Available online at www.sciencedirect.com

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 111 (2005) 161–182

www.elsevier.com/locate/entcs

Model-Based Built-In Tests [⋆]

Hans-Gerhard Gross [1]

Fraunhofer IESE, Kaiserslautern, Germany,

Ina Schieferdecker [2]

Technical University Berlin
Fraunhofer FOKUS, Berlin, Germany,

George Din [3]

Fraunhofer FOKUS, Berlin, Germany,

Abstract

Assembling new software systems from prefabricated components as an attractive alternative to traditional software development practices is more and more investigated. Component technologies like CCM, .Net or EJB are accompanied by model-based approaches like MDA. However, still the emphasis is rather on system design and development and not on system validation and testing. The expected reductions in development time and effort will only arise if separately developed components can be made to work effectively together with minimal effort. Lengthy and costly in-situ verification and acceptance testing directly undermines the benefits of heterogeneous components and late system integration. This paper extends contract-based built-in tests where components are equipped with the ability to check their execution environment at run-time with approaches to derive built-in tests from system models, represent them on model level, and to generate executable tests from these test models. This model-based approach increases the automation level in generating and realizing built-in tests and therefore also increases the quality of and reduces needed resources for developing built-in tests.

Keywords: UML Modeling, Model-based Testing, Test Modeling, Test Automation, Testing and Test Control Notation, Component-based Testing

doi:10.1016/j.entcs.2004.12.001

1 Introduction

The vision of model-driven 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 that are specified and realized with models. Since large parts of an application may therefore be constructed from prefabricated pieces, 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 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 components. 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.

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 [22] in which 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 requires a verification that a system of components as a whole will behave correctly.

⋆ This work has been partially funded by the German BMBF project MDTS (01AK933) on Model-Driven Development of Telecommunication Systems.

[1] Email: grossh@iese.fhg.de
[2] Email: ina@cs.tu-berlin.de
schieferdecker@fokus.fraunhofer.de
[3] Email: din@fokus.fraunhofer.de

Built-in contract testing is based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" dynamically at deployment time will fulfill their contract. Consideration of built-in test artifacts begins early in the design phase as soon as the overall architecture of a system is developed and/or the interfaces of components are modeled. Built-in test can be and should be derived from the system model and should be developed in an integrated manner together with the system design, system model and system implementation.

In the literature built-in testing typically refers to all concepts that are added to a component's code for facilitating testing, or checking assertions and conditions at run-time. Built-in testing is not a new concept. Assertions were among the first built-in testing concepts in software engineering, although one cannot really call them built-in testing. An assertion is a Boolean expression that defines necessary conditions for the correct execution of an object or a component [12,22]. Assertions in software follow the same principles as the self-checking facilities that are readily built into hardware components. Whenever the Boolean expression of an assertion is executed it checks whether the value of some monitored variable is within its expected domain.

In the literature assertions and built-in testing are often used as synonyms, e.g. [22], however we believe that they are fundamentally different concepts: assertions are lacking one important ingredient for representing built-in testing, which is the notion of a test or a test case. Assertions can be used in a test and provide valuable information for error detection during test execution, but one cannot say that an assertion or its execution represents a test. A test is an experiment under controlled conditions that applies a set of test cases in a test procedure or test framework; so that we can only talk about built-in testing if we have test cases as integral part of components. Built-in testing strategies comprise the built-in self-test metaphor whose idea is derived from the self-test capabilities commonly built into hardware components. Built-in self test components are software modules that comprise their own test cases. In sloppy terms we could say that these are software components coming with their own tests for checking their own implementation.

One motivation for building self-tests into a software component is to check differing variations of component implementations that are all based on a single component specification [10,11]. In other words, a component is not merely viewed as its implementation, or the physical thing that we will deploy in our system, but it is viewed as the collection of all descriptive documents that fully describe the component's interfaces, structure and behavior, plus an arbitrary implementation of this model. The implementation self-test strategy is a typical component development time testing approach. It means it is a way to test individual units. It is not so suitable for checking component interactions in an assembly of individual units. Object technology provides the right tools for the implementation of this type of object testing. We can have a base class in Java, for instance, that includes the implementation of the functionality and then extend that with testing code through inheritance. The testing code of the extended class provides the test cases that will check the implementation of the base class. The extension will also provide some interface that can be used to invoke the test. If we would like to test such a component, we can simply instantiate its testable version, and start the tests through the additional testing interface. If we would like to integrate the class in an existing component infrastructure, we may simply instantiate the original version, the base class. During the test, the extended class can access the original functionality through the inheritance mechanism as long as the class does not define any private items that are not inherited. The advantage of this type of built-in self test is that we can break the encapsulation boundary of the tested object, but yet keep functionality and testing entirely separate in different classes. The fact that we can break the encapsulation boundary for testing can be seen both as a cure and curse. It is a cure because we can apply highly implementation specific test cases. After all, we can access all internal attributes and this result in high observability and controllability of the tested object. At a first glance this is very good for testing. However, breaking the encapsulation boundary in that way also is a curse, because if we apply such highly implementation dependent test cases we will never be able to reuse the tests in a different implementation. For instance, we might access distinct attributes in a test, which do simply not exist in another implementation, so that we can actually scrap the test.

Another similar approach with a slightly different motivation is to add component self tests and leave them permanently in an object or component implementation for reuse [7,8,9]. The main idea behind this strategy is to exploit the extension mechanism of typical object technologies to inherit not only the functionality of a class to its sub-classes but additionally its testing functionality in form of readily built-in test cases. The test cases are invoked through an additional testing interface that the object provides. Users of an object may access its normal interface and get its specified functional behavior in normal mode, but they can additionally access the object's testing interface and execute its built-in test cases in test mode. At first sight, it might seem the same as the previous approach but there is a subtle difference. In the previous case, the approach was simply to have unit tests attached and detached to and from a component implementation in a convenient way. Here, we have the tests always built into an object, and they are inherited from the base class into all extended classes. So, we have the same built-in testing facilities that the base class provides in all inherited classes. The main motivation of this approach is that software components or objects will get the same self test capability that can be regarded as a standard in most hardware components. Additionally, in contrast to hardware components, software components may inherit their features, and thus provide the same self-test capabilities in subsequent versions. However, software components differ from hardware components in one important respect. Whereas hardware components are built from materials that can physically degrade over time, software components are not. Software components are encoded in digital formats which can easily be checked (and if necessary corrected) to ensure that there is no change over time. Thus, the concept of a self-test in software similar to the hardware approach [8,9] is not directly applicable or useful in component-based software testing. This type of built-in testing is only useful in component evolution and maintenance because only such activities will in fact change the code of the component, and make a regression test necessary. There is no point in a component rerunning previously executed tests on itself, because by definition the component itself does not change. What can and usually does change, however, is the environment in which a component finds itself. The objective of built-in testing should therefore not be a test of a component's own functionality, because we can check that through typical unit regression testing, but an assessment of the component's environment and how well it interacts with that. In other words we have to assess that the component's environment into which it will be deployed does not deviate from what the component was developed to expect, and that the component does not deviate from what its new environment was developed to expect. Built-in contract testing addresses this.

So far, the principles of built-in tests have been developed [8,11]. However, there model-based design, specification and execution has not yet been considered to a large extend. In particular in the realm of MDA (model-driven architectures), the modeling of built-in tests as an integral part of component and system development needs to be addressed. With the development of UML 2.0 [20] together with the UML 2.0 Testing Profile [21,27] he technological base for model-based component and test development has been created. This paper will specifically discuss the use of UML for the model-based approach towards built-in tests.

The paper describes at first the principal concepts of built-in tests and proceeds with the generation of built-in tests from system models. Subsequently, the specification of built-in tests and their execution are discussed. An example demonstrates the application of model-based built-in tests. The paper concludes with an outlook.

2 Model-Based Built-In Tests

Meyer 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 [22]. 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 pair wise 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 reconfiguration is performed.

2.1 Built-in Tester Components

Configuration involves the creation of individual pair wise 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. 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 client ship connection between them).

In order to fulfill its obligations towards its own clients, a component that acquires a new server must verify the server's semantic compliance to its client ship contract. It means the client must check that the server provides the service that the client has been developed to expect. The client is therefore augmented with built-in test software in form of a tester component. This is called a server tester component, and 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 server tester component. If the test fails, the tester component may raise a contract testing exception and point the application programmer to the location of failure.

A test involves the invocation of the methods of an associated component with predefined input values and the checking of the returned results against the expected results. The input data and expected results are referred to as a test case. Under the object paradigm, a test case also often not only involves the checking of the results of the method invocations but also the checking of the correctness of the state transitions according to the external states. A test suite for a server tester component therefore contains a number of test cases that are developed according to distinct testing criteria, for example the coverage of the state transition model or the coverage of the functional specification. These are typically augmented with tests according to equivalence-class partitioning and boundary value analysis [1,5,6].

A client that owns a tester component and performs a contract test on its acquired server is called testing client or a testing component [13].

2.2 Build-in Testing Interfaces

The object-oriented and as a consequence the component-based development paradigm builds on the principles of abstract data types, which advocate to the combination of data and functionality in 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 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 properly. The contract testing paradigm is therefore based on the principle that components should expose their logical or externally visible (as opposed to internal) states by extending the normal functional server.

A testing interface provides additional operations that read from and write to internal state attributes that collectively determine the logical states. These auxiliary interface operations are usually derived through typical assertion checking techniques [1,4], although for contract testing they are much more formally defined and applied, since they essentially become part of a component's normal functionality.

A testing interface augments the functionality of the tested server with state checking and setting operations. The state checking operations verify whether the component is currently residing in a distinct logical state (for verifying the post conditions of a test case). The state setting operations set the component's internal attributes to represent a distinct logical state (for satisfying the preconditions of a test case). State checking operations are more fundamental than state setting operations. The latter may often involve quite considerable development effort. Thus, in most cases state setting will be achieved by invoking the operations of the normal functional interface. Subsequently, the state checking methods may be used to verify that the preconditions (initial state) for a test case are satisfied.

Within the server tester component a test case may be applied in two alternative ways. In the first way the state setting operations, if applicable, are invoked to ensure the preconditions required for a test case, the tested operation is invoked with the predetermined input parameters according to the testing criterion, and finally, the state checking operations are invoked to verify the post conditions required for a test case. The second way is applied when no state setting operations are provided by the tested component. Then, the operations of the component's normal functional interface have to be invoked to bring the component into the desired initial state for a test. Since these operations are part of the software that should be tested, the state checking operations have then to be invoked to verify the correct precondition for the application of a test case. Finally, the test method is called, and the state checking operations are used to verify the post conditions against the expected outcome.

A component that provides a testing interface and that is tested by a contract tester component is called testable server or a testable component [13].

2.3 Built-in Test Components

The distinction between clients and servers is only intended to refer to the roles that can be played in a pair wise interaction between two components. When viewed from a global perspective, individual components can, and usually do, play the role of both clients and servers. Any of the client/server relationships of components may be subject to contract tests. In the server role, a component provides a testing interface that supports the tests performed by its client's tester components, and in the client role, the component owns tester components that use the testing interfaces of its associated servers.

A component that plays both roles (i.e. provides a testing interface to its clients and contains its own tester components to test its servers) is called Built-in Test (or BIT) component.

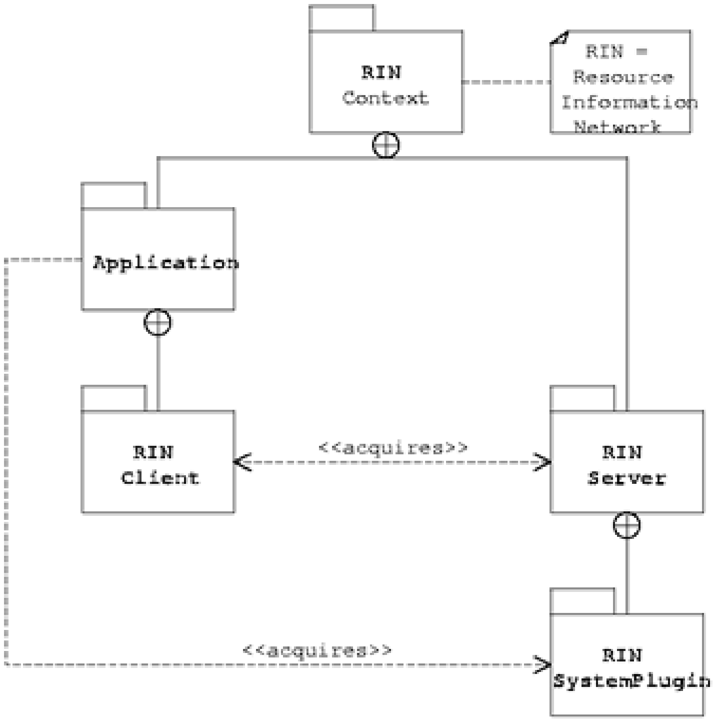

Fig. 1. General architecture of built-in testing.

2.4 Built-in Test Architecture

Figure 1 displays the principle of the contract testing architecture. Each tested component provides a testing interface that extends the original component (shaded in the figure), and it provides testing operations that associated client tester components may use to support the testing. Each testing component (client) owns a server tester component. This contains tests that check the server's compliance to its contract with the client.

3 Generation of Built-In Tests from UML

Testing that is based on the UML has many concepts in common with traditional code-based testing techniques. Source code can be seen as a concrete representation of a system, or parts thereof, and UML models are more abstract representations of the same system. More concrete representations contain more and more detailed information about the workings of a system. It can be compared with zooming in on the considered artifacts, generating a finer grained representation but gradually loosing the overview on the entire system. Less concrete representations contain less information about details but show more of the entire system. This can be compared with zooming out to a coarser grained level of representation making it easier to overview the entire system but loosing the details out of sight. The advantage of using model-based development techniques and the UML for development and testing is that a system may be represented entirely through one single notation over all levels of detail, that goes from very high level and abstract representations of the system showing only its main parts and most fundamental functions, down to the most concrete possible levels of abstraction similar and very close to source code representations. It means that in a development project we are only concerned with removing the generality in our descriptive documents without having to move between and ensure consistency among different notations. The same is true when testing is considered.

Code-based testing is concerned with identifying test scenarios that satisfy given code coverage criteria, and exactly the same concepts can be applied to more abstract representations of that code, i.e. the UML models. In that respect we can certainly also have model coverage criteria for testing. In other words, more abstract representations of a system lead to more abstract test artifacts, and more concrete representations lead to more concrete test artifacts of that system. Therefore, in the same way in that we are removing the generality of our representations in order to receive finer grained levels of detail and eventually our final source code representation of the system, in parallel we have to remove the generality of the testing artifacts for that system and move progressively towards finer grained levels of testing detail. Also, the system models in UML have to be accompanied by test models in UML, for which purpose the UML Testing Profile [21,27] has been developed.

3.1 White-Box Coverage Criteria and the UML

Coverage is an old and fundamental concept in software testing. Coverage criteria in testing are used, based on the assumption that only the execution of a faulty piece of code may exhibit the fault in terms of a malfunction or a deviation from what is expected. If the faulty section is never executed in a test it is unlikely to be identified through testing, so program path testing techniques, for example, are amongst the oldest software testing and test case generation concepts [30] in software development projects. This idea of coverage has led to quite a number of structural testing techniques over the years that are primarily based upon program flow-graphs [2] such as branch coverage, predicate coverage, or DU-path-coverage to name only a few. These traditional coverage criteria all have in common that they are based on documents (i.e. flow graphs, source code) very close to the implementation level.

Traditionally, these coverage criteria are only applied at the unit level which sees the tested module as a white box for which its implementation is known and available to the tester. On a higher level, in an integration test, the individual modules are only treated as black boxes for which no internal knowledge is assumed. An integration test is traditionally typically performed on the outermost sub-system that incorporates all the individually tested units, so that we assume white-box knowledge of that outermost sub-component, but not of the integrated individual units. Traditional developments only separate between these two levels: white box test in unit testing, and black box test in integration testing. Additionally, there may be an acceptance test of the entire system driven by the highest-level requirements.

More modern recursive and component-based development approaches do not advocate this strict separation since individual units may be regarded as sub-systems in their own right, i.e. components for which no internal knowledge is available, or integrating sub-systems, i.e. also components, for which internal knowledge may be readily available. Particularly in component-based developments where we cannot really strictly separate units from sub-systems both approaches may be readily applied in parallel according to whether only black-box information, e.g. external visible functionality and behavior, or additionally white-box information, e.g. internal functionality and behavior, are available.

Typical white-box strategies comprise statement coverage or node coverage on the lowest level of abstraction. In this instance, test cases may only be developed when the concrete implementation is available (i.e. for statement coverage), or if at least the implementing algorithm is known in form of a flow-chart (i.e. for node coverage). Statement coverage is typically not feasible, or practical with the UML, unless we produce a model that directly maps to source code statements, but node coverage may be practical if it is based on a low-level UML activity diagram. Activity diagrams are very similar to traditional flow-charts, although activity diagrams may also represent collaboration between entities (i.e. through so-called swim lanes). Other coverage criteria such as decision coverage, condition coverage, or path coverage, may also be applicable under the UML but it always depends on the type and level of information that we can extract from the model.

3.2 Black-box Testing Techniques and the UML

Most functional test-case generation techniques are based upon domain analysis and partitioning. Domain analysis replaces or supplements the common heuristic method for checking extreme values and limit values of inputs [3].

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 of the input space belong to the domain of interest. A domain may map to equivalent functionality or behavior, for instance. Domain analysis is used for and sometimes also referred to as partitioning testing, and most functional test case generation techniques are based on that. Equivalence partitioning, for example, is one technique out of this group that divides the set of all possible inputs into equivalence classes. This equivalence relation defines the properties for which input sets are belonging to the same partition.

Traditionally, this technique is only concerned with input value domains but with the advent of object technology it can be extended to behavioral equivalence classes. UML behavioral models such as state charts for example, provide a good basis for such a behavioral equivalence analysis, i.e. test case design concentrates on differences or similarities in externally visible behavior that is defined through the state model.

3.3 Test Specification with the UML Testing Profile

Since the majority of tests are developed manually as automated test derivation techniques still bear several limitations and/or are extended and enhanced manually, their separate specification is advantageous. Out of this motivation, the OMG has initiated the development of a UML testing profile that is specifically addressing typical testing concepts in model-based development.

The UML testing profile is an extension of UML 2.0 being based upon the UML metamodel. It defines a modeling language for visualizing, specifying, analyzing, constructing and documenting the artifacts of a test system. The testing profile particularly supports the specification and modeling of software testing infrastructures. It follows the same fundamental principles of UML in that it provides concepts for the structural aspects of testing such as the definition of test components, test contexts and test system interfaces, and behavioral aspects of testing such as the definition of test procedures, test setup, execution and evaluation. The core UML may be used to model and describe testing functionality since test software development can be seen as any other development for functional software properties. However, as software testing is based on a number of special test-related concepts these are provided by the testing profile as extensions to UML. The concepts are mainly grouped into concepts for test architecture, test behavior and test data.

A test architecture specifies the structural aspects of a test system and includes:

- The System Under Test (SUT), where one or more objects within a test specification can be identified as the SUT.
- Test components, which are defined as objects within the test system that can communicate with the SUT or other components to realize the test behavior.
- A means for evaluating test results derived by different objects within the test system in order to determine an overall verdict for a test case or test suite. This evaluation process is called arbitration. Users can either use the default arbitration scheme of the profile (i.e. the classical functional arbitration, where negative results have priority over positive results), or define their own arbitration scheme using an arbitration test component.

Test behaviors specify the actions and evaluations necessary to check the test objective, which describes what should be tested. For example, UML interaction diagrams, state machines and activity diagrams can be used to define test stimuli, observations from the SUT, test control/invocations, coordination and actions. However, when such behaviors are specified as tests the prime focus is given to the definition of normative or expected behaviors.

The handling of unexpected messages is achieved through the specification of defaults providing the means to define more complete, yet abstract test models. This simplifies validation and improves the readability of test models. The separate behavior of defaults is triggered if an event is observed that is not explicitly handled by the main test case behavior. The partitioning between the main test behavior and the default behavior is up to the designer. Within the testing profile default behaviors are applied to static behavioral structures. For example, defaults can be applied to combined fragments (within interactions), state machines, states, and regions.

The testing profile introduces further concepts that are necessary for test behavior specification such as:

- A test case, which is an operation of a test suite specifying how a set of cooperating test components interact with the SUT to realize a test objective.
- A verdict is a predefined enumeration specifying possible test results e.g. pass, inconclusive, fail, and error.
- A validation action, which can be performed by the local test component to denote that the arbiter is informed of a local test result.
- A log action, which is used to log entries during the execution for further analysis.
- A finish action to denote the completion of the test case behavior of a component, without terminating the component.

Another important aspect of test specification is the use of wildcards in test data. For example, pattern matching and regular expressions are very useful when specifying behavior for handling unexpected events, or events containing many different values. Therefore, the UML testing profile introduces wildcards allowing the specification of: (1) any value, denoting any value out of a set of possible values, and (2) any or omitted values, denoting any value or the lack of a value (in the case where multiplicities range from 0 upwards).

These concepts provide the capabilities required to construct precise test specifications and to develop systems and tests in an integrated manner using UML 2.0. This also applies to built-in tests as they require the same concepts as for traditional black-box tests.

4 Execution of Built-In Tests

If the built-in tests have been generated and specified in the UML 2.0 Testing Profile (U2TP), their executable versions can be generated with mappings towards existing test execution environments:

(i) JUnit is an open source unit testing framework, which is widely used by developers who implement unit tests in Java. The mapping primarily focuses on the JUnit framework. For instance, when no trivial mapping exists to the JUnit framework, existing JUnit extensions such as for repeated test case runs or for active test cases can be used.

(ii) TTCN-3 (Testing and Test Control Notation [14,28,25]) is widely accepted as a standard for test system development in the telecommunication and data communication area. TTCN-3 is a test specification and implementation language to define test procedures for black-box testing of distributed systems. Although TTCN-3 was one basis for the development of the testing profile, they differ in some aspects, but the UML testing profile specifications can be represented by TTCN-3 modules and executed on TTCN-3 test platforms.

Still, U2TP and TTCN-3 are on different levels of abstractions: TTCN-3 is on a detailed test case specification level, i.e. on a level from which executable tests can directly be derived. U2TP can also be used on more abstract levels by defining just the principal constituents of e.g. a test objective or of a test case without giving all the details needed to execute the tests. While this is of great advantage in the test design process, additional means have to be taken in order to generate executable tests. For example, the expressiveness of UML 2.0 sequence diagrams allows to describe a whole set of test cases by just one diagram, so that test generation methods have to be applied in order to derive these tests from such diagrams.

As TTCN-3 is a powerful option to execute the built-in tests and having an own TTCN-3 test platform [29], we have chosen to use it. TTCN-3 is built from a set of base testing concepts, which makes TTCN-3 quite universal and application independent. A TTCN-3 test specification consists of different parts including type definitions for test data structures, templates definitions for concrete test data, function and test case definitions for test behavior, and control definitions for the execution of test cases. The semantics of these concepts is well-defined and the main principles of test execution is defined in form of TTCN-3's test execution interfaces [15,16].

The constituents of a TTCN-3 test specification are derived from the U2TP built-in test specification including e.g. test type and data definitions and the definition of test behavior functions and test cases. A built-in test component is equipped with the component-specific executable versions of the TTCN-3 tests together with the access to a TTCN-3 runtime environment for test execution.

5 An Example

This section elaborates with an example the individual parts and steps towards model-based built-in tests. The RIN system [23], which supports working floor maintenance staff in their everyday working tasks, is taken as an example. This communication system hosts multiple communication devices that are interconnected through a radio network and controlled and supported by a number of desktop working places. The desktop working places help the maintenance staff to achieve their tasks and provide additional information. They can guide a worker through complex tasks by looking at the video signals from the workers video facility, give advice to the worker through the audio device, and provide additional information, for example the download of user manuals or video-based repair guides. Each of the communication devices has defined capabilities that are made public to all the other devices through the Resource Information Network.

Every device that is part of the network will have a RIN client, a RIN server and a number of RIN plug-ins installed. The server controls the resource plug-ins and communicates with the client of a connected other device. The client gets the information from associated device's RIN server. All the devices within the range of the communication system can, before they communicate, retrieve information from their associated nodes as to which things they are capable of at the moment. In that way, the individual nodes are never overloaded with data that they cannot process as expected. For ex-

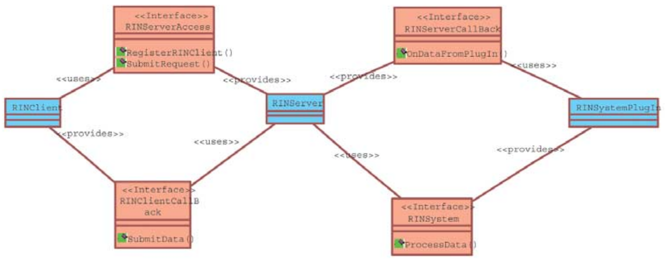

Fig. 2. UML-style representation of the RIN system components.

ample, the video application of a desktop station may determine the current memory state of a handheld device and decide according to that information whether it can send colored frames or only frames in black and white, it may also reduce the frame rate, or ask the handheld device to remove some of its unused applications from its memory. These decisions depend on the profile of the user and the priority of the applications that use the resource information network.

As shown in Figure 2 the RIN system consists of three components: RINClient, RINServer and RINSystemPlugin.

- RINClient: This component needs host system information like "system memory state" or "system power state". It connects to the RINServer component for such requests. For this connection RINClient uses ("uses") the provided ("provides") RINServer's interface RINServerAccess. The component RINClient provides also an interface RINClientCallBack. This interface is used from the RINServer in order to transmit required information to the RINClient back.
- RINServer: This component has to exist on each computer; it receives the client requests (via RINServerAccess) and transmits the client requests to the appropriate plug-in component RINSystemPlugin, which provides the interface RINSystem for receiving client requests. Via the interface RINServerCallBack, the RINServer receives the necessary information from the appropriate plug-in.
- RINSystemPlugin: This component runs on each computer and provides different system information to RINServers. The RINServer uses the RINSystemPlugins's interface RINSystem in order to transmit the RINClient's request. After the RINSystemPlugin component has collected all required information, it returns all data via the interface RINServerCallBack provided by the RINServer to the RINServer.

The RIN system provides the backbone for a highly flexible communication

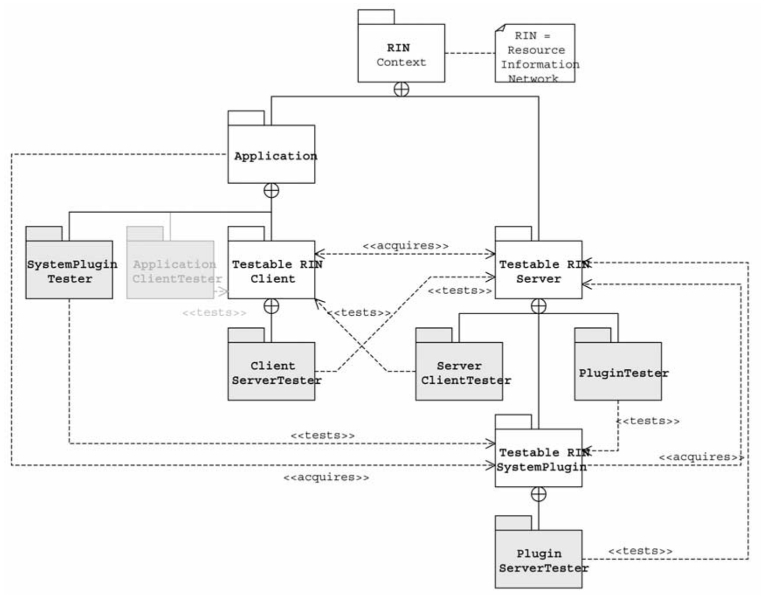

Fig. 3. RIN system architecture inlcuding the built-in test architecture (shaded boxes).

system that can handle many heterogeneous mobile devices. It means that the individual devices may be based on the full variety of available platforms, with differences in hardware, operating systems and applications. The usage of the system is only restricted through the applications that incorporate its services. It means that in the same networked application quite a number of different platforms may be used on which the RIN system is operating. The RIN system must therefore be tested on each platform and configuration. This is a platform test that checks the correct connection to the underlying middleware platform. An additional feature is the provision of different system plug-ins according to how a device is equipped with hardware. For testing it means that different types of plug-in components may be added on different platforms. And they must all abide by the contract that the server is implementing for communicating with these plug-ins. These tests are all situated on a lower-level of abstraction, the network or communication level. An additional application-level test assures that the application retrieves the right information from the respective plug-ins. That is, a number of high-level requests that are sent from the application to the plug-in (via the server). Each application will be augmented with a tester component for each plug-in that it is accessing. All tests are solely based on the contract testing approach. Figure 3 displays the organization of the RIN system with built-in contract testing.

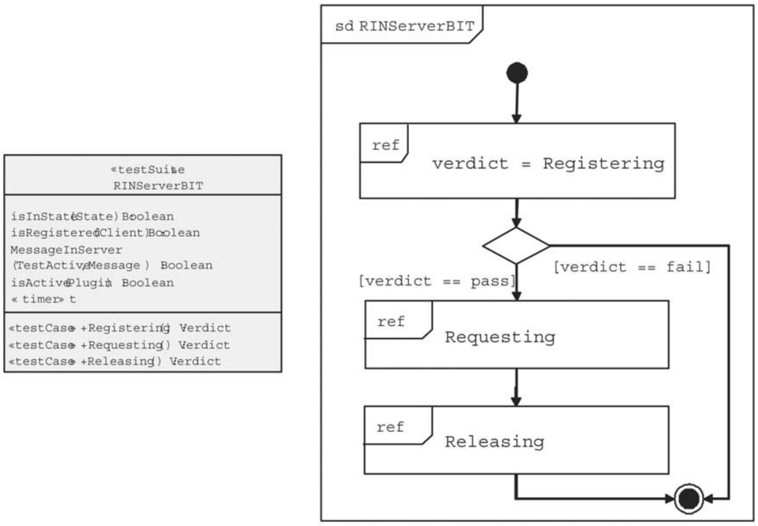

Fig. 4. Model of the built-in tests for the RIN server.

Test cases for the server cover for example the registration of clients (test case Registering), requests or "bypass" requests from clients (test case Requesting), or releasing the resources (test case Releasing). The set of test cases, i.e. the test suite, for the RIN server together with the execution order of these tests is depicted in Figure 4 The test case Requesting and Releasing is only performed if the test case Registering has been successful. The details of the Requesting test case are shown in Figure 5. It is derived from the behavioral model of the RIN server given as a state chart.

This test is translated to TTCN-3 and results in a simplified form (without giving test type and data information) in

```
external function
validClient() return Client;

altstep Default()
runs on RINClient {
   [] testPort.getreply {setverdict(fail); stop}
   [] catch.timeout {setverdict(fail); stop;}
}

testcase Registering()
runs on RINClient system RINServer
{
   activate(Default());
   // check the precondition
   testPort.call(IsInState(waiting),t) {
   [] testPort.getreply(IsInState(-): true)
   {setverdict(pass)}
   [] testPort.getreply(IsInState(-): false)
   {setverdict(inconc}); stop}
   [] catch.timeout {setverdict(inconc}); stop;}
   }
```

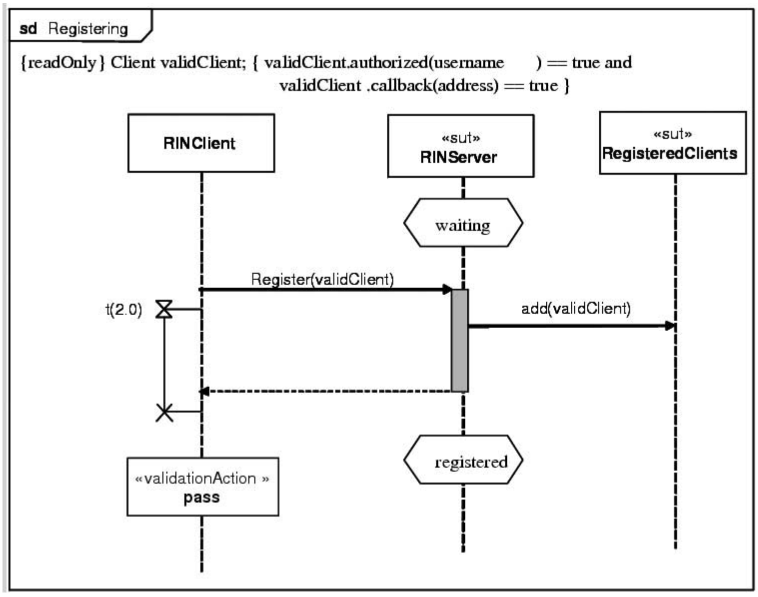

Fig. 5. UML-style sequence model for the example test case *Registering*.

```
    // main test body
    testPort.call(Register(validClient),t) {
    [] testPort.getreply(Register(validClient))
    {setverdict(pass)}
    }
    // check the result
    testPort.call(IsRegistered(validClient),t) {
    [] testPort.getreply(IsRegistered(-): true)
    {setverdict(pass)}
    }
    // check the postcondition
    testPort.call(IsInState(registered),t) {
    [] testPort.getreply}(IsInState(-): true)
    {setverdict(pass)}
    }
}
```

The external function `validClient()` is used to retrieve the information about a valid client from the environment. The test case `Registering` can be executed by the `Client` built-in test component (runs on) and tests the `RINServer` component (system). It initially activates a default to handle all unexpected or no responses from the server. The test initially checks the precondition for the tests and the proceeds with the main body by trying to register the valid client. Afterwards, the result is checked: is the client really registered and is the `RINServer` in state `registered`. All valid executions of the tests result in a `pass` verdict. Invalid executions lead to `fail`. If the precondition for the test is not fulfilled an `inconclusive` will be returned.

The tests in TTCN-3 are detailed enough to generate executable code, in

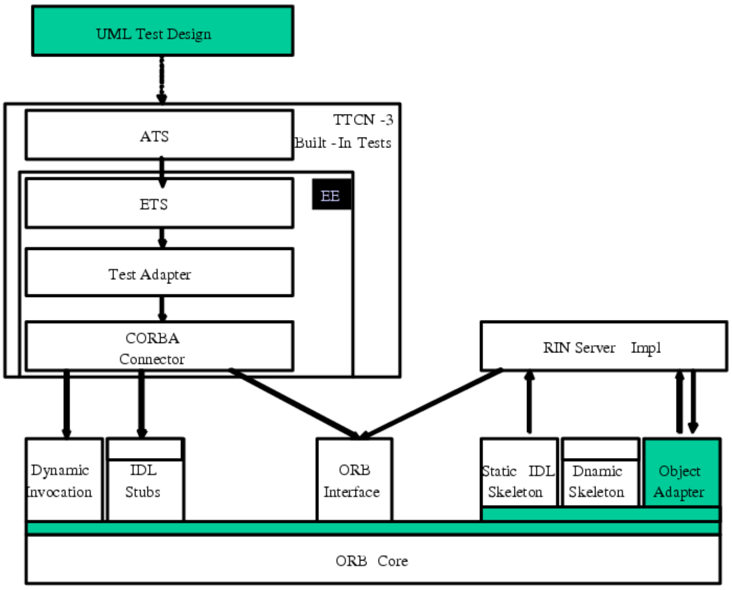

Fig. 6. Test execution architecture.

our case in Java. The compiled Java byte code is deployed and executed in a TTCN-3 execution environment.

The test execution architecture is shown in Figure 6. To be able to connect the test components with the target components, a test adaptor according to the TTCN-3 execution interfaces TRI [15] and TCI [16,26] is needed. The Test Adapter uses so called Connectors to mediate between different communication middleware (CORBA, CCM, RMI, Siena, UDP, etc.), which make the adapter generic and reusable for built-in test components in various execution context. The adapter implements the abstract operations of the test system (connect, send, receive, call, getreply, etc.). For the RIN system, the CORBA connector is used to invoke the RIN server methods.

6 Summary

This paper presents concepts and approaches towards the model-based development of built-in tests in UML by using the UML Testing Profile and an automated execution of these tests with the Testing and Test Control Notation TTCN-3 and respective TTCN-3 tools and runtime environment. Currently, the test generation from system models in UML as well as the mapping from UML Testing Profile specifications to TTCN-3 is done manually.

Future work will consider the further automation of this process by providing tool support for the test generation and translation. Also, further case studies will be elaborated in order to validate the expressiveness of the UML Testing Profile and its application for built-in tests.

References

[1] R.S. Pressman: Software Engineering, a practioner's approach. McGraw-Hill, New York, 1997.

[2] Beizer, B.: Software Testing Techniques. Von Nostrand Reinhold, N.Y., 1990.

[3] B. Beizer: Black-box Testing, Techniques for Functional Testing of Software and Systems. John Wiley & Sons, New York, 1995.

[4] I. Somerville: Software Engineering. Addison-Wesley, Reading MA, 1995.

[5] R. Binder: Testing Object-Oriented Systems: Models, Patterns and Tools. Addison-Wesley, 2000.

[6] Atkinson, C., et. al.: Component-based Product Line Engineering with UML. Addison-Wesley, London, 2002.

[7] Y. Wang, G. King, I. Court, M. Ross, G. Staples: On Testable Object-Oriented Programming. ACM Software Engineering Notes, vol. 22(4), 1997.

[8] Y. Wang, D. Patel, G. King, S.Patel: BIT: A Method for Built-in Tests in Object-Oriented Programming. Chap. 47 in Handbook of Object Technology, Zamir (ed.), CRC Press, 1998.

[9] Y.Wang, G. King, H. Wickburg: A Method for Built-in Tests in Component-based Software Maintenance. Proc. of IEEE Intl Conf. on Software Maintenance and Reengineering (CSMR-99), pp. 186-189, 1999.

[10] D. Deveaux, J.-M. Jzquel, Y. Le Traon: Reliable Objects: Lightweight Testing for OO Languages. IEEE Software, July/August 2001.

[11] Y. Le Traon, D. Deveaux, J.-M. Jzquel: Self-testable components: from pragmatic tests to design for testability methodology. Proc. of Technology of Object-Oriented Languages and Systems, Nancy, France, June 1999.

[12] D.S. Rosenblum: A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, vol. 21(1), pp. 19-31, Jan. 1995.

[13] Component+ Consortium: Built-in Testing for Component-based Development. Technical Report D.3, www.component-plus.org, 2001.

[14] European Telecommunication Standards Institute: The Testing and Test Control Notation: Core Language. ES 201 873-1, V.2.2.1, Oct. 2002.

[15] European Telecommunication Standards Institute: The TTCN-3 Run-time Interface (TRI). ES 201 873-5, V.1.0, Oct. 2002.

[16] European Telecommunication Standards Institute: The TTCN-3 Control Interfaces (TCI). ES 201 873-6, V.1.0, Mar. 2003.

[17] Gross, H.-G., Atkinson, C., Barbier, F., Belloir, N., and Bruel, M.: Built-in Contract Testing for Component-Based Development. In: Business Component-Based Software Engineering, Barbier (Ed.), Kluwer, 2003.

[18] Gross, H.-G., Atkinson, C., and Barbier, F.: Component Integration Through Built-in Contract Testing. In: Component-based Software Quality, Cechich, Piattini, Vallcillo (Eds.), Lecture Notes in Computer Science, Vol. 2693, Springer, Heidelberg, 2003.

[19] Gross, H.-G., and Mayer, N.F.: Built-in Contract Testing in Component Integration Testing. Electronic Notes in Theoretical Computer Science, 82(6), 2003.

[20] Object Management Group (OMG): ADTF, 2nd revised submission on Unified Modeling Language: Superstructure. Version 2.0, ad/03-02-01, 2003.

[21] Object Management Group (OMG): ADTF: The UML 2.0 Testing Profile, ad/03-03-26, July 2003.

[22] Meyer, B.: Object-oriented Software Construction. Prentice-Hall, Upper Saddle River, 1999.

[23] Fraunhofer IGD: RIN System Specification. Darmstadt, Germany, http://www.igd.fraunhofer.de.

[24] Santos, P., Ritter, T., Born, M.: Rapid Engineering of Collaborative and Adaptive Multimedia Systems on Top of CORBA Components. Irmscher, K. (ed.), Kommunikation in Verteilten Systemen, VDE, Offenbach, 2003.

[25] Schieferdecker, I.: System-Level Tests with TTCN-3. Proc. Of World Conference on Integrated Design and Process Technology, IDPT, Austin, Texas, 2003.

[26] Schieferdecker, I., Vassiliou-Gioles, T.: Realizing distributed TTCN-3 test systems with TCI, IFIP 15th Intern. Conf. on Testing Communicating Systems, TestCom 2003, Cannes, France, May 2003, Best Paper Award.

[27] Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The UML 2.0 Testing Profile and its Relation to TTCN-3, IFIP 15th Intern. Conf. on Testing Communicating Systems, TestCom 2003, Cannes, France, May 2003.

[28] Grabowski, J., Hogrefe, D., Rethy G., Schieferdecker, I., Wiles, A., Willcock, C.: An Introduction into the Testing and Test Control Notation (TTCN-3). Computer Networks Journal, Vol.42, Issue 3, 2003.

[29] Testing Technologies: TTCN-3 Tool Series. http://www.testing-technologies.de/products.

[30] Warner, C.D.: Evaluation of Program Testing. IBM Data Systems Division Development Laboratories, Poughkeepsie, N.Y, 1964.