

CCM testing environment¹

A. Hoffmann, A. Rennoch, I. Schubert, A. Vouffo-Feudjio

Fraunhofer Fokus

<http://www.fokus.fhg.de/tip>

Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany

phone: +49 30 3463-7000, fax: +49 30 3463-8000,

email: a.hoffmann@fokus.fhg.de

Abstract

The introduction and usage of the CCM infrastructure implementations provide new possibilities to the developers of Corba-based applications. The OMG has standardized both a framework for an easy integration of business logic code within the Corba programming environment and the procedures to assemble and deploy several cooperating Corba components on a target networked system. We investigate in the analysis of the testing issues which are due to the new technology.

Keywords: Testing, Corba components, CCM, Test component, TTCN3

1 Introduction

The component-based approach for software engineering is gaining increasing acceptance due to its benefits like maintainability, reusability, flexibility and productivity. Component based programming puts emphasis on the implementation of business logics in contrast to the details of programming constraints and therefore enables the development of distributed applications possibly involving heterogeneous components or subsystems, e.g. components or subsystems provided by different vendors.

The Corba Component Model (CCM) [1] from the Object Management Group (OMG) gives further opportunities to software developers since CCM is a component model based on the widely-established and programming language independent Common Object Request Broker Architecture (CORBA) [2]. The programming language and platform independence of CCM

1. This work is partially supported by the European IST Project Coach (IST-2001-34445, Component Based Open Source Architecture for Distributed Telecom Applications, <http://www.ist-coach.org>).

could be seen as its major advantages in comparison with other component models like EJB² and COM models [11].

With the growing amount of ongoing work on component-based technologies, the issue of providing appropriate testing methods and architectures for software based on that approach becomes more and more essential. In fact, as stated by Weyuker [20] and others ([17], [18], [19]) component technology raises fully new issues w.r.t testing which should absolutely be considered to ensure reliability.

While some publications discussed the theoretical aspects of that problem e.g Test adequacy [16] or integration tests [17], most of the few implementations on that area either focus on unit testing of single components (e.g [10]) or imply limitations due to dependence on a specific programming language (e.g SmallTalk [18]).

In this paper we propose a test approach for component-based software, which uses CCM components but could be applied to other component models, too. We benefit from the use of the Testing and Test Control Notation (TTCN3) [4] for specifying tests in a platform-independent manner. TTCN3 is the successor of Tree and Table Combined Notation (TTCN2) and gives means for testing message-based communication between entities (suitable for protocol-testing), but also operation-based communication (essential for interface- and application-testing).

The papers starts in the following chapter 2 with an analysis of the potential testing issues. In chapter 3 we present the testing requirements and the approach to get and run test programs. Special attention to the application of the only standardized test notation TTCN3 [4] has been given in chapter 4. Finally there is a chapter 5 on proxy-based (passive) test components and some concluding remarks.

2 Testing targets

At the beginning of the work it is necessary to find the user groups who benefit from testing in the CCM context. There are at least three target groups: (a) the providers of the CCM infrastructure, (b) the application developers of CCM-based applications, and (c) the users of the CCM-based applications.

We see a lot of reasons to work on a test suite development project which verifies the conformance of a CCM implementation against its standardized requirements: Conformance testing has been accepted to be a means for improving a product's correctness and its interoperability. E.g, the VSOrb conformance test suite for Corba2.3 ORB implementations has helped to find various bugs in prominent Orb products [6]. Its scope covers syntax and semantics tests on IDL compiler, the ORB API, and the GIOP/IIOP. Unfortunately there is only a little return of invest for the test suite development costs due to the small group of commercial (non open source) ORB products.

A testing infrastructure for CCM-based applications is the other big challenge for the acceptance of the CCM technology. The provision of a CCM based test environment to the application developers can help to shorten the development time of application products. Application testing is possible at different stages of the application development:

2. CCM even provides interoperability with EJB.

- Testing may focus on a single component under test, and validate its executor programming code to be provided for the different component implementation variants (e.g. monolithic or locator-based) or its home executor. Here we have to take into account different types of interfaces, external or local (container-related) interfaces. This level is referred as unit testing, too.
- In the development process we may consider the interworking of two or more components communicating either via receptacle/facet or event source/sink (subsystem testing).
- Special emphasize should be given to the deployment facilities in CCM. We see a possibility to extend a CCM assembly package by test components to allow even end customers the operation of prepared application tests on his target environment (after deployment of the assembly).

For each of the testing level the (semi-automatic) generation/derivation of a test environment is a key issue to allow e.g. even customers an easy testing of a component/assembly package. The degree of the automation is obviously dependant on the volume and characteristics of the information available about the system under test (SUT). There is a list of heterogeneous specification parts about the SUT available which may be used to support the test developer in the test derivation procedure:

- IDL specification of the SUT components (IDL 3.0 defined by [2]),
- component implementation information on the units of the implementation (CIDL/PSDL)
- XML descriptors files:
 - software package descriptor (.csd)
 - Corba component descriptor (.ccd)
 - component assembly descriptor (.cad)
 - property file descriptor (.cpf)
- and related compiler output files

It has to be noted that not all pieces of information can be expected from a CCM application. E.g. the PSDL description is not supported by some CCM implementations. In the context of CCM application testing it is necessary to verify the core programming code which is provided by the application developer. We can distinguish the following categories of developer code:

- standardized callback operation provided for the CCM environment, e.g. `set_session_context`, `ccm_activate`, `ccm_passivate`, `ccm_remove`,
- component and home executor code, i.e. `MyComponent_impl`, `MyComonentHome_impl`, `MyEventType_impl`,
- entry point declaration of the component homes.

From the users point of view it appears valuable to verify the availability and semantic correctness of the external interfaces which comprise

- application defined operations, and
- CCM standardized operations (e.g. for navigation).

3 Approach

Our approach considers a testing requirement analysis for both, a general telecom domain component technology and the CCM based component technology. In [8], [9] the following three major demands have been stated for testing:

- a distributed test system including (semi-automatic) generated components as part of the tester,
- an appropriate, widely accepted, and executable test notation, and
- the integration of test infrastructure within the SUT configuration (and its package specification).

Futhermore there are two requirements with less priority (implementation is optional):

- the additional implementation of a component interface for test specific operations (e.g. to monitor internal states), and
- the consideration of network environment information (e.g. load measurements) to improve the interpretation of observations on SUT behaviour.

The collection of application (or specification) code or XML descriptors (or fragments only) can be used to derive parts or full test scenario programs (or at least test purposes) to become a portion of a test suite. A mapping of user provided definitions to the code skeletons is needed. It is envisaged that a set of standard tests, which are independent of the business logic of the CCM application can be derived from the available specifications (e.g. at the equivalent interface). The test scenarios which address the business logic code from the application have to be build on the developers know-how. This knowledge on the CCM application should be used to define the tests. Therefore an easy test environment has to be provided to enable a fast and simple test development and operation.

The motivation of the application developer to define the test business code depends on the amount of work (e.g. lines of test code) to be provided. Since the tests should be applicable in different programming environment it is better to define them not in the target programming language, but with a simple abstract description language. It is important that there are compilers to generate executable test programs for the customer execution platforms.

Sample CCM application test scenarios may address heterogeneous categories, like robustness tests or behaviour conformance tests. A possible test approach is to introduce some misbehaving test components which substitutes regular components within an assembly (e.g. empty or wrong executor logic code). Alternatively expected (valid) test traces can be compared with the test observations. Parts of the expectations may be derived semi-automatically from design models.

All the tests which are derived from the standard CCM interfaces and the application logic dependent test will form a test suite which may be added to a CCM assembly zip-archive. The test code is offered to a customer in an “all-inclusive” CCM assembly to be deployed and run with the end-customer environment.

The implementation of the test generation automation benefits from the availability of CCM open source infrastructure. E.g. a modification of a CCM idl compiler supports the generation of application dependent test code or skeletons.

4 Components testing using TTCN3

In our approach we've decided to apply an abstract test notation for the definition of the test cases. Standardized test notations like TTCN3 [4] or the testing profile for UML [13] appears as suitable candidates. TTCN3 have been applied in the context of Corba for testing both synchronous and asynchronous communication [5][12]. The UML testing profile is another alternative test notation but it is due to the incomplete standardization process of UML2.0 within the OMG not available yet.

The initial exercise to run and test the SUT using TTCN3 is to provide a mapping between selected TTCN3 language elements and the (programming) execution statements to initialize and run the CCM infrastructure and CCM applications. In the context of the well-known CCM examples (e.g. "Dining Philosophers") we have to consider at least the following issues³:

1. to initialize the ORB, and
to obtain CORBA services (e.g. Naming, Transaction),
2. to obtain Component servers,
to obtain home factories and homes,
to install archives, and
to create homes
3. to create components, and
to connect components,
4. to access (test) the interfaces.

The issues are different w.r.t. their numbers of occurrences, e.g. the first two issues (first group) have to be executed only once. Issues of the second group have to be done for every server involved in the application test. The execution of operations of the third group depend on the particular configuration of the SUT. The access to the interfaces of the components under test is test case specific. W.r.t. this overview it appears economic that operations of the first and even some of the second group have to be executed only once in the context of a test suite (e.g. as part of the TTCN3 control part).

Due to the programming language character of TTCN3 and the possibility to define external functions (which can be implemented in the so-called test adapter using a programming languages) it is obvious that in principle the TTCN3 language can express any of the envisaged test scenarios. Nevertheless TTCN3 has been designed for testing, i.e. its value is an unambiguous relationship to testing concepts including the specification of the SUT ports and operation types, verdict assignments etc.

So far we've used the following TTCN3 language constructs to specify the main CCM issues listed above:

- `type component` for the SUT configuration,
- `.call`, `.getreply` and `.catch` to control synchronous component operation calls, and
- `external function` for the initialization of the ORB and SUT configuration.

Parts of a sample TTCN3 test suite are given in Figure 1. Two external function have been

3. This list corresponds to a (Java) client implementation found within the OpenCCM [3] demo.

implemented to separate test case independent and specific initializations for the ORB (OpenCCM v0.2/Orbacus) and SUT (the parking simulation demo of OpenCCM). Interfaces between the test system and the SUT are defined by ports. They have to be addressed within the abstract TTCN3 components which describe the test system (MTC_Type) and the SUT (mySUT). Further definitions are on the signatures of operations, their data types and values (templates). Only one test case is defined: it comprise the mapping of a test system port to an interface of the SUT and a basic test scenario with call/reply pair on a synchronous SUT interface operation. There is also a simple control part which invokes an external function and

```

module BarrierTester {
  external function InitOpenCCM() return boolean;
  external function InitsUT() return boolean;
  type port CORBA_VehicleCapacity procedure {inout
    CORBA_VehicleCapacity_cross_barrier};
  type port CORBA_BarrierAccess procedure {inout CORBA_BarrierAccess_cross};
  type component MTC_Type {
    port CORBA_VehicleCapacity MTCpco;
    port CORBA_BarrierAccess BarrierPCO}
  type component mySUT {
    port CORBA_VehicleCapacity v1,v2,v3,v4;
    port CORBA_BarrierAccess b_provide_for_vehicles1,
      b_provide_for_vehicles2,b_provide_for_vehicles3}
  type enumerated CORBA_VehicleState {CORBA_INSIDE, CORBA_OUTSIDE};
  const charstring Parking_WrongWay :=
    "fr.lifl.goal.OpenCCM.parkingsim.WrongWay";
  const charstring Parking_Full := "fr.lifl.goal.OpenCCM.parkingsim.Full";
  const charstring Parking_Closed := "fr.lifl.goal.OpenCCM.parkingsim.Closed";
  signature CORBA_VehicleCapacity_cross_barrier ();
  signature CORBA_BarrierAccess_cross (in CORBA_VehicleState state)
    exception(charstring);
  template CORBA_BarrierAccess_cross from_outside := {CORBA_OUTSIDE};
  template CORBA_BarrierAccess_cross from_inside := {CORBA_INSIDE};
  testcase BarrierAccessENTRY() runs on MTC_Type system mySUT{
    if (not(InitsUT())){ verdict.set(inconc); stop;}
    map(self:BarrierPCO, system:b_provide_for_vehicles2);
    BarrierPCO.call (CORBA_BarrierAccess_cross:from_outside);
    alt
    {[] BarrierPCO.getreply;
     [] BarrierPCO.catch() { verdict.set(inconc);}
    }
    BarrierPCO.call (CORBA_BarrierAccess_cross:from_inside);
    alt
    {[] BarrierPCO.getreply { verdict.set(fail); stop;}
     [] BarrierPCO.catch(CORBA_BarrierAccess_cross, Parking_WrongWay)
       { verdict.set(pass);}
     [] BarrierPCO.catch() { verdict.set(inconc);}
    }
    unmap(self:BarrierPCO, system:b_provide_for_vehicles2); stop;
  }
  control{
    if (InitOpenCCM())
      { execute(BarrierAccessENTRY());}
  }
}

```

Figure 1: TTCN3 test case sample

the test case execution only.

It has been compiled and executed using TTthree, TTman and a corresponding runtime environment [7]. The test adapter which is required for the implementation of the abstract TTCN3 statements has been implemented in Java.

5 Derivation and deployment of test components

In the previous section the executable test behaviour of test components has been defined and derived using TTCN3. In opposite to these test purpose related behavioural scenarios there are also test components which have to fulfil a passive responder role only, e.g. to substitute one or more components of the SUT. Furthermore it appears meaningful to monitor the communication at the interfaces between two SUT components. The observations made during a test campaign can be delivered and evaluated by TTCN3 based test components, too.

Please see Figure 2 for an example test configuration which include a proxy-based test component. Three test system components have been introduced: “Tester 1“ is the active test driver (e.g. TTCN3 based) which initiates component 1 to communicate with its peer. “Tester 2“ has the passive (proxy) role which means that all operations between component 1 and 2 is recognized by this test system component. The resulting behaviour (if observable) may be verified either directly by the test driver or via an optional “Tester 3“. The observations of the proxy will be delivered for test interpretation to the other test system components.

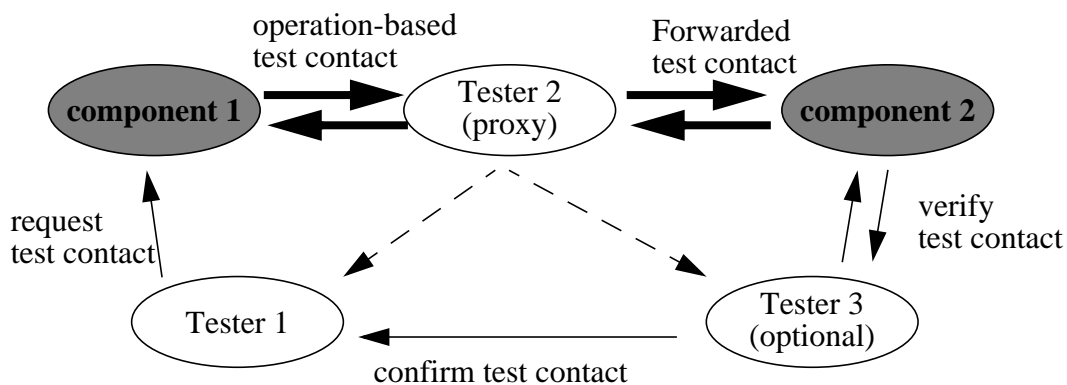


Figure 2: Test configuration with proxy based test component

Based on an IDL3 specification of a SUT component the IDL3 definition of the corresponding proxy can be derived. In the example illustrated in Figure 2 interfaces and attributes provided by component 2 have to be offered by the proxy, too. It has to be noted that all interfaces provided by Tester 2 are also used by the proxy from component 2. On the other hand the proxy does not address the interfaces which are used by the original SUT component.

Let’s consider an example taken from the OpenCCM example “parking simulation” [3]: W.r.t the definitions used in Figure 2 the ‘Barrier’ component corresponds to component 2. It offers interface operation to another ‘Vehicle’ component (component 1 role) and is defined by the following:

```

component Barrier
{
    // barrier states
    readonly attribute string description;
    readonly attribute BarrierKind kind;
}
  
```

```

// barrier receptacles
provides BarrierAcces for_vehicle;
uses ParkingAccess parking;
}

```

We can distinguish between proxies which do cover only one interface of a SUT component and a proxy which covers all interfaces of the SUT component. The IDL3 definition in the latter case will be below:

```

component tester2
{
    // provided to components using "Barrier".
    readonly attribute string description;
    readonly attribute BarrierKind kind;
    provides BarrierAcces for_vehicle;
    // used from the SUT component "Barrier".
    uses BarrierAccess from_barrier;
};

```

A more compact alternative is to apply IDL inheritance: W.r.t. substitution in the next suggestion the component type of the proxy has the same type as the component of the related SUT component. A drawback on the other hand is that the proxy inherits also used interfaces of the SUT component, although not necessary.

```

component tester2 : Barrier
{
    // used from the SUT component "Barrier".
    uses BarrierAccess from_barrier;
};

```

Independently of the selected option the behaviour implementation of the proxy can be generated automatically. For each operation offered by the interfaces of the proxy the incoming operation calls are forwarded to the corresponding interfaces of the hidden SUT component. Additionally such information can be made available to other test system components. Either

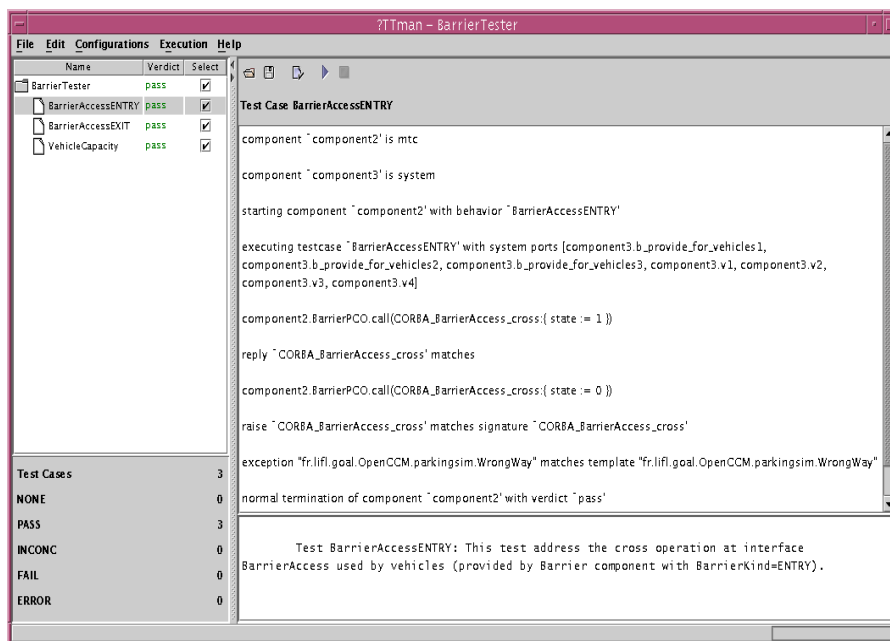


Figure 3: TTCN3 test manager

the proxy publishes an event or uses an interface operation of the cooperating testers (pushing model), or the proxy offers the information at an own (provided) interface (polling model).

To enable an easy test campaign installation and management at different locations the test system components can be added to the SUT assembly and its XML descriptor file. Passive test components as described above will be deployed and instantiated like SUT components. Currently the TTCN3 components will be deployed but not started by a CCM deployment tool. A TTCN3 test manager has to be invoked by the test operator until a CCM interface is available. See Figure 3 for a snapshot of the test management tool TTman which we have used during a test campaign run of our TTCN3 test suite.

6 Summary and outlook

In this paper we've presented our analysis and roadmap for testing CCM components. It's based on an abstract test definition using TTCN3 and covers also passive (monitoring) test components (proxies). In opposite to other approaches e.g. using in-house test notations [15] or addressing Java-Beans [10] we have used a standardized test notation which is independent of the component programming environments and supports a distributed test system.

We do not expect a formal specification of the SUT. This might be used for automatic test case generation but will create problem for test execution due to its abstraction level [14]. Our approach could be seen as a pragmatic solution which let the test system become part of the component assembly and be deployed together with the SUT at the target system.

No assumptions could be done yet on a component test interface to allow further access to the internal state information and/or so-called build-in tests. The latter issue is under discussion within the OMG and promote the idea that test software is already part of the components under test and will be activated during the test campaign.

References

- [1] OMG: CORBA Component Model, v3.0, <http://www.omg.org/technology/documents/formal/components.htm>
- [2] OMG: CORBA / IIOP Specification, v3.0, http://www.omg.org/technology/documents/formal/corba_iiop.htm
- [3] Open CORBA Component Model Platform (OpenCCM). <http://corbaweb.lifl.fr/OpenCCM/>
- [4] ETSI: Testing and Test Control Notation (TTCN3). <http://www.etsi.org/ptcc/ptccttn3.htm>
- [5] M. Schünemann et al.: Improving test software using TTCN3, GMD Report No. 153, Dec. 2001. <http://www.gmd.de/publications/report/0153/>
- [6] I. Schieferdecker et al.: The CORVAL2 Contribution to achieve Confidence in Middleware. ERCIM News No. 49, April 2002. http://www.ercim.org/publication/Ercim_News/enw49/
- [7] Testing Technologies: TT Tool Series. <http://www.testingtech.com/products/>
- [8] COACH Deliverable 1.1, Telecom domain requirements upon component architectures. Sept. 2002.

- [9] COACH Deliverable 1.3, Requirements for the component tool chain and the component architecture. Sept. 2002.
- [10] N. Koppalkar et al.: Testing of Component-Based Software Systems. 3rd Software Testing Conference, Bangalore, India, Nov. 2001.
- [11] R. Marvie, P. Merle: CORBA Component Model: Discussion and Use with OpenCCM. http://corbaweb.lifl.fr/OpenCCM/docs/2001_06_Informatica.ps
- [12] A. Yin et al.: Operation-based interface testing on different abstraction levels. ICSSEA'2001, Paris, Dec.2001.
- [13] OMG ADTF. UML testing profile. <http://www.fokus.fhg.de/tip/u2tp/>
- [14] I. Ryl et al.: A component oriented notation for behavioral specification and validation. SAVCBS'2001, Tampa Bay (FL), October 2001. <http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/>
- [15] D. Flater: Manufacturer's CORBA Interface Testing Toolkit: Overview, Journal of Research of the NIST, v.104, n.2, 193-200, 1999. <http://www.nist.gov/msidlibrary/doc/j42fla.pdf>
- [16] D. Rosenblum: Adequate Testing of Component-Based Software: Technical Report 97-34, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, Aug. 1997.
- [17] Y. Wu et al.: Testing Component-Based Software. Information and Software Engineering Dept, George Mason University, Fairfax, U.S.A. <http://www.isse.gmu.edu/~wuye/classes/637/paper/cbs.pdf>.
- [18] L. Martin et al.: Extending SUnit to Test Components. In: K. Bauknecht et al. (editors): Informatik 2001, vol. 2, page 834 - 839, Sep. 2001
- [19] A. McCarthy: Unit and Regression Testing. Dr. Dobbs Journal, Feb. 1997.
- [20] E. Weyuker: Testing Component-Based Software - A Cautionary Tale: IEEE Software, pages 54-59, Sep./Oct. 1998