Modeling and Implementation of Built-In Contract Tests

Hans-Gerhard Gross[1], Ina Schieferdecker[2], and George Din[2]

[1] Fraunhofer IESE
Kaiserslautern, Germany
grossh@iese.fhg.de
[2] Fraunhofer FOKUS
Berlin, Germany
{schieferdecker|din}@fokus.fhg.de

Summary. Built-in contract testing is based on the idea of building tests directly into components, so that each component can assess whether it will be integrated into a suitable environment, and the environment can assess whether a newly integrated component will be acceptable.

Component technologies such as CCM, .NET, or EJB are more and more being supported by model-based approaches like the OMG's Model Driven Architecture (MDA). The idea is to gain considerable momentum in the development of component-based architectures through high-level modeling and automatic code generation. However, the emphasis with these approaches is currently more on system design and development and not as much on system validation and testing, so the expected reductions in component-based development in terms of time and effort can only be realized in component and application construction and not during test development. Lengthy and costly in-situ verification and acceptance testing, that is mainly still performed manually, undermines the benefits of these modern development approaches.

This chapter demonstrates how built-in contract testing can be integrated with and made to supplement automatic approaches to derive application testing from system models, represent them on the model level, and generate executable tests from these models. This model-based testing approach increases the degree of automation in generating and realizing built-in contract tests; therefore, it also increases the quality of the built-in tests and reduces the resources required for developing them.

1 Introduction

The vision of modern recursive model-driven development approaches is for applications to be specified and designed at high levels of abstraction through graphical representations, for available components to be identified and assembled on this abstract level, and for the missing links or the adaptors be-

tween the components to be easily derived from the models, to a large extent through automatic code generation [57]. This allows 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 and to integrate them easily, almost in a plug-and-play fashion. Although there has been considerable recent effort to put this vision into reality, it has been almost exclusively dedicated to the model-based design and implementation of the functional software rather than the testing. If test software development and testing will not be supported in the same way as is the case with functional software, we will in fact create a gap in the overall development life cycle, since functional development will be readily supported largely by sophisticated automatic tools but testing will not be. In other words, applications that are becoming increasingly more complex will be developed with increasing speed, but their assessment and validation will continue to be performed traditionally, and will actually slow down the momentum gained in the first stage. This will drastically increase the validation effort of a typical project instead of decreasing it, or developers will simply reduce the overall testing activity. In our opinion, both scenarios are unacceptable.

Built-in contract testing [155–157] is a step in the right direction to integrate the efforts for functional and test development. The philosophy of the technology is parallel modeling and implementation of the functional software together with the test software, and the derivation of the test models from the functional models. Since the test software is built directly into the functional software, it merely represents an additional development effort that is performed almost in exactly the same way as any other software development for a system's original functionality (rather than its testing).

With the principles of built-in contract testing, we already have a powerful means for modeling and implementing test software in tandem with the functional software. However, testing is based on a number of very specific concepts that are not typically needed for a system's normal functionality. For model-based test development, we can now apply the recently released UML testing profile [315] that is put forward by the Object Management Group (OMG) and represents a very distinct extension to the core UML. Additionally, for implementation and execution of the test software, we can resort to the Testing and Test Control Notation (TTCN-3) [114–116] that is traditionally used in the telecommunication domain, although it is also suitable for any software domain. This realizes a generic way of specifying the test software and transforming it into a concrete implementation automatically.

This chapter presents an introduction to each of the technologies, and illustrates how they can supplement each other and be integrated into an overall model-based test development process for component-oriented applications. We believe the integration of these technologies represents a simple yet powerful way of realizing test design and development in the spirit of the MDA. In addition to the introduction of the technologies, we show how such

a model-driven test development process may be applied to a subcomponent of a large communication platform.

2 Built-In Contract Testing

The correct functioning of a system of components at runtime 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 the set of rules governing the interaction of a pair of objects (and, thus, components) is typically referred to as a contract [277]. This characterizes the relationship between a component and its clients as a contract, a formal agreement, 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 (the server in a client/server relationship), or requiring a service (the client in a client/server relationship). Testing the correct functioning of individual client/server interactions against their specified contracts therefore represents a validation that a system of components as a whole will behave correctly.

The ideas and the technology behind built-in contract testing [155–157] have been mainly developed within the European Union funded project Component+ [71], and the technology focuses on verifying these pairwise client/server interactions between two components when an application is assembled and integrated. It is based on the notion of building contract tests into components so that they can validate that the servers into which they are "plugged dynamically", at deployment time, will fulfill their contracts. 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 artifacts can, and should, be derived from the system model, and they should be developed in an integrated process, together and in parallel with the system design, the system model, and the system implementation.

2.1 Built-In Tester Components and Testing Interfaces

The configuration of a system involves the creation of individual pairwise client/server relations between the components in that system. This is usually done by an outside "third party," which we refer to as the context of the components. It creates the instances of the client and the server, and passes the reference of the server to the client (thereby establishing the client connection between them). In order to fulfill its obligations toward its own clients, a component that acquires a new server must verify the server's semantic compliance to its client contract. This means that 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 the 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 the failure. The client's tester component will be realized according to the specification of the client's required interface.

The object-oriented and, as a consequence, the component-based development paradigms build on the principles of abstract data types, which advocate the combination of data and functionality in a single entity. The compliance of the component's externally visible states and transitions to the expected states and transitions of the specification state model must therefore be checked. These externally visible or logical 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 external test software of a component cannot therefore get or set any internal state information. 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 interface of a component through a so-called testing interface. A testing interface provides additional operations that read from and write to internal state attributes that collectively determine the logical states.

2.2 Basic Model of Built-In Contract Testing

The distinction between clients and servers in the previous section is intended only to refer to the roles that can be played in a pairwise interaction between two components. When viewed from a global perspective, individual components can, and usually do, play the role of both client and server. 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 [71].

Figure 1 displays the organization of the contract testing architecture (in the shaded box). Each tested component, or each server role, provides a testing interface that extends the original component, and provides testing operations that associated client tester components may use for the testing. The testing interfaces enhance a server's testability. The testing interface comprises typical operations for state information, or assertion checking. The chapter 'COTS

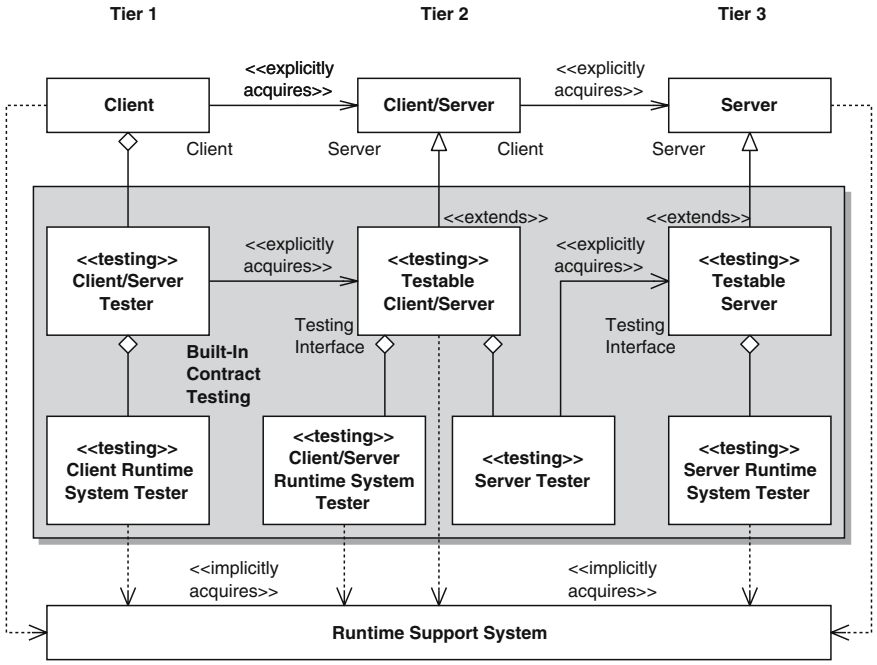

Fig. 1. Basic model of built-in contract testing in a typical 3-tier architecture.

Component Testing through Built-In Test' provides more detailed information on how testing interfaces may be designed and implemented; in this chapter, we focus more on the implementation and automatic generation of a testing architecture that is based on built-in contract testing, and on the design and generic realization of tester components.

Each testing component, or each client role, owns a server tester component that contains test cases. The test cases check the server's semantic compliance to its contract with the client. The tests in the client's tester component are defined according to the specification of the client, and not of the server. They represent the client's expectations of the tested server, and can be derived according to any existing test case generation technique, e.g., traditional coverage criteria and functional testing techniques, or, more specifically, according to model coverage criteria as proposed in [154].

In addition to testing the explicit components in a component framework, built-in contract testing addresses also the testing of the implicit server components that are provided by the underlying component platform, by the runtime system, or by the operating system. These components are implicitly acquired servers because they are available to be used by the explicit component framework of the developed application. Implicit and explicit servers represent the entire import interface of a component.

3 Generation of Built-In Contract 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 detailed information about the workings of a system. It can be compared to zooming in on the artifacts considered, generating a finer grained representation, but gradually losing the overview of 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 losing the details out of sight. The advantage of using models such as the UML for the development as well as the testing is that a system may be represented entirely through one single notation over all levels of detail. It means that in a development project we are concerned only with removing the generality in our descriptive documents without having to move between and ensure consistency among different notations.

Therefore, in the same way in which 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, we have to in parallel remove the generality of the testing artifacts for that system and move progressively towards finer grained levels of testing detail. So, the system models in UML have to be accompanied by test models in UML, and the UML Testing Profile [314,315] has been defined in order to support the development of the testing models in UML explicitly.

3.1 Black- and White-Box Coverage Criteria and the UML

Coverage is an old and fundamental concept in software testing. Coverage criteria in testing are based on the assumption that only the execution of a faulty piece of code or a faulty specification 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 path coverage testing techniques, for example, are among the oldest software testing and test case generation concepts [418] in software development projects. The idea of coverage has led to quite a number of functional and structural testing techniques over the years that are primarily based upon flow graphs and specification pieces (e.g., function points) [24]. These traditional testing criteria all have in common that they are often based on documents (i.e., flow graphs, source code) very close to the implementation level. Traditionally, these coverage criteria are applied only 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 performed on the outermost subsystem that incorporates all the individually tested units, so that we assume white-box knowledge of that outermost subcomponent, but not of the integrated individual units. Traditional development separates only 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 subsystems in their own right, i.e., components for which no internal knowledge is available, or integrating subsystems, i.e., components for which internal knowledge may be readily available. Particularly, in component-based development where we cannot strictly separate units from subsystems, 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. Here, we can also apply all the traditional testing criteria, although on different levels of abstraction. The difference is that, at the beginning of a development project, models will provide only part of the information that is required for testing; but, since we are progressively moving our project toward more concrete representations, we will eventually be able to also extract more concrete information for the testing.

3.2 Test Specification with the UML Testing Profile

Since the majority of tests are developed manually as automated test derivation techniques, they still bear several limitations (or they are extended and enhanced manually), their separate specification and realization is advantageous. From this motivation, the Object Management Group has initiated the development of a UML testing profile that is specifically addressing typical testing concepts in model-based development [315]. The UML testing profile is an extension of the UML 2.0 that is also based on the UML meta-model [314]. It defines a modeling language for visualizing, specifying, analyzing, constructing, and documenting the artifacts of a test system that can be developed in parallel with the original functional system. The testing profile particularly supports the specification and modeling of software testing infrastructures. It follows the same fundamental principles of the 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 the 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 concepts are provided by the testing profile as extensions to the UML. The concepts are grouped

mainly into 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, 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 a 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 that are necessary to check the test objective, which describes what should be tested. For example, UML interaction, state, 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 primary focus is given to the definition of normal 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 the designer's responsibility. 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 also introduces 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, which 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, which is used 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 that contain many different values. The UML testing profile introduces wildcards for

dealing with that, and permits the specification of "any value," denoting any value out of a set of possible values, and "any or omitted values," denoting any value or the lack of a value (in the case where multiplicities range from 0 upward). All these concepts provide the capabilities that are required to construct precise test specifications and to develop systems and tests in an integrated way in parallel by using the UML. This also applies to built-in tests, as they require the same concepts, as well as to traditional black-box tests.

4 Implementation and Execution of Built-in Tests

If the built-in tests have been defined in terms of what the UML testing profile is suggesting, their executable versions can be generated with mappings according to existing test execution environments, e.g.,

- JUnit is an open source unit testing framework, which is widely used by developers who implement unit tests in Java. The mapping focuses primarily 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. It is important to note that built-in contract tests are not unit tests of the same tenor. A unit test is defined according to the specification of the tested unit, whereas a contract test is defined according to the specification of the client's usage profile of the tested unit. In other words, a unit test is performed according to the component developer's mind, whereas a contract test is performed according to the component customer's mind.
- TTCN-3 (Testing and Test Control Notation [114]) is a widely accepted standard for test system development in the telecommunication and data communication domains. TTCN-3 is a test specification and implementation language that can be used to define test procedures for black-box testing of distributed systems. Although TTCN-3 was a basis for the development of the testing profile, they differ in some respects; but the UML testing profile specifications can be represented by TTCN-3 modules and executed on TTCN-3 test platforms. TTCN-3 is the chosen platform throughout the rest of this chapter.

Although the testing profile and TTCN-3 are used on different levels of abstraction, with TTCN-3 on a more detailed and concrete test case specification level and the testing profile on a more abstract level, TTCN-3 can also be used on more abstract levels by defining just the principal constituents of a test objective, or of a test case, for example, without giving all the details needed to execute the tests. While this is of great advantage in the test design process, since it hides too much detail we have to add that missing information later on in order to generate executable tests. For example, the expressiveness of UML 2.0 sequence diagrams allows us to describe a whole set of test

cases with just one diagram, so test generation methods have to be applied in order to derive these tests from the diagrams. TTCN-3 represents a powerful way to manage and execute built-in contract tests, and it comes with its own tool suite [390]. It is built from a set of basic testing concepts that make it quite universal, and independent of any particular application or domain. A TTCN-3 test specification consists of different parts including type definitions for test data structures, template 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 are defined in the form of TTCN-3's test execution interfaces [115, 116]. All the required parts of a TTCN-3 test specification are derived from the built-in test models in the testing profile, including, e.g., test type and data definitions and the definition of test behavior functions and test cases. A built-in tester component is then implemented through component-specific executable versions of the automatically generated TTCN-3 tests, together with access to a TTCN-3 runtime environment for test execution. This sounds bulky, but it represents a very elegant way of managing the built-in contract tests in an abstract form and instantiating them automatically for any required component platform.

5 A Working Example

This section shows, by means of an example system, how the individual parts in the previously described testing process are put together, and it elaborates the steps that we have to take in order derive executable tests from abstract models that have been developed according to the philosophy of built-in contract testing. The example is a Resource Information Network (RIN) from Fraunhofer IGD in Darmstadt, Germany [193, 357], one part of a larger communication system that supports working floor maintenance staff in their everyday working tasks. The large communication system hosts multiple communication devices that are interconnected through a radio network and controlled and supported by a number of desktop workplaces. The desktop workplaces help the maintenance staff accomplish their tasks, and provide additional information. They can guide a worker through complex tasks by looking at the video signals from the worker's video facility, giving advice to the worker through the audio device, and providing additional information, for example, online user manuals or video repair guides. Each of the communication devices has defined capabilities that are made public to all the other devices through the RIN. 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 clients of connected devices. The client gets the information from the 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

through the RIN about what things they are capable of doing at a given time. This way, the individual nodes are never overloaded with data that they cannot process as expected. For example, the video application of a desktop station may determine the current memory state of a handheld device, and decide based on that information whether it can send colored frames or only black-and-white frames; it may also reduce the frame rate, or ask the handheld device to momentarily 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 RIN. As shown in Fig. 2 the RIN system represents a 3-tier architecture with: `RINClient`, `RINServer`, and `RINSystemPlugin`.

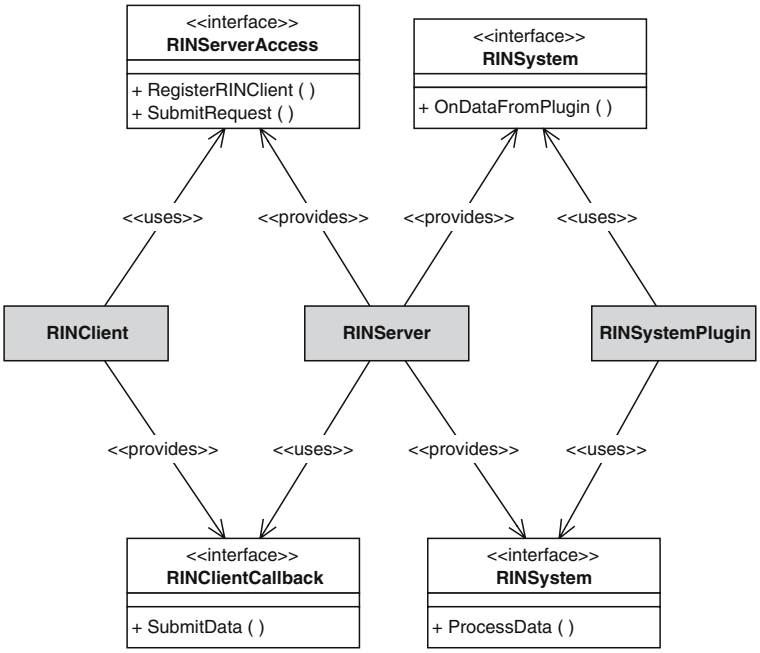

Fig. 2. UML-style structural organization of the Resource Information Network.

- `RINClient` requires system information, such as "system memory state" or "system power state," from an associated host. It connects to the host's `RINServer` for such requests. For this, `RINClient` uses the `RINServer`'s provided interface, `RINServerAccess`. The `RINClient` provides also an interface `RINClientCallBack`. This interface is used by `RINServer` in order to re-transmit required information to `RINClient`.
- `RINServer` resides on each host, receives the client's requests (via `RINServerAccess`), and transmits it to the appropriate `RINSystemPlugin`. This provides the interface `RINSystem` for receiving client requests and returning responses to the server through `RINServerCallBack`.

- `RINSystemPlugin` resides in each `RINserver`, and provides different system information. After `RINSystemPlugin` has collected all required information, it returns all data via the server's `RINServerCallBack` interface.

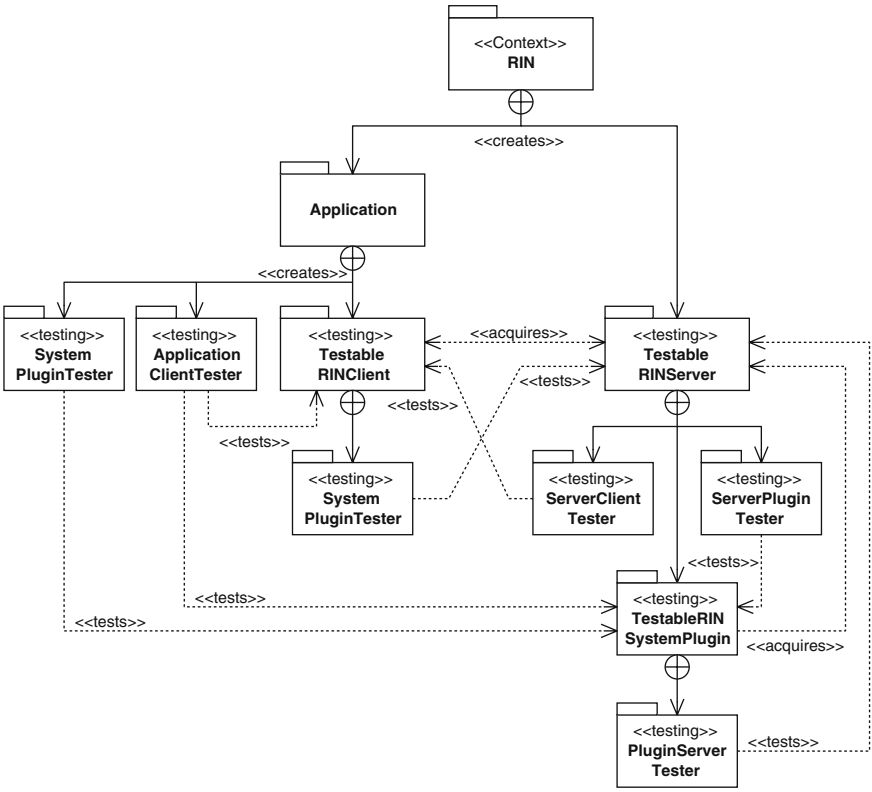

Fig. 3. Containment of the built-in contract testing architecture for the RIN system.

Since the integrating communication application is supposed to execute on quite a different number of hosts, the RIN system must be tested on each of these different platforms and configurations. This can be seen as a deployment or platform test that checks the correct connection to the underlying middleware platform. An additional feature is the provision of different system plug-ins, based on how a device that is equipped with hardware must be checked. For testing this means, that different types of plug-in components may be added to the servers on different platforms, and they must all abide by the contract that the server is implementing for communicating with the plug-ins. The previously described tests are all carried out on a lower level of abstraction, the network or communication level. An additional application-

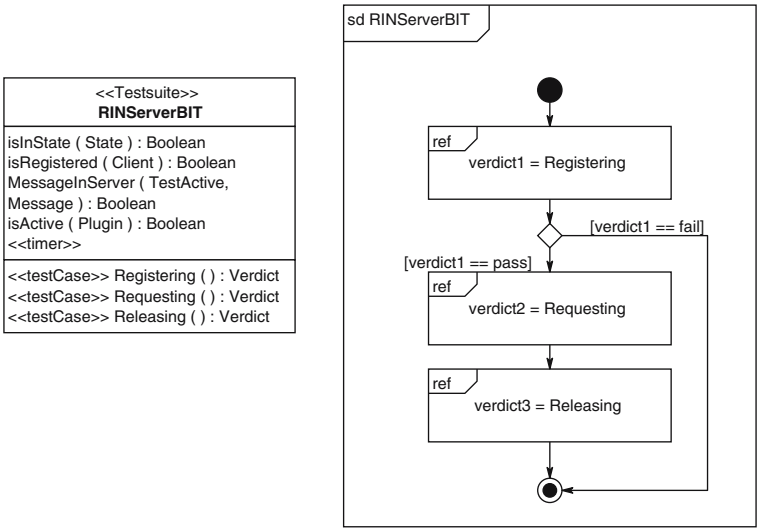

Fig. 4. Specification of the BIT server component with its testing behavior.

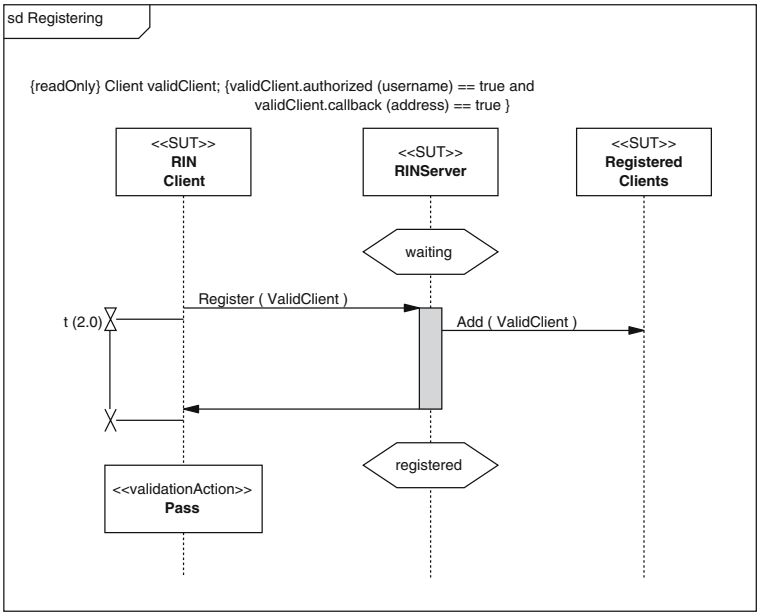

Fig. 5. Sequence diagram for the test case "Registering".

level test assures that the application (the large communication system) retrieves the right information from the respective plug-ins. This means that a number of high-level requests must be sent from the application to the plug-in (via the server) in order to assess its correctness. Each application will be augmented with a tester component for each plug-in that it is accessing. All tests are based solely on the contract testing approach. The model of the built-in contract testing architecture for this example is displayed in Fig. 3. For example, the test cases for the server cover the registration of clients (test case "Registering"), normal 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 Fig. 4. The test cases "Requesting" and "Releasing" are performed only if the test case "Registering" has been successful. The detailed behavior of the "Registering" test case is illustrated through the sequence diagram in Fig. 5. It is derived from the behavioral model of the RIN server given as a state chart.

The test is translated into TTCN-3. The next paragraph shows a simplified form of a TTCN-3 specification for this test (without giving test type and data information):

```
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 pre-condition
  testPort.call(IsInState(waiting),t) {
    [ ] testPort.getreply(IsInState(-): true)
    {setverdict(pass)}

    [ ] testPort.getreply(IsInState(-): false)
    {setverdict(inconc); stop}

    [ ] catch.timeout {setverdict(inconc); stop;}
  }

  // 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 post-condition
    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's built-in test component and test RINServer component. It initially activates a default test to handle all unexpected responses from the server. The test validates the pre-condition for the tests, and then proceeds with the main body by trying to register the valid client. Afterward, the result is checked: is the client really registered; and is the RINServer in the state `registered`? All valid executions of the tests will result in the verdict "pass." Invalid executions lead to "fail." If the pre-condition for the test is not fulfilled, "inconclusive" will be returned. The tests in TTCN-3 are detailed enough to generate executable code. For the example, we generate Java code. The compiled Java bytecode is deployed and executed in a TTCN-3 execution environment.

The test execution architecture of the TTCN-3 suite is displayed in Fig. 6. In order to connect the test components with the target components, a test adapter according to the TTCN-3 execution interfaces TRI [116] and TCI [115] is required. The test adapter uses so-called connectors to mediate between different communication middleware (e.g., CORBA, CCM, RMI, Siena, UDP, etc.), which make the adapter generic and reusable for built-in test components in various execution contexts. The adapter implements the abstract operations of the test system (e.g., `connect`, `send`, `receive`, `call`, `getreply`, etc.). For the RIN system, the CORBA connector is used to invoke the RIN server methods.

6 Summary and Conclusion

This chapter has presented and introduced an integrated model-driven development and test process based on novel and existing technologies: Built-In Contract Testing, the UML Testing Profile, and the Testing and Test Control Notation, in its third incarnation, TTCN-3. Built-in contract testing represents the overall framework for testing component-based applications. It provides guidelines on how to augment functional components with the capability

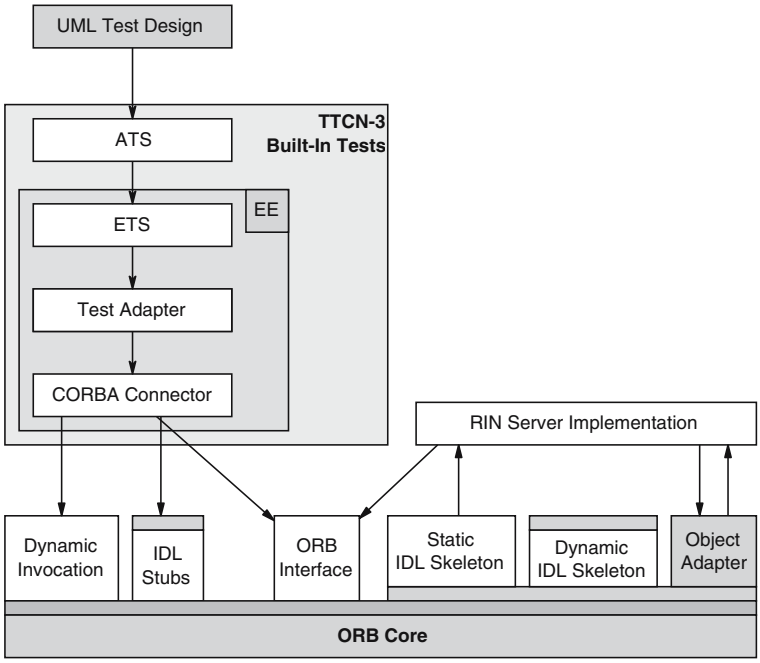

Fig. 6. Architecture of the TTCN-3 suite.

to test other associated server components and to be tested by other client components. For modeling the high-level built-in contract testing architecture, the core UML provides sufficient support. Because the testing architecture is not really focusing on concrete test concepts, such as test behavior, or test data. But if the specification of the testing artifacts becomes more concrete further down the design of a project, the UML testing profile provides all necessary concepts for the specification of the nitty-gritty testing details. In the next step, we have to implement the testing infrastructure, and this is the case mainly for test components and their incorporated test suites. Integration of the components is carried out according to the integration rules of the underlying component platform, e.g., through IDL-type interface mappings, or through component adapters that realize functional, behavioral and semantic mappings between two components. Since all the testing artifacts are readily built in, they are also part of these mappings, and they are treated as normal functionality. The TTCN-3 permits a generic implementation of the test components and their respective test cases out of the UML testing models, independently of any concrete underlying component platform or programming language. The "embodiment" into concrete source code is performed automatically. TTCN-3 supports a number of implementation technologies, such as Java, C++, and some newer component models.

Currently, the test generation from system models in UML and the mapping from the UML testing profile specifications into TTCN-3 are still performed manually because of a lack of appropriate tools. From TTCN-3 onward, everything is automatic. Future work must attempt to increase the automation throughout the entire process, especially in the early modeling phases and for the extraction of tests from models. The development and provision of suitable tools for the test generation and translation is therefore one of the most important future activities. Another important route is the assessment and improvement of the UML testing profile, through more case studies, in order to validate its expressiveness and improve its application for built-in contract testing. Overall, we believe that the topics described in this chapter, and, of course, their integration, can make a significant contribution to component-based system development.

Acknowledgment

This work has been partially funded by the German National Department of Education and Research (BMBF) under the MDTS project acronym. The project deals with model-driven development and testing methods for telecommunication systems.