# UML Framework for Automated Generation of Component-Based Test Systems

Marc Born, Ina Schieferdecker and Mang Li

*GMD FOKUS*
*Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany*
*phone: +49 30 3463-7000, fax: +49 30 3463-8000*

*{born, schieferdecker, m.li}@fokus.gmd.de*
*www.fokus.gmd.de*

**Abstract**

*Component-based technologies are recently used to implement complex distributed systems. Testing of these systems becomes even more complicated. New test methods need to be investigated. Components of a system under test communicate via well-defined interfaces. It is to validate the behavior of (a set of) components at their interfaces. The paper proposes to build a test system itself from test components, which interact with the components under test via their interfaces. For the development of such test systems, a UML framework is used which contains base types for test components with generic behavior for the setup and configuration of a test system and for the exchange of coordination messages between test components. The paper discusses the aspect of automatic code generation of test components from UML specifications of test systems. An example of using the test framework is given.*

**Keywords**

*Test framework, UML, code generation, components-based systems, distributed testing*

## 1. Introduction

Component-based technologies are considered to be adequate for distributed applications operating in heterogeneous environments. A component is in general a replaceable part of a system. Emerging component frameworks, for example, CORBA, COM+, EJB are in their nature Object-Oriented (OO) technologies, which model components as objects that provide services at interfaces. Interfaces are specified by means of signature and interaction scenarios with context-dependencies, if needed.

Component-based technologies that lead to greater flexibility, generality and productivity bring new challenge but also new possibilities to testing distributed systems.

According to [11], the concurrent nature of distributed systems is the source of making their testing harder than testing sequential systems:

- The probe effect, which reflects the effect of changed behavior of a system when attempting to observe it, may occur.
- Racing conditions in concurrent activities may lead to non-reproducible behavior.
- A synchronized global clock has to be realized for observability of test events.

Both [4] and [11] propose a test methodology that tests components at first in isolation and then in combination. [12] discusses that a test system for a component-based system should be designed such that the test system mirrors the component structure of the system under test, i.e. there is a one-to-one mapping of components of the system under test to components of the test system.

Such a test architecture allows an immediate reflection of interaction scenarios of interfaces to be tested within the test component attached to this interface. In addition, this test architecture allows to distribute the test system itself in order to locate test components nearby the components under test. A distributed test system is of particular advantage for reducing the impact of transmission delays while controlling and observing the system under test (SUT) by the test system (TS). This is significant for delay-critical tests, e.g. performance tests.

Practical experiences of testing a TINA access session server [3] have shown that a number of concepts and mechanisms used for functional testing of a distributed test system are also applicable and re-usable for other types of testing such as performance and scalability testing. Generalization of basic concepts and mechanism for a wider class of test types lead to a set of generic test components. These generic (i.e. abstract) test components are the basis for developing dedicated TS consisting of a set of specific (i.e. concrete) test components by their specialization.

The set of generic test components in combination with rules and guidelines for their specialization constitute a so called *test framework* (TF) for distributed component-based systems. TF is the main subject of this paper.

Test development is a time- and resource-consuming activity. One of the most important aspect gained from generalizing the underlying concepts and mechanisms with generic test components in the TF is the increased efficiency in developing a concrete distributed test system. Only attention to the specific test objectives is required. The general things are inherited from the test framework.

A consistent, extensible and easy to use specification of the test framework is needed to make it practical. An adequate specification technique has to be selected. Unified Modeling Language (UML [6]) is a widely accepted modeling language for OO systems. In fact, UML combines a number of modern OO techniques, such as OMT, Statecharts, OCL (Object Constraint Language). The flexibility of UML is supported by extension mechanisms (e.g. stereotypes) and well-formedness rules (e.g. OCL).

A comprehensive introduction to UML from the testing perspective is given in [1]. It discusses generally the limitations of UML diagrams and possible extensions for adequate test models.

In comparison to this approach, our test framework targets the system or application level testing. It contains an architecture for component-based, distributed test systems. It also proposes to specify such test systems by structural and dynamic models using UML notations, in order to facilitate automated realization of test systems.

## 2. Foundations

### 2.1 Test architecture

The proposed test framework is based on a component-based test architecture. This architecture is intended for the testing of functional or operational capabilities of a component-based system under test (SUT). As shown in Figure 1, the SUT is considered to be an integration of service components (SCs), which must not be identical with the internal architecture of the SUT, but can be any logical grouping of system objects reasonable for system users. Parallel test components (PTCs) are aligned to the identified SCs. The coordination of PTCs during the test execution is supported by a main test component (MTC). PTCs and the MTC may be located on distributed nodes. The setup of distributed test components is provided by front ends (FEs). A test manager provides the control of front ends and test components directly or indirectly by human operators via a supervisory interface.

Such a test architecture is beneficial not only for the functional design of test components, but also for the modularity and scalability of the whole test system.

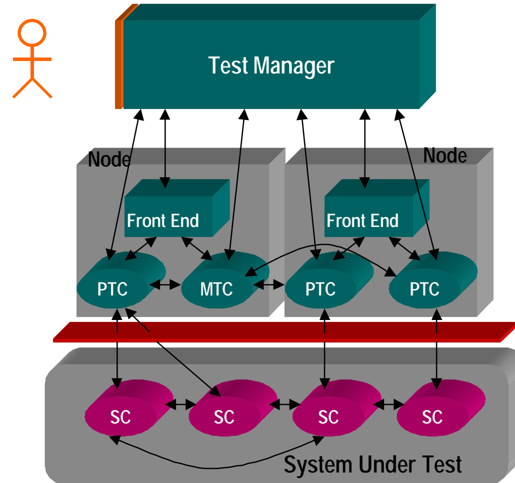


**Figure 1** Component-based test architecture

### 2.2 Separation of generic and user-defined test systems

The test framework generalizes also test case independent activities such as set up of test configurations, initiation of tests, coordination between test components and collection of test results. Thus, it is meaningful to separate a generic test system from user-defined test systems:

- *Generic test system* (GTS) is the generalization of the component-based test system. It contains test case independent, but test process typical behavior which can be shared between different test suites.
- A *user-defined test system* (UTS) is a specialization of the GTS. It adds aspects that are specific for the system under test (SUT). Since the SUT contains system specific descriptions used by the UTS, the SUT is also part of the test framework.

### 2.3 Specification Technique

A description technique for the test framework is desired to have the following properties:

- object-orientation to support the generalization-specialization aspect;
- technology independence to ensure that the framework can be realized with different technologies for different application domains;
- ability to cover structural as well as behavioral descriptions to allow adequate definition of test systems;
- customizability;
- support by tools to allow automated generation of executable test components.

The Unified Modeling Language (UML [6]) fulfils these requirements. UML places few restriction on the usage of the notation. It provides extension mechanisms such as stereotypes, tagged values and constraints. Due to the extension mechanisms, “fragmentary, incomplete, inconsistent, and ambiguous models are easily produced without violating any of the UML’s requirements” [1]. Thus, the selection of the proper notations is important.

We use UML class diagrams for the structural specification of GTS and UTS, where test system components are described by classes, interfaces and relationships. When SUT is specified in UML, static information can be immediately shared with UTS.

For modeling the dynamic aspect of OO systems, UML provides sequence, collaboration, statechart and activity diagrams. Sequence and collaboration diagrams visualize interactions between object instances, while statechart and activity diagrams are of interest when lifetime of objects is modeled[1]. Since the test framework targets functional and operational testing, where observable external behavior is essential, sequence diagrams are primarily used. Details of the test framework models are given in the following section.

## 3. The test framework in UML

### 3.1 Generic test system

The GTS is modeled by computational objects (COs). COs communicate via a set of interfaces, which consists of a set of operations and attributes. An interfaces can be declared together with a CO as *supported* in the sense that the CO provides services over it, or as *required* in case that it is provided by the

1. [1] gives a thorough discussion on statechart-based test design for class test. It is however not in the scope of this work.

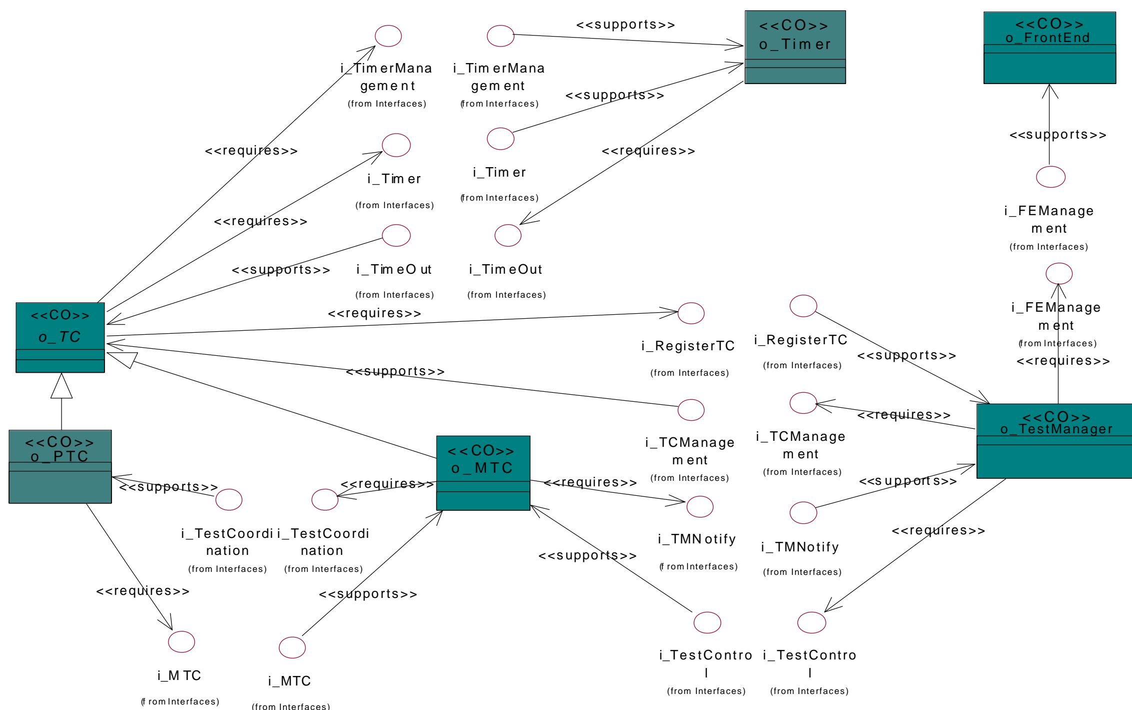


**Figure 2** Specification of the test Framework

CO's environment.

The following COs of the GTS reflect the general test architecture (see Figure 1):

**o_TestManager** represents the test manager. It provides the interface i_RegisterTC to o_TC for the registration of test components and the interface i_TMNotify to o_MTC for notifications from the MTC, e.g. submission of test results. It requires the interfaces i_FEManagement, i_TCManagement and i_TestControl.

**o_FrontEnd** represents front ends. It provides the interfaces i_FEManagement to o_TestManager for the control of front ends.

**o_TC** is an abstract base for test components. It supports the interfaces i_TimeOut to o_Timer for the indication of timeout, and i_TCManagement to o_TestManager for the lifecycle control of test components. It requires the interfaces i_TimeManagement, i_Timer and i_RegisterTC.

**o_MTC** is a specialization of o_TC and provides the functionality of the main test component (MTC). The interfaces i_TestControl and i_MTC are supported. i_TestControl allows o_TestManager the setup and termination of test cases. i_MTC is used by o_PTC for example to submit local test results. o_MTC requires the interfaces i_TMNotify and i_TestCoordination.

**o_PTC** inherits also from o_TC. It is the base class of parallel test components (PTCs). It provides o_MTC the interface i_TestCoordination and uses the i_MTC interface of o_MTC.

**o_Timer** is a timer control and management component. It allows o_TC to create timers over the i_TimerManagement interface and the start, reset and cancel of timers over the i_Timer interface.

The main flow of the GTS's behavior is represented by a set of sequence diagrams, where each of them documents a partial view of the entire functionality.

Each of the following sequence diagram represents one part of the test process:

- **Establish Configuration** fulfils the setup of a test configuration that consists of a main and one or more parallel test component(s). Test manager, front end and test component are the involved object instances (see Figure 3), represented by::o_TestManager, ::o_FrontEnd and ::o_TC. In this scenario, test manager calls startTC() at the i_FEManagement interface of front end, passing the name and the number of instance of the test components to be created, as well as the reference to the test manager's i_Register interface. After the instantiation, each test component registers itself with the test manager, upon which an identifier for the test component is returned.

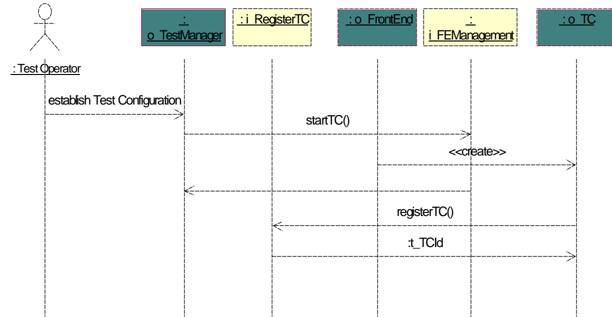


**Figure 3** Establish a configuration

- **Test Initiation** deals with the distribution of test case specific behavior and parameters via the test manager to test components. A test operator has the opportunity to select test cases and set parameters via the test manager's supervisory interface.
- **Generic Test Case** contains the common initial behavior of all test cases, which will be refined by user-defined test system. That is, the main test component requires the start of the test case on every parallel test components. **End Test** represents the termination of a test case under normal condition, while,
- **Release Configuration** is used when a test case is interrupted. In case that a test component does not respond, the test manager initiates the release of a test configuration.

The logic between individual scenarios is depicted by an activity diagram as shown in Figure 4. The activity diagram describes the order of test configuration establishment, test initiation and test case execution. Since the concrete test case is not known by the GTS, *Generic Test Case* serves as a placeholder that will be overloaded by a UTS. A test case may have four outcomes:

- PASS: It is the verdict when the test purpose is fulfilled;
- FAIL: It is assigned when the SUT does not behave as defined by the test purpose;
- INCONCLUSIVE: It indicates unsuccessful test execution due to test purpose independent reasons that are captured by the test case.
- Abort without verdict. In contrast to the previous three outcomes that yield orderly termination of the test case, the fourth outcome does not result in

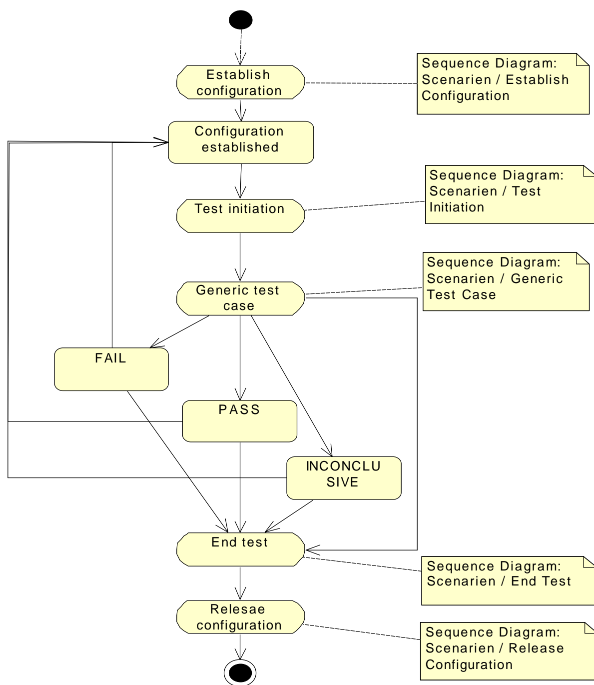


**Figure 4** Generic test activity model

a test verdict. Intervention via the supervisory interface or events that are not considered in the test case may lead to a shut down of test components by the test manager.

### 3.2 User-defined test system

A user-defined test system (UTS) refines the generic test system. First of all, the UTS's MTC and PTCs inherit the generic MTC and PTC of the GTS. The refinement focuses on the behavioral description. It may consist of test case independent and test case specific behavior. Further, if the UML model of the SUT is available, it is merged with the test framework model. In this manner, static information of the SUT can be shared with the UTS.

The major dynamic model of a UTS is again reflected in sequence diagrams for each test case. A test case sequence diagram involves SUT, PTC and timer instances.

The following section elaborates the generation of test system source code from sequence diagram descriptions of a UTS.

## 4. Code generation

### 4.1 Open issues with UML

Automated generation of UTSs require adequate specifications. The UML sequence diagram used for the behavioral description has restrictions to represent the following aspects:

- **Multiple instances**. There may be more than one instance of the same class taking part in a particular scenario.
- **References**. To perform the interaction between two instances there must be a way to obtain the reference of the target instance.
- **Computations/decisions** and **handling of variables**. For a particular sequence of operation calls, output of one operation may be used as input for the next operation. Some variables are required to hold the values of the corresponding messages.
- **Synchronization**. Distributed test components have to be synchronized.
- **Timing**. Especially for performance testing there is a need to model timing constraints for message invocations.

Most of the problems are already solved in the test framework. However, the synchronization of test components and the handling of multiple instances (for performance testing) need further study.

### 4.2 Example test case

We discuss subsequently possible solutions[1] for the problems introduced above, with an example test case. This test case is used for the functional testing of a TINA Retailer Reference Point (Ret-RP) implementation [10]. Ret-RP is the reference point between the business roles consumer and retailer. The purpose of the test case *checkRET_RP* is to test the initial interaction which originates from a consumer to establish an access session with the retailer by passing valid user identification. The test case can also be used for performance evaluation [3].

The SUT in this example is an implementation of the retailer domain Ret-RP. A PTC named Customer_PTC plays an active role in the test case, i.e. it controls and observes the behavior of the SUT. A timer object to support the test execution is also involved.

1. Although the example code is provided in C++, other languages can be also used for the implementation of test components.

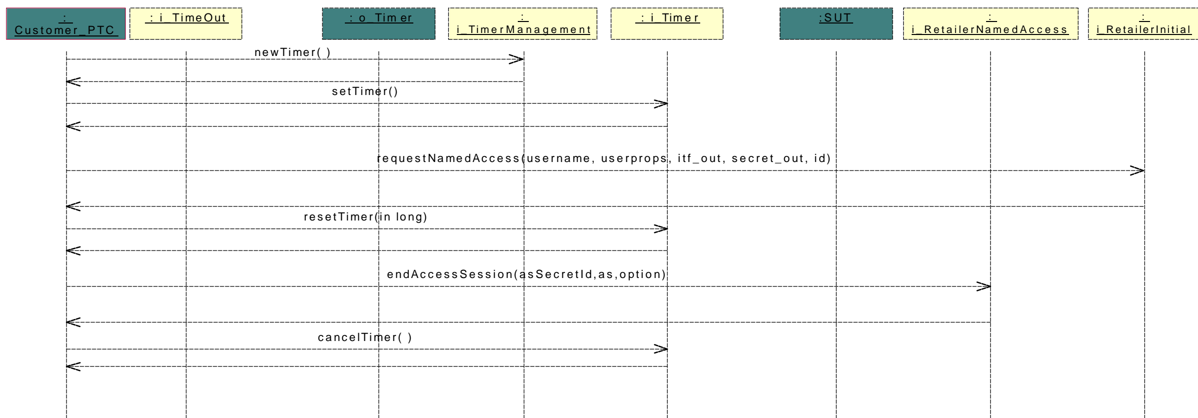


**Figure 5** Test case example checkRET_RP

The test case is described by a sequence diagram (Figure 5). It represents the following behavior:

- Customer_PTC creates a timer instance and sets the timer value, before
- a request on the operation requestNamedAccess() of the i_RetailerInitial interface of the SUT is sent;
- a reply of requestNamedAccess() is received by the Customer_PTC, which indicates the successful establishment of an access session (at this point in time, the test purpose is fulfilled);
- Customer_PTC resets the timer, and
- releases the access session by sending an endAccessSession() request;
- cancel of the timer terminates the test case.

### 4.3 Code structure

The general assumption we made for code generation is, that each test case corresponds to one sequence diagram. The sequence diagram represents the flow of events from the test components to the SUT and vice versa, which is assumed to be a valid flow, i.e. a flow that finally produces the verdict PASS.

Since each test component can be involved in more than one test case, the behavior of a test component with respect to a test case is defined in an own single method of the class, which implements the test component.

In the example, we have the class Customer_PTC and the test case *checkRET_RP*. The method checkRET_RP() has to be implemented for Customer_PTC in a way as defined by the sequence diagram.

The actual test case is determined during the setup phase of the framework. The test manager invokes the method setTestBehavior() of the i_TCMangement interface to determine the test case to be executed. After the setup has been finished, the test components know, which test case has to be performed and which method has to be invoked.

The behavior of the test case is started by using the beginTest() method of the i_TestControl interface provided by the test manager.

In the following part of this section it is shown, how the method checkRET_RP() can be generated to perform the behavior of the test case.

The initial code of the class looks as follows:

```
class Customer_PTC:public PTC{
public:
        CustomerPTC();
        ~CustomerPTC(){};
        void checkRET_RP();
        setVerdict(t_Verdict verdict);
}
```

### 4.4 Sequence of test events

The sequence of test events, which has to be performed in the method checkRET_RP(), is determined by the sequence diagram. The method requestNamedAccess() is invoked and after the result is received, endAccessSession() is called.

However, a number of problems (references, variables, exceptions) must be solved if the code should be automatically generated from the diagram. Subsequent sections 4.5, 4.6 and 4.7 will focus on these problems.

Although not included in the example test case, sometimes a test component has to wait for an external invocation before it can continue its main behavior. This is modelled by the operation waitFor() with the name of the method and the interface at which the

invocation is expected as parameters.

```
waitFor(„method_name“, “interface_name“);
```

The implementation of this operation will suspend the execution of the current thread until an invocation at an interface of the test component is received. In the implementation of the called method, the suspended thread will be activated. It then checks whether the expected method was called and continues its execution, or simply returns with the verdict FAIL if another method than the expected one has been invoked.

### 4.5 Exceptions

Each method which is invoked in the sequence of actions between test components and the SUT can cause exceptions. Some of the exceptions are defined as user exceptions in the signature specifications of the interfaces. Other exceptions are system exceptions which may be raised in case of errors in the underlying software, infrastructure or even hardware.

If a method does not answer with a normal reply as indicated by the sequence diagram but raises an exception, the test case cannot be completed successfully. It has to be terminated. The verdict depends on the kind of the exception:

- It is FAIL in case of a user exception.
- It is INCONCLUSIVE in case of a system exception.

If the test case should test whether the SUT raises the proper exception in a certain situation i.e. if the exception is the expected behavior, this has to be specified in the sequence diagram explicitly.

### 4.6 Interface references

In order to make a call to an interface of the SUT, a test component must obtain first a reference to the interface. The sequence diagram defines exactly, with which interfaces a test component communicates. These are in the example i_RetailerInitial and i_RetailerNamedAccess. For all those interfaces variables are defined to hold interface references. The name of a variable is either determined by the instance name in the sequence diagram (if any) or generated by default as <interface name>__Ref.

There are several possibilities to set values for the variables. They may be obtained via a lookup in the Name Service, or by using test case parameters or as result of a previous operation invocation. This information cannot be obtained directly from the sequence diagram. It is modeled by additional properties for each message in the diagram. It is possible to set the manner how a value is determined and additional parameters[1] if required.

### 4.7 Variables

Variables are used for different purposes in the implementation of a test component. They hold the values of parameters for invoked operations, serve for parameter passing from one operation to another, or may be used to compare expected and received results of calls. Variables are declared as attributes of the class specification to which they belong.

It is possible to use the declared variables (attributes of the UML class) as parameters of operation invocations in the sequence diagrams. This enables the automatic code generation. If values must be assigned to variables prior to operation invocations, and this can not be done as part of an initialisation specification, an additional mechanism is required. Since a sequence diagram by default does not provide a possibility to provide the needed code, we have extended the specification of messages and introduced additional properties preMessageCode and postMessageCode. Those properties may hold text values, which then contain the code for the variable assignment.

The preMessageCode for requestNamedAccess() looks as follows:

```
username = „user1“;
userprops.length(1);
userprops[0].name="password";
userprops[0].value <<= "";
```

Another possibility instead of using preMessageCode and postMessageCode would be to generate the code of a test component only partially and insert the additional code directly into the generated fragment.

### 4.8 Decisions

Sometimes the success of an operation depends on the returned value of the operation. Thus, a kind of decision possibility in the code is needed. This problem cannot be solved directly using sequence diagrams. The same mechanism as for variables is proposed. That is to use the properties preMessageCode and postMessageCode to hold the code for the decisions

1. For example in case of a Naming Service, a parameter is the name of the naming context.

to be made.

### 4.9 Timing

The sequence diagram for a test case uses an instance of the object o_Timer for the management of timers. At the interfaces of this object it is possible to get new timers, and to set, reset or cancel timers. If a timeout occurs, a message from the timer object to the i_TimeOut interface of the PTC that uses the timer is sent. Per definition, each PTC has such an interface instance for each running timer. If the method timeOut() is called when not expected, i.e. it was not specified in the sequence diagram, the method setVerdict() is invoked either with the verdict FAIL or INCONCLUSIVE. Which verdict is assigned, must be specified by the user. Therefore, it is possible to assign an additional property to the setTimer() message specification in the sequence diagram which holds the verdict as value.

If timeout is specified in the sequence diagram as a test event, then waiting for a timeout is implemented like any other event using the method waitFor().

The whole timer concept has been modeled as part of the test framework. Hence, the problem of timing is solved by explicit modeling of the timer object.

### 4.10 Implementation of the code generator

The code generator who's principles are discussed above was implemented in a plug-in technology for existing UML tools. The interface of an existing UML tool was used to get all information from the UML model of a UTS, which is needed to generate code for the UTS. Although the code generator itself is independent from a specific UML tool, we have used Rational ROSE to implement it. This tool provides a Microsoft COM-interface to access the UML model. It allows also to define additional properties for model elements.

## 5. Conclusions

In this paper a test framework has been specified that is based on an architecture that allows the control of distributed test components.

In this test framework, test purpose independent aspects are extracted by a generic test system (GTS), so that developers of user-defined test systems (UTSs) can concentrate on only those aspects that are specific for the system under tests (SUTs).

The intend of the test framework is to support automated generation of code for UTSs. The Unified Modelling Language (UML) is selected for the specification of GTS and UTSs. The static models of test systems are described by class diagrams, while sequence diagrams are mainly used for the behavioral specification.

Restrictions of UML sequence diagrams are discussed. Practical solutions facilitating code generation are presented. The test framework was applied to a simple test example.

We will continue the investigation of distributed test systems by using it for more complex cases, for example for performance evaluation of object-oriented systems.

## 6. References


[1] R. V. Binder, "Testing Object-Oriented Systems, Models, Patterns and Tools", Addison-Wesley, 1999.

[2] M. Benattou, L. Cacciari, R. Pasini, O. Rafiq, "Principles and Tools for Testing Open Distributed Systems", in Proc. of the 12th Intern. Workshop on Testing of Communicating Systems, Budapest, Hungary, Sept. 1999.

[3] M. Born, A. Hoffmann, I. Schieferdecker, Th. Vassiliou-Gioles, M. Winkler, "Performance Testing of a TINA Platform", in Proc. of TINA'99, Hawaii, USA, April 1999.

[4] U. Buy, C. Ghezzi, A. Orso, M. Pezze, and M. Valsasna, "A Framework for Testing Object-Oriented Components", Proc. of the First Intern. ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, U.S.A, May 1999.

[5] ETSI TC-MTS, "Methods for Testing and Specification (MTS), Test Synchronization, Architectural Reference, Test Synchronization Protocol 1 (TSP1) Specification", ETSI Tech-nical Report ETR 303, Sophia Antipolis, Jan. 1997.

[6] ITU-T X.903, ISO/IEC 10746-3, "Open Distributed Processing - Reference Model, Part 3", Geneva, Swiss, 1997.

[7] ITU-T Z.130, "Object Definition Language", Geneva, Swiss, Mar. 1999.

[8] OMG, "Common Object Request Broker Architecture (CORBA)", ver. 2.3, 1999.

[9] OMG, "Unified Modeling Language (UML)", ver. 1.3, 1999.

[10] TINA-C, "Ret Retailer Reference Point Specification", ver. 1.1, 1999.

[11] A. Ulrich, "Test Case Generation and Test Realization in Distributed Systems", PhD Thesis (in german only), Otto-von-Guericke-University Magdeburg, Germany, 1998.

[12] T. Walter, I. Schieferdecker, J. Grabowski: Test Architectures for Distributed Systems - State of the Art and Beyond (Invited Paper). - In Proc. of the 11th Intern. Workshop on Testing of Communicating Systems, Tomsk, Russia, Sept. 1998.