources. Here, the component provides a configuration interface through which any arbitrary tester component that represents the client's view on a tested server may be set. This provides flexibility in terms of how much testing will be performed at deployment time, and additionally it provides flexibility as to which type of tester will be applied according to an instantiated product line. Such an association may be represented through a UML aggregation association, or more specifically through the stereotype ≪acquires≫ that identifies the tester component as an externally acquired server (e.g. Fig. 3 and 4).

The Server and the Server's Testing Interface. In a server role, a component must be much more closely connected to its testing interface because its implementation must be able to access the server's internal variables (i.e. for setting and getting the states). The testing interface is therefore directly built in to the component and extends its normal functionality with additional testing functionality.

More flexible is a typical extension (inheritance) mechanism. Enabling and disabling the built-in testing interface can then be achieved simply by instantiating the desired type of object accordingly. Such an association can be indicated through the UML extension symbol plus the ≪extends≫ stereotype. In any case, the testing interface of a component must be visible at its external boundary. For components with nested objects it means that each of these sub-objects must be dealt with individually inside the component.

The Client's Tester and the Server's Testing Interface. The tester component of the client and the server's testing interface inter-operate in the same way as their respective functional counterparts. Since testers and testing interfaces are directly built into the system, testing represents only additional functionality that happens to be executed at component integration. The tester component must only "know" the reference of the tested server, and this is simply passed into the tester component when the clientship relation is established. Testing in this respect is only executing some additional code that uses some additional interface operations. Therefore, built-in contract testing is initially only a distinct way of implementing functionality that is executed when components are interconnected during deployment. This only concerns the architecture of a system (i.e. which components will expose additional interfaces, which components will contain tester components). The test cases inside a tester component are arbitrary, and they can be developed according to traditional criteria or any of the criteria introduced in Section 3.

4.4 Fitting It All Together – The Contract Testing Method

In a development project where no built-in contract testing is applied at the component level, the following steps may be applied in order to add built-in integration testing artifacts to test the individual semantic component interactions. Once these artifacts have been introduced, they can be reused in the same way as the components' functionality is reused in subsequent projects. In other words, every repeated integration of such an augmented component with other components in a new project can be automatically verified.

Step 1 – The Tested Interactions and the Contract Testing Architecture. In general, any client-server interaction may be augmented with built-in testing interfaces and tester components. Such interactions are represented through any arbitrary association in a structural diagram.

Associations between classes that are encapsulated in a reusable component are likely to stay fixed throughout a component's life cycle. Such associations may be augmented with removable built-in contract testing artifacts for development-time component integration testing. Such variability in testing may be implemented through a development- or compile-time configuration mechanism (e.g. include in C++), or through a run-time configuration interface that dynamically allocates tester components, and testable components with testing interfaces.

Components as units of integration will have permanent built-in testing interactions at their boundaries. This means that every external association that requires or imports an interface will be permanently augmented with a built-in contract tester component whose tests reflect its expectations towards its servers, and every external association that provides or exports an interface will be permanently augmented with a built-in testing interface.

The stereotype ≪acquires≫ represents dynamic associations that may be configured according to the needs of the application (i.e. component boundaries). The decisions about where contract testing artifacts will be built in are

documented in the structure of the system. This is simply an additional software construction effort in the overall development process that adds functionality to the application. For any client-server relationship we will have to add testing interface and tester component as modeled in Fig. 4.

Step 2 – Specification of the Testing Interface and the Tester Component. The testing interface is used to set and retrieve state information about the tested server component. This is defined in the server's behavioral model. Each state in this model represents a setting for which the behavior of some operation is distinctively different from any other settings. The individual states that the behavioral model defines is therefore an ideal basis for specifying state setting and checking operations. Each state in the state model therefore maps to one state setting and one state checking method, according to which strategy is used (compare with the class diagram in Fig. 5).

The tester components are developed according to the expectation of the testing client. In other words, a client's specification (e.g. its behavioral model) represents a description of what the client needs from its environment in order to fulfill its own obligations. It represents the expectation of the testing component on its environment. So, the tests are not defined by the specification of the associated server - in this case it would only be a unit test of the server that the producer of such component may already have performed.

For example, the tester component for a *BankingCard* component that is built into the client *CardReader* component may comprise tests according to method sequence testing criteria as defined in Table 4. If the associated server component *BankingCard* exports a built-in contract testing interface according to the structural definition in Fig. 5 (first/left alternative implementation) the tester component can apply an integration test sequence as displayed in Table 5. This corresponds to the specification of test case # 4 in Table 4.

Step 3 – Component Integration. Once all the functional component artifacts and the built-in contract testing component artifacts on both sides of a component contract have been properly defined and implemented, the two components can be integrated (plugged together). This follows the typical process for component integration, i.e. a wrapper is defined and implemented for the client or the server, or an adaptor is designed and implemented that realizes the mapping between the two roles. Since the testing artifacts are integral parts of the individual components on either sides of the contract they are not subject to any special treatment, they are just treated like any normal functionality. For example, an adaptor takes the operation calls from the client and transforms them to into a format that the server can understand. If the server produces results, the adaptor takes these and translates them back into the format of the client. Since the built-in contract testing artifacts are part of the client's and server's contracts they will be mapped through the adaptor as well. Component platforms such as CORBA Components already provide support for this type of mapping.

Table 5. Example test case for a *BankingCard* Tester Component that will be built into the *CardReader* component

Initial State Setup	Operation Invocation & Parameter Constraints	Expected Outcome & Final State
SetToState(Cleared)	validatePin (Pin) AND [Invalid Pin]	PinError expected
	validatePin (Pin) AND [Invalid Pin]	PinError expected
	validatePin (Pin) AND [Invalid Pin]	PinError expected
	unlockCard (SecurityPin) AND [Invalid SecurityPin]	SecurityPinError expected
	unlockCard (SecurityPin) AND [Valid SecurityPin]	CustomerDetails expected AND IsInState(Cleared)

5 Built-in Contract Testing and the Component Paradigm

The previous Section introduced the basic principles that guide the development of built-in contract testing artifacts for the two roles in a client/server-relationship between components. These are valid for both object-oriented and component-based development. In this Section we describe how these ideas can be integrated with mainstream industrial component technologies.

5.1 Built-in Contract Testing and Reuse

Testing takes a big share of the total effort in the development of large and/or complex software. Nevertheless, component-based software engineering has mainly focused on cutting development time and cost by reusing functional code. If the components cannot be used in new target domains without extensive re-work or re-testing, the time saving becomes questionable [9]. Hence, there is a need to reuse not only functional code but also the tests and test environments that verify a component's interactions on the target platform. To attain effective test reuse in software development, there are several aspects that must be taken into account.

Contract testing includes a flexible architecture that focuses on these aspects. It is the application of this architecture that makes reuse possible. In the initial approach of built-in testing as proposed by Wang et. al. [17], complete test cases are put inside the components and are therefore automatically reused with the component. While this strategy seems attractive at first sight, it is not flexible enough to suit the general case. A component needs different types of tests in different environments and it is neither feasible nor sensible to have them all built-in permanently.

Under the contract testing paradigm test cases are separated from their respective components and put in separate tester components. The components still have some built-in test mechanisms, but only to increase their accessibility for testing by clients. The actual testing is done by the tester components that are associated with the client components through their interfaces. In this way, an arbitrary number of tester components can be connected to an arbitrary number of functional components. This offers a much more flexible way to reuse tests as they do not have to be identical to the ones originally delivered with a component. The tests can be customized to fit the context of the component at all stages in the component's life cycle.

The overall concept of test reuse in built-in contract testing follows the fundamental reuse principles of all object and component technologies. Because testing is inherently built into an application or parts thereof (the components) testing will be reused whenever functionality is reused. In fact testing in this respect is normal functionality. Only the time when this functionality is executed distinguishes it from the other non-testing functionality, that is at configuration or deployment time.

5.2 Built-in Contract Testing and Commercial Off-the-Shelf (COTS) Components

The integration of commercially available third party components into new applications represents one of the main driving factors for component-based software development since it greatly reduces the effort involved in generating new applications. Such components are aimed at solving typical problems in distinct domains, and ideally they can be purchased off-the-shelf and simply plugged into an application. However, this ideal scenario is still some way from reality. Although they reduce the effort involved in achieving new functionality, third party components typically increase the effort involved in integrating the overall application, since they are typically available in a form that presents some integration difficulties. For example, they may provide syntactically or semantically different interfaces from what is expected and required, or they may not be entirely fit for the intended purpose. In any way, the usage and integration of third party components typically requires either the development of wrappers or adaptor components that hide and compensate for these differences, or changes in the design and implementation of the integrating client component.

Once a new component can communicate syntactically with a provided COTS component through some mechanism, the next step is to make sure that they can also communicate semantically. In other words, the fact that two components are capable of functioning together says nothing about the correctness of that interaction. This is where built-in contract testing provides its greatest benefits.

Ideally, all commercially available components should provide testability features such as the introspection mechanism that is realized through built-in contract testing interfaces. Such components will naturally fit into applications that are driven by the built-in contract testing paradigm. They simply need to be interconnected syntactically, and this of course includes the functional-

ity as well as the testing aspects of the two components. The built-in contract testing paradigm not only suggests that testing interfaces should be provided with commercial components but also that these should be provided according to well-defined templates, so that the syntactic integration effort may eventually be removed completely. However, since the technology has not yet penetrated into the component industry, it is likely that component vendors will not provide their components with such testability features.

Commercial third party components cannot typically be augmented with an additional built-in contract testing interface that provides a client with an introspection mechanism for improved testability and observability. This means that COTS components can only be tested through their provided functional interface, as is traditionally the case in object and component testing. However, modern object languages or component platforms such as Java do provide mechanisms that enable internal access to a component. They can break the encapsulation boundary of binary components in a controlled way and offer internal access. Such mechanisms can be used to realize testing interfaces according to the built-in contract testing philosophy for any arbitrary third party component. [4] describe how this can be achieved using Java's reflection mechanism in the form of a suitable Java Library. Here, we only give a brief overview. The architecture of the library is displayed in Fig. 6. Built-in contract testing can initially be carried out by using three primary concepts. These are the testability contract, the tester and test case. These are the fundamental features that support the assessment of test results, control of the execution environment, and actions to be taken if faults are encountered. Additionally, the library provides state based testing support that is more essential to built-in contract testing. These concepts are the state-based testability contract, the state-based tester, and the state-based test case. The state-based concepts abide by the principles of Harel's state machines [4].

5.3 Built-in Contract Testing and Web-Services

Web-Services are commercial software applications that are executed on remote hosts and provide individual services which are used to realize distributed component-based systems. Web-Services fulfill all the requirements of Szyperski's component definition [16], that is a service is only described and used based on interface descriptions and more importantly, it is independently deployable. This means that a Web-Service provides its own run-time environment, so that a component-based application is not bound to a specific platform. Every part of such an application is entirely independent from any other part, and there is no overall run-time support system but the underlying network infrastructure. Web-Services represent the ultimate means of implementing component-based systems.

Contract testing provides the ideal technique for checking dynamic and distributed component-based systems that are implemented through Web-Services. In fact this is the scenario for which built-in contract testing provides the most benefits. The syntactic compatibility between a client and a Web-based server

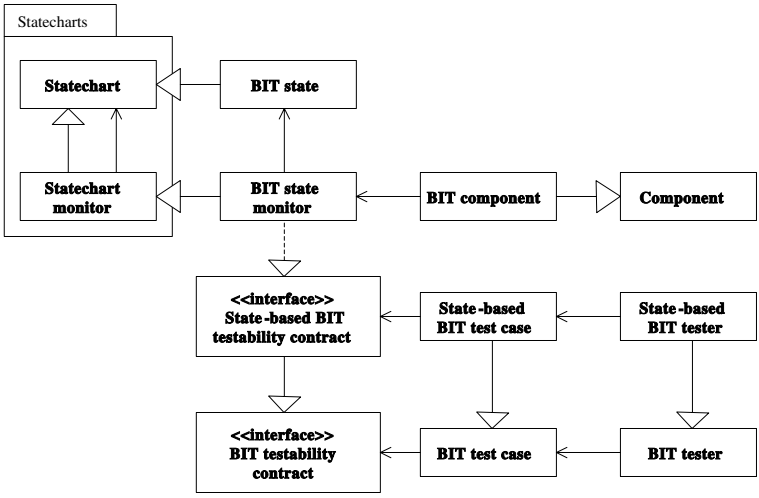

Fig. 6. Organization of the built-in contract testing Java support library [4]

is ensured through an XML mapping between their interfaces. Their semantic compatibility is checked through the built-in server tester components inside the client that are executed to validate the associated server. These tests are performed when the client is registering with a service for the first time (i.e. during configuration,) or if the client requests the same specification of the server from a different Web-Service provider (i.e. during re-configuration).

Fig. 7 displays a sample containment hierarchy of a remote connection between the ATM component (in the teller machine) and the bank's transaction manager that collects all ATM transactions. The ≪remotely acquires≫-relationship indicates that the *TransactionManager* (in this case a testable component) is not locally available. This means that this relationship will be implemented through some underlying networking infrastructure. The stereotype ≪remotely acquires≫ hides the underlying complexity of the network implementation and only considers the level of abstraction that is important for testing. This technique is termed stratification [2]. As soon as the connection between the two interacting components is established, a normal contract test may be initiated regardless of how the connection is realized in practice. The server provides a suitable test interface that the client's built-in tests can use. The client and server do not "know" that they are communicating through Web-Interfaces. This connection is established through their respective contexts when the context of the *ATM* component registers with the context of the *TransactionManager* component (Fig. 7).

Web-Services typically provide instances that are ready to use. As a result, the server component that is provided through the remote service is already configured and set to a distinct required state. A run-time test is therefore likely to change or destroy the server's initial configuration, so that it may not be usable by the client any more. Clearly, for the client, such a changed server is of

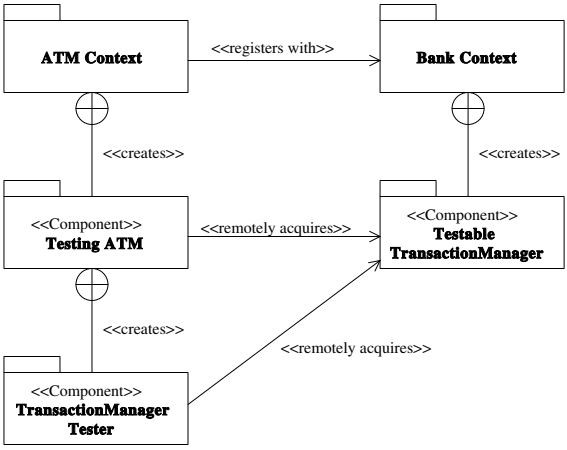

Fig. 7. Association between two remotely connected components based on a WEB-Service implementation

no use and this creates a fundamental dilemma for built-in contract testing of Web-Services.

Under object-oriented run-time systems the client can solve this dilemma by simply creating a clone of the tested component and passing the clone to the test software. This works because client and server are handled by the same run-time environment. For example, in Java this is performed through the *Object.clone* method. In this case, the test software may completely corrupt the newly created clone without any effect on the original instance, it is simply thrown away after the test, and the original is used as working server. However, in a Web-Service environment the run-time system of the client is different from that of the server, so that the client cannot construct a new instance from an existing one. The client and server are residing within completely different run-time scopes on completely different network nodes. Contract testing can therefore only be applied in a Web-Service context if the Web-Service provides some way for the client to have a clone created and accessed for testing. Some contemporary component technologies such as CORBA Components are capable of doing exactly that.

6 Summary and Conclusions

This chapter has described the methodology and process of built-in contract testing for component integration in model-driven component-based application construction. It is a technology that is based on building the test software directly into components - built-in tester components at the client side of a component interaction and built-in testing interfaces at its server side. In this way, a component can check whether it has been brought into a suitable environment, and an environment can check whether each component provides the right services. This enables system integrators who reuse such components to validate immediately

and automatically whether a component is working correctly in the environment of a newly created application.

Built-in contract testing delivers the same basic benefits as any reuse technology. The effort of building test software into individual components is paid back depending on how often such a component will be reused in a new context, and a component is reused depending on how easily it may be reused in new contexts. Built-in contract testing greatly simplifies the reuse of a component because once it has been integrated syntactically into a new environment its semantic compliance with the expectations of that new environment may be automatically assessed.

Current and future work focuses on the integration of built-in contract testing with an abstract test description notation, the TTCN-3 technology, and the extension of the contract term to cope with non-functional (i.e. response time) requirements.

Acknowledgements

This work has been partially supported by the EC IST Framework Programme under IST-1999-20162 Component+ project, and the German National Department of Education and Research under the MDTS project acronym.

References

1. A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *3rd Intl. Conf. on the Unified Modeling Language (UML'00)*, pages 383–395, York, UK, October 2000.
2. C. Atkinson, C. Bunse, H.-G. Gross, and T. Kühne. Component model for web-based applications. *Annals of Software Engineering*, 13, 2002.
3. C. Atkinson and et al. *Component-Based Product-Line Engineering with UML*. Addison-Wesley, London, 2001.
4. F. Barbier, N. Belloir, and J.M. Bruel. Incorporation of test functionality in software components. In *2nd Intl. Conference on COTS-based Software Systems*, Ottawa, Canada, February, 10.-12. 2003.
5. B. Beizer. *Black-box Testing, Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, 1995.
6. R. Binder. *Testing Object-Oriented Systems - Models, Patterns and Tools*. Addison-Wesley, 2000.
7. G. Booch. *Software Components with Ada: Structures, Tools and Subsystems*. 1987.
8. Component+. Built-in testing for component-based development. Technical report, Component+ Project, http://www.component-plus.org, 2001.
9. D.S. Guindi, W.B. Ligon, W.M. McCracken, and S. Rugaber. The impact of verification and validation of reusable components on software productivity. In *22nd Annual Hawaii International Conference on System Sciences*, pages 1016–1024, 1989.
10. IEEE. *Standard Glossary of Software Engineering Terminology*. IEEE Std-610.12-1990, 1999.

11. J. McGregor and D. Sykes, editors. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.
12. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.
13. S. Meyer and R. Sandfoss. Applying use-case methodology to SRE and system testing. In *STAR West Conference*, October 1998.
14. Quasar. German national funded quasar project. Technical report, http://www.first.gmd.de/quasar/, 2001.
15. J. Ryser and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. In *12th Intl. Conf. on Software and Systems Engineering and their Applications (ICS-SEA'99)*, Paris, France,, 1999.
16. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
17. Y. Wang, G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross. On built-in test reuse in object-oriented framework design. In *ACM Journal on Computing Surveys*, volume 32, March 2000.