Implementation of an Open Source Toolset for CCM Components and Systems Testing *

Harold Batteram[1], Wim Hellenthal[1], Willem Romijn[1],
Andreas Hoffmann[2], Axel Rennoch[2], Alain Vouffo[2]

[1]Bell Labs Advanced Technologies Nederland,
Larenseweg 50, NL-1200 BD Hilversum, The Netherlands
{batteram, whellenthal, romijn}@lucent.com
www.lucent.nl/bell-labs
[2]Fraunhofer FOKUS, Competence Center TIP,
Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
{a.hoffmann, rennoch, vouffo}@fokus.fhg.de
www.fokus.fhg.de/tip/

Abstract. Following the success of CORBA based systems the OMG has standardized the CORBA Component Model (CCM) to improve the implementation process of large distributed systems. The European project COACH [16] has been set up to build an Open Source development platform to construct CCM applications. As part of COACH a toolset for CCM components and system testing has been defined and implemented. This paper introduces the various components and features which have been foreseen and implemented for test activities such as interactive component testing, test trace visualization, or the application of abstract test specifications. The resulting test infrastructure addresses the CCM specifics but also benefits from CCM, e.g. by incorporating component communication facilities.

1 Introduction

This contribution presents a framework for testing software components and systems that are based on the CORBA Component Model (CCM)[17] standard. An important aspect of CCM based systems is that the system must be verifiable and testable at the abstraction level of its design and independent of the chosen component implementation language. Component based systems allow the development and testing of components to be partitioned across development groups, working in parallel. However, dependencies between separately developed components may cause delay in testing. Component oriented software engineering has enjoyed an increasing interest in research and industry over the last decade. Component oriented software engineering focuses on system composition from components as elementary building blocks. This approach has many advantages and is generally regarded as an efficient way to handle complexity in large systems and (when properly applied) to

* This work is partly sponsored by the European IST Project Coach (IST-2001-34445, Component Based Open Source Architecture for Distributed Telecom Applications).

R. Groz and R.M. Hierons (Eds.): TestCom 2004, LNCS 2978, pp. 1-16, 2004.
© IFIP 2004

improve overall system quality. Component frameworks such as Enterprise Java Beans (EJB) from Sun Microsystems, COM (and more recently .NET) from Microsoft have gained widespread acceptance in software development communities in a short time. The CCM is the component model proposed by the Object Management Group (OMG). As opposed to EJB and .NET the CCM is not proprietary but is an open industry standard. While EJB and .NET are usually applied in web based, client-server application domains, the CCM can be applied in many industrial domains including the telecommunication industry.

Using the CCM requires a different approach to the development cycle. Software systems are now composed from multiple components and separate the task of component development from system composition. This also requires a different approach for the testing of CCM based systems.

There are multiple parts during the whole development cycle of a CCM based system where testing is needed. For example, a component developer needs to test individual components, or small clusters of components working together. System developers compose systems from readily available components but must test the various interaction scenarios for the specific composition in which the components are used. Finally, with large and complex systems integration and conformance testing is often done by specialized system testers and not by developers.

In the context of testing CCM components there is less related work to be mentioned. The majority of the CCM implementers did not address testing by specific techniques or tools. Furthermore we did not assume the availability of an UML specification of the System under Test (SUT) as it has been done in other approaches [4][6].

We have only found the following work carried out by L. Johnson and E. Teiniker on a project called CORBA Component Model Tools [12]. Their aim is to provide tools used for generating CORBA components, test components, and test programs based on source IDL files. Testing has been identified as an important issue. They intend to test their applications on different levels during development: Every class of the business logic that will be part of a component has to be tested (Class Level Testing). For every component a counter component that looks like a mirror to the original component will be created. This counter component has a receptacle for every facet of the original component and vice versa (Component Level Testing). Testing should include a set of connected components (Assembly Level Testing).

Johnson and Teiniker follow the methodology of Beck [3] who introduces the Test Driven Development and starts implementing the test before implementing the application. A test client coordinates creating and connecting the components, as well as the test calls to the facets of a given component C. The mirror component C' and the test client are always generated at the same time as the component itself, without additional development effort. The process begins automatically after compiling. The development of the local component is separated from the development of the test client. A deployed component can be used in different applications at the same time. Currently there are no test tools documented in the project that have been started recently in 2003.

From this latest survey of existing testing tools no test implementations do fit the requirements stated within the COACH project [1][9] and there is still the need to build an appropriate CCM testing environment.

The test architecture described in this paper addresses two distinct test audiences, developers and system testers. Although both groups have a common goal of testing

the overall quality of a CCM system, each group has a different focus and a different approach to reach that goal. Component programmers are concerned with testing at a lower level and have to take infrastructure specifics into account such as concurrency, performance, stress testing etc. During the development cycle, component programmers also often use test tools as a debugging facility to locate implementation errors in components, or for regression testing to verify modifications done on a component implementation has not affected other component functions.

Considering system testing there is a special need to compare application tests and the test results gathered from test campaigns executed with different vendor implementations and heterogeneous user environments. In this context the conformance to (standardized) system requirements has to be shown and an international standardized test notation has to be applied. Test languages such as TTCN-3 [13] are specifically designed by ETSI and ITU-T for such kind of testing.

Our CCM test architecture describes a full set of tools that cover both component developer oriented testing and system integration and conformance testing for system testers.

This paper is organised as follows: Section 2 specifies the component test framework describing the different parts of our framework. Section 3 addresses test notations which can be applied to the components under test in order to automate the tests. Section 4 investigates in the test trace viewing facilities. Section 5 describes an application of the test tools. And finally the conclusions will summarise the major results of this paper.

2 Component Test Framework Description

Our test framework, i.e. the COACH test framework, is only concerned with testing CCM components and interactions between CCM components. This means that the framework can be used to identify components that do not behave according to their specifications. Once a component containing a fault has been identified, further localization of the fault within the component can be done using the test and debug facilities that are usually part of the implementation language specific development environment. This implementation language specific testing and debugging is outside the scope of the test framework specified in this contribution.

Tests on CCM components and the observation of interactions between components are expressed using IDL data types and are independent of the data types of the implementation language of the component. Our CCM test framework provides the testers with the following essential capabilities:

- The ability to invoke CORBA operations on selected component facets and observe the response, using a so-called Actor component.
- The ability to intercept CORBA operations on two different levels:

  - Inside a component using the portable interceptors

  - Outside the component using so-called Proxy Components

- The ability to extend the range of test scenarios for components that have dependencies with other components whose implementation is unavailable; using so-called Reactor components as substitutes.
- The ability to visualize causality relations between invocations at runtime.
- The ability to run standardized abstract TTCN-3 test cases against CCM applications under test.

The following figure gives an overview of the test framework. The elements that are above and on the right side of the SUT are used during the development phase of the system under test. The elements left from the SUT are used after the SUT is finished and ready for deployment.

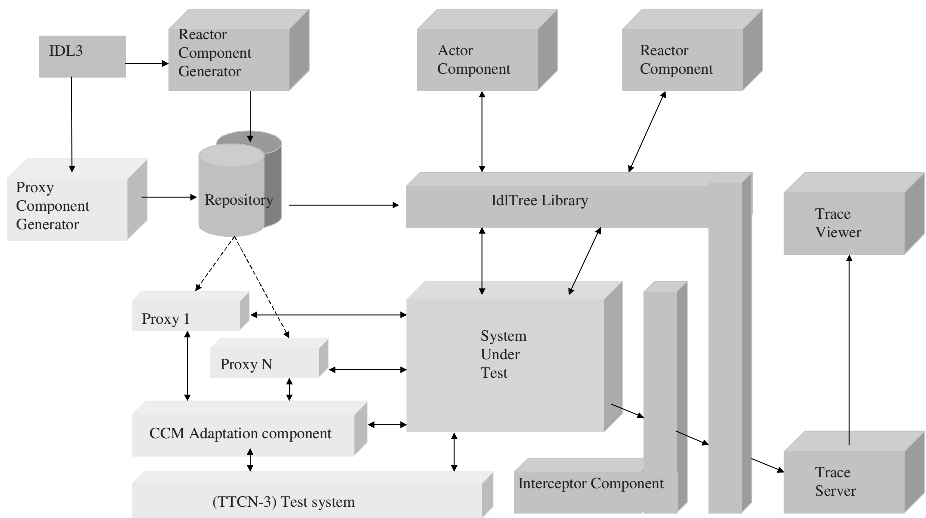

Fig. 1. Component Test Framework

The CCM test framework addresses each capability as will be explained further in the next sections.

2.1 Developers Testing Tools

The CCM is well suited to develop large-scale distributed applications. Teams of developers typically develop these types of applications where each team is responsible for a subsystem. One developer may be responsible for the development of a group of components that will eventually need to be integrated into the system. A developer will need to test the set of components (s)he is responsible for as much as possible before integrating them into the whole system.

The ability to test components may be severely restricted when the components under test depend on interactions between other components that are not yet implemented. To reduce this restriction, the dependent components can be substituted

by, so-called, Reactor components for the purpose of testing only. Reactor components must provide the same set of facets, operations and events as their real counterparts. If the IDL specification is known, Reactor component implementations can be generated automatically.

The Reactor-implementation-generation process is similar to the generation of Stub, Skeleton and other implementation classes by the IDL compiler. The IDL specification is read as input and Reactor implementation classes are produced as output that then needs to be compiled by the implementation language compiler. Although IDL can be translated into different kind of implementation languages, the Reactor components do not necessarily need to be implemented in the same language as the components they are substituting. After all, one of the strengths of CORBA systems is heterogeneity. It does not matter in what language a component is implemented as long as it supports the same IDL interfaces. For practical purposes we have chosen to generate Reactor components in Java.

The implementation of the Reactor component must be configurable to allow different kind of responses. The response may be interactive allowing the tester to examine the parameter values and construct a reply using an interactive IDL type editor, or the response is automated. The Reactor can be hard-coded to give an automated response, or by executing a general purpose scripting language that can be loaded and interpreted by the Reactor at runtime. The latter is obviously more flexible but may not be necessary for simple test cases or may have an unacceptable performance penalty. In any case, the tester must be able to make the choice. When an invocation arrives on a Reactor component facet it can reply (within limits) as if the real component is in place. The range of possible test scenarios is now extended for the components under test and can reduce the probability of errors when the final components replace the Reactor components when they are available.

Of course, the behavior of Reactor components is determined by the interactive or programmed response and will most likely differ from their real implementation. Nevertheless, the presence of Reactor components can demonstrate correct behavior of the components under test for various interaction scenarios. In particular error conditions occurring in the Reactor components can usually be simulated more easily using Reactors then real implementations. Even when real implementations become available, Reactor components are still useful for regression testing.

Another part of the test framework is the Actor component that acts as a general purpose CCM client component that can invoke operations on other components. The Actor can also load and execute test scripts or can be run in interactive mode. In interactive mode the tester can interactively fill in parameter values for a selected operation, invoke the operation and examine the result. In order to invoke an operation on a facet of a target component, the Interoperable Object Reference (IOR) of the component must be obtainable. In CCM systems, key components usually publish their IORs using a naming server. References to other components may be passed as return values of operations. References to component facets can be obtained by using the navigation operations provided by the component interface.

In addition to providing the tester with a means of testing components using an actor and reactor, the CCM test framework allows the tester to trace and visualize the propagation of invocations between CCM components, see section 4 for the details.

With the combination of Actor, Reactor, and Invocation tracing viewer the implementers of CCM components have a powerful set of tools available to test their CCM components at an early stage.

To illustrate the developers' test approach with an example, suppose a developer has implemented components C1 and C2 as depicted in Figure 2. Component C1 interacts with C2 using facet f3. The implementation of C1 also needs to interact with facet f5 of component R1 and facet f6 of component R2 as shown in Figure 2. This figure also shows that component C2 interacts with facet f7 of component R2. The implementation of R1 and R2 is outside the scope of our developer and the implementation may not become available for some time. This situation limits the test scenario possibilities for components C1 and C2 since the test scenarios in which invocations of R1 and R2 occur must be omitted. However, with the IDL specification known for R1 and R2, Reactor components are generated and instantiated as part of the test system. The test scenarios for C1 and C2 are now extended to include invocations to R1 and R2.

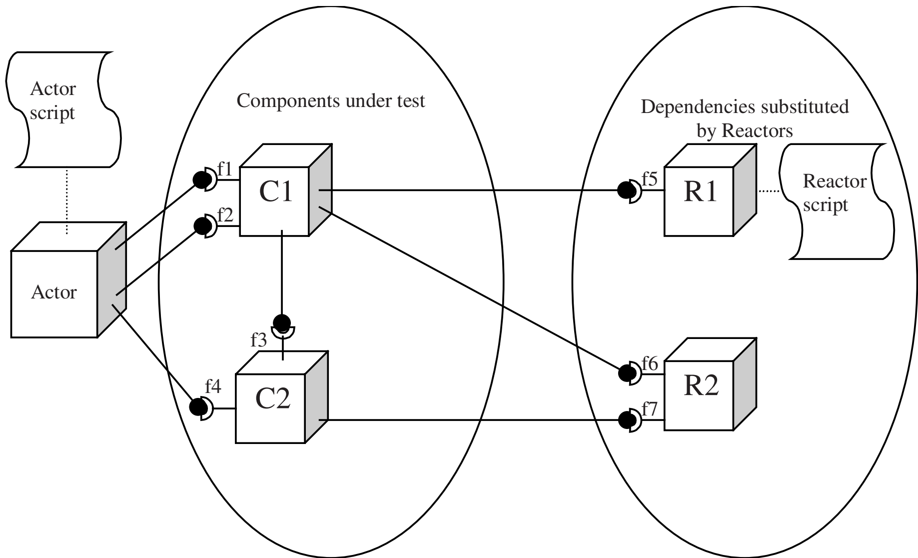

Fig. 2. Actor-reactor test environment

To test C1 and C2 the Actor is used. Figure 2 shows how the Actor user interface is used to invoke operations on the facets of the components under test C1 and C2 interactively. The Actor includes a naming server browser that allows the tester to examine the content of the naming server and to select one of the registered references. If the selected reference is a component reference, the Actor will introspect the component to obtain information about the facets it supports and will display a new window that allows the tester to select a target facet reference. Once a target facet is selected, another window is shown in which parameter values can be interactively filled in. The signatures of the operations and the structure of the parameters is obtained from an interface repository. In this example Figure 3 shows the selected operation of facet f2 of C1 on the left panel. After 'method3' has been invoked the result of the operation is shown on the right panel.

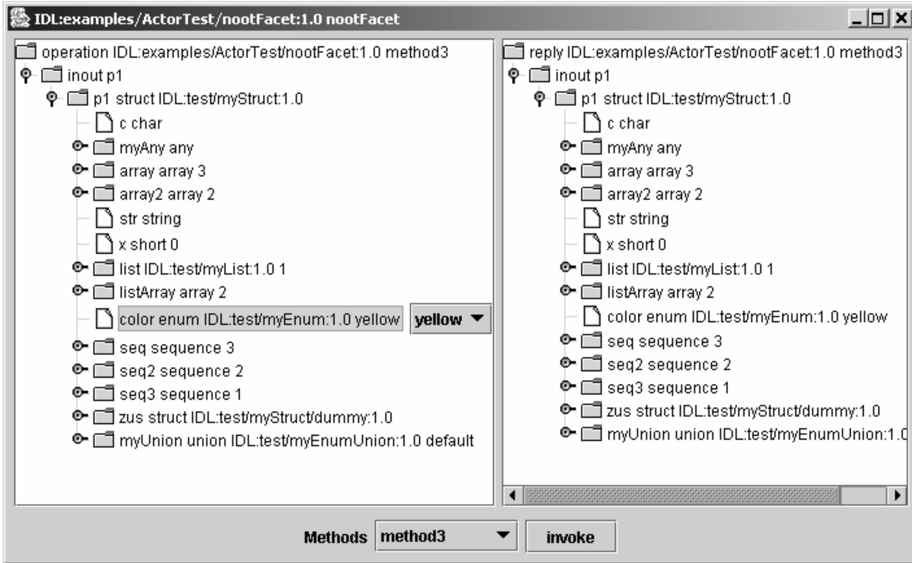

Fig. 3. Actor invocation dialog

After the test scenarios for C1 and C2 are completed by repeating the method described above for each step in the scenario the tester can visualize the propagation of invocations between C1 and C2, this invocation trace is presented in standard web browser. Figure 7 shows a similar trace from another example. The tester can now compare this message sequence chart with the same chart in the specification and verify if the components behave accordingly.

2.2 Systems Testing Tools

System testers are more engaged in the acceptance of the complete (customers) target application. The requirements and test purposes from the application users are different from the developer's viewpoint. System testers focus on the whole system or at least a meaningful subsystem that fulfills a particular work. The in-depth testing phase of the component developer using such test tools as described in the previous section is a prerequisite for the examination of a large composed system. Single components A and B have to operate according to their interface definitions and semantics specification. A system including multiple components A and B interworking possibly with each other and further components of different type fulfills an overall functionality which has to be evaluated and/or demonstrated according to an unambiguous test plan and has to deliver a meaningful test report.

It is often sufficient to run applications tests at the API provided for end-customers only. In this situation the test engineers have a choice between lots of test tools provided at the software market. An easy approach to integrate observation points into component-based system implementation is proposed for system testing: We have implemented CCM components that can be introduced at communication points between components under test (SUT components). The primary task of such so-

called *proxy* components is the collection of proper information about the interaction between SUT components and the distribution of corresponding notifications to various applications (e.g. an arbiter). In contrast to the interceptor approach the proxy approach is more flexible (e.g. if C++ is used it does not need any component sources).

Due to the expressiveness of the IDL specification of the involved SUT components we had to address a number of design questions which we considered with the following features and decisions for the proxy components:

- One proxy component will cover a full communication profile (interfaces and event sinks) provided by one original SUT component; i.e. the proxy reflects the "incoming" side of a SUT component.
- It is subject to the test engineer (i.e. his interests and preferred test configuration) to couple (deploy) all or only parts of the proxy interfaces (e.g. in the following Figure 4 a proxy for component C2 is introduced, but only one facet will be used).
- Each proxy component requires an individual component identifier to be identified within the SUT after deployment.
- In case of synchronous communication proxy components should not block any interaction during waiting for a reaction in response to an operation invocation.

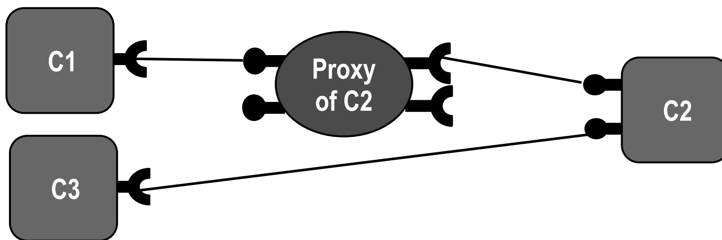

Fig. 4. Sample proxy configuration

Other major questions address the test system applications that should take care on the observations of the proxy component and how to distribute the proxy notifications. The approach depends on the facilities and intention of the test engineers. Conceptionally a specific test system component is foreseen to adapt the preferred test system application responsible for filtering, analysis or presentation of the observations (see the following sections for sample applications).

A comprehensive IDL3 event type has been defined to serve as a common interface for the different adaptation approaches. Such proxy notifications are delivered using CCM event channels. In our implementation we've distinguished *proxy_request, proxy_reply* and *proxy_exception* notifications for synchronous SUT interaction (see Figure 5) and a single *proxy_event* notification for asynchronous SUT interaction. Transported data comprise details on operation name, parameters (in/out, results), location (proxy identifier) and timestamps of the observations.

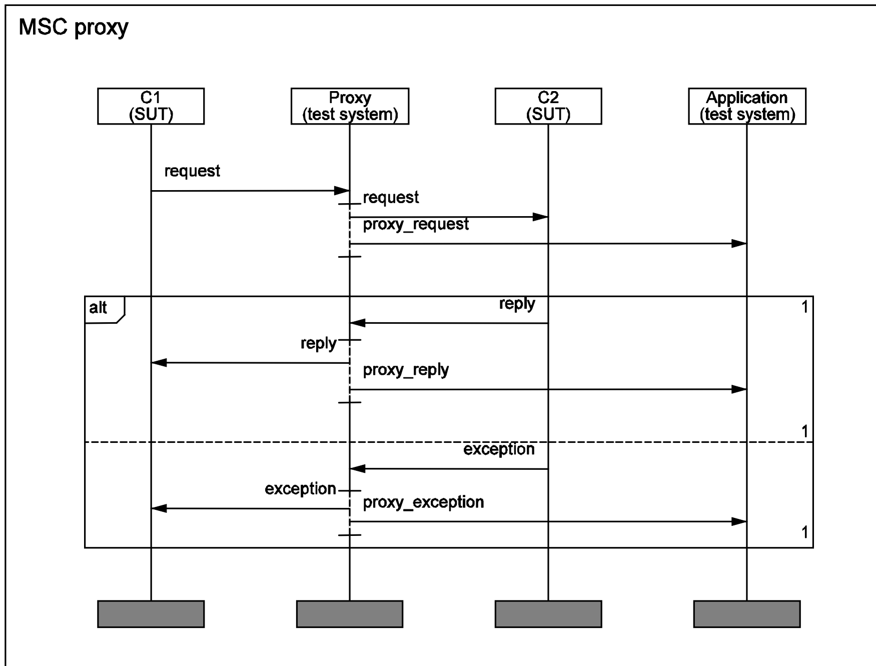

Fig. 5. Proxy component interactions

With the broadcast (publishing) feature of the CCM event communication it is possible to use different concurrent adaptation components, i.e. evaluation systems. Such adaptation components fulfill a bridge function to a non CCM-based test equipment. Unfortunately it is required to implement the specific needs for each application. On the other hand there is no implementation necessary to get the proxy components: We've selected the Open source CCM development environment Qedo [21] to generate proxy source code which can be compiled, linked and deployed with conventional tools. Due to the CCM standardized interfaces the SUT components can be based on any implementation language and do not need to be developed with Qedo.

Our implementation extends the Qedo compiler in such a way that it is sufficient to list the components (using the new "- - *proxy*" option) for which proxies should be build when Qedo is called for an IDL3 specification of the SUT. Proxy components will be generated once and can be instantiated and deployed multiple times.

3 Test Definition Languages

The COACH test framework contains a library, which allows scripts written in the popular TCL language to interact with the CCM environment. The TCL library works closely with the IdlTree library and uses the freely available Jacl [8] implementation of TCL written in Java. Although TCL is currently used for the COACH test tools, libraries for other scripting languages such as Python [2] can easily be used in the

same way. The choice for TCL is mainly made from a pragmatic point of view since it easily integrates with Java and because it is widely accepted as an effective scripting language.

The goal of the COACH TCL library is to support CCM programmers, which need to test their components in an easy way. The TCL library consists of a set of TCL procedures that provide an API with the CORBA and CCM environment. With the help of the IdlTree library, complex IDL parameter values can be easily constructed and modified within a TCL script. These API procedures allow CCM components to be located, their facet references obtained, parameter values to be constructed and invocations to be done on a facet from a TCL script. The result of the invocation can be compared or used as a parameter value for subsequent operations.

The Actor can load and execute TCL test script in a TCL shell window. The tester can also interactively type in TCL statements and observe the result in the shell.

If large SUT from different vendors have to be validated in multiple user environments abstract test specifications are recommended for system testing. Their advantages are good readability as well as a standardized and common understanding of the test semantics. Test languages such as TTCN are specifically designed by the ITU for such kind of testing while scripting languages such as TCL, Python and IDLscript are more suitable for developers.

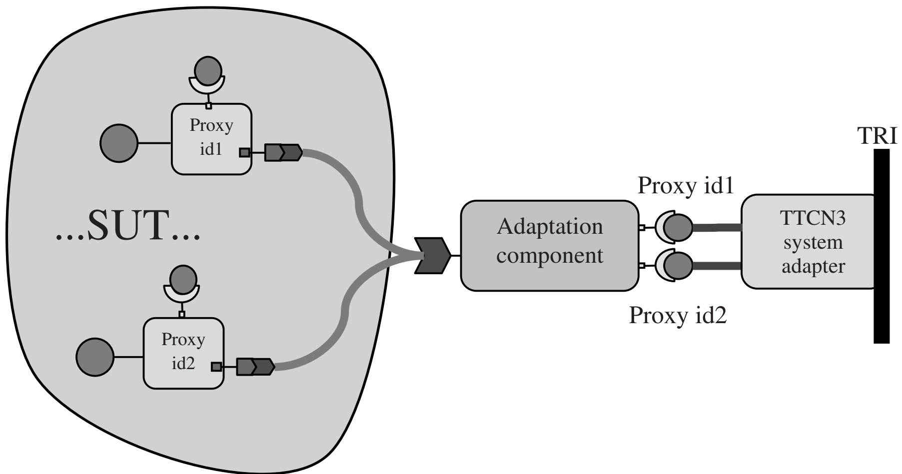

Fig. 6. TTCN-3 test system adaptation sample

Standardized test notations like TTCN-3 [13] or the testing profile for UML [18] are mature candidates and go beyond XML-based test notations as proposed in [5]. Message-based protocol conformance testing and operation-based CORBA application tests have already been done with TTCN-3 [10][11]. The UML testing profile was another alternative test notation but it was due to the incomplete standardization process of UML2.0 within the OMG not available at the beginning of our work.

In principle a TTCN-3 based test system can access operations under test at every known (or registered) CCM component. Initial trails for the application of TTCN-3 to test CCM applications have been already presented in [7], but have been limited to operation-based component interfaces and did not consider the proxy components introduced above. Furthermore, in the meanwhile the IDL2.* to TTCN-3 mapping standard from ETSI [14] could be used and requires only small additions for covering IDL3 event communication: Events can be exchanged at message-based TTCN-3 ports. Event types can be expressed by TTCN-3 record structures. The TTCN-3 *group* definition may collect all data types and templates that are related to an event type.

Our CCM adaptation component for TTCN-3 assumes a CORBA interface (for each proxy) provided by the TTCN-3 system adapter that collects (incoming) operation calls to related port instances defined in the TTCN-3 abstract test suite. Figure 6 illustrates the situation and the interfaces exemplarily involving two proxy components with a TTCN-3 system adapter that provides the related CORBA interfaces and applies to the TTCN-3 runtime interface (TRI) [15].

4 Test Trace Viewing Facilities

In addition to providing the tester with a means of testing components using an actor and reactor, the CCM test framework allows the tester to trace and visualize the propagation of invocations between CCM components. Invocation tracing is useful for such things as comparing the runtime behavior of a planned system with its design specifications. The Tracer framework presented in this section consists of two parts:

- TraceServer
- TraceViewer

The TraceServer is a CCM component that contains a collection of events that occurred within the system under test. An event basically is an interaction between CCM components. The TraceServer responds to queries by returning the requested event data formatted in XML, including complex parameter data types.

The TraceViewer (see Figure 7) is a combination of a web server and a web client. The web server acts as an intermediary between the TraceServer and the web client. It translates HTTP requests from the web client into TraceServer queries using CORBA invocations. The result is send back to the web client as plain text XML. The web client is the vehicle to visualize the data received from the web server in a user-friendly manner. The client depends heavily on JavaScript code, not only for dynamic SVG [24] creation but also for user control about how the information is presented.

Events are part of an invocation trail with a start point and an end point. As an invocation propagates through a CCM system it carries context information. This context information is extracted and updated at each interaction point.

The trace framework is generally concerned with tracing and visualizing invocations between CCM components. However, in a CCM system several communicating entities can be distinguished such as facets, receptacles event sources, event sinks, components, containers and plain CORBA objects. In order to present a meaningful picture of the various interactions we must know what the relation between an event and the communication entity is.

Fig. 7. Invocation tracing viewer

The underlying communication mechanism in CCM is still based on standard CORBA. The CORBA IDL compiler generates stubs and skeletons. An invocation on a CORBA object passes through a stub instance on exit and passes through a skeleton on entry. At the CORBA level, an invocation enters an entity at the POA_IN. It leaves an entity at the STUB. This results in the definition of the interaction points as shown in Listing 1. Each trace event contains information from which interaction point it was sent.

```
module tracing {
   enum InteractionPoint {
      STUB_OUT,
      POA_IN,
      POA_OUT,
      POA_OUT_EXCEPTION,
      STUB_IN,
      STUB_IN_EXCEPTION,
      ONEWAY_STUB_OUT,
      ONEWAY_POA_IN
   };
};
```

Listing 1. IDL Interaction points specification

At each interaction point a trace event must be send to the TraceServer component with timing and identity information about the interaction. This requires that the

invocation flow at the interactions points is intercepted to allow for the additional actions to collect and sent the trace information. The CORBA Portable Interceptor (PI) specification [19] defines a portable mechanism to intercept the invocation flow of an operation on a CORBA object. Since CCM Component facets are implemented as normal CORBA objects, this mechanism is also suitable for the implementation of invocation tracing for CCM component interactions. The PI mechanism also allows additional service data to be propagated transparently between CORBA invocations. This is used for example with a transaction service to propagate a transaction context. For the purpose of invocation tracing, a TraceService can be created which propagates tracing context information between CORBA invocations. The IDL specification of the propagation context is shown in Listing 2.

```
struct PropagationContext {
    // user defined indentification string to mark a segment of an invocation trail
    string trail_label;

    // Id of the originating thread at the start of the invocation chain.
    string trail_id;
    long interaction_counter;
};
```

Listing 2. IDL Propagation Context specification.

The originator_id field is initialised at the start of an invocation trail and uniquely labels the trail as it propagates through distributed components. The interaction_counter field is incremented at each interaction point and is used to determine the proper order between interactions. The trailLabel field is an optional, additional label to mark an invocation trail. It can be used in combination with the Actor and the scripting environment to highlight specific sections of an invocation trail with a user defined label. TraceViewer applications can use this label to visualize such segment in the graphical output.

5 Application of the Test Tools

Due to the scope of the different test tools introduced above there are various possibilities for test engineers to benefit, i.e. the tools can be combined according to the specific needs (e.g. see Figure 8 which illustrates a combination of the proxy components with the tracing tools). The preliminary version of the test tools are an input to the COACH project workpackages developing large CCM based applications for the telecom domain (a Network Management Framework and a Parlay Platform) which are due to the COACH project termination in spring 2004. At the time of writing the experiences with the usage of our toolset are restricted to our own test applications.

The developers testing tools have been already successfully used during the implementation of some own components like e.g. the TraceServer. Predecessors of the Actor and Reactor test components were already successfully used in the

WINMAN project [23] starting at the specification phase until the final integration phase. During the integration phase they used Reactor components to isolate part of the system without disturbing the operation of the rest of the system. The actor was used to execute acceptance test scripts.

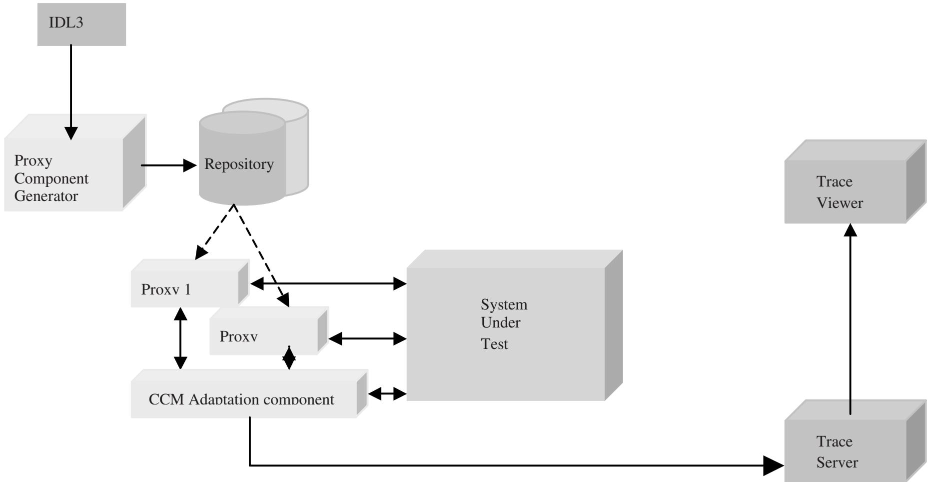

Fig. 8. Sample test configuration for online monitoring

The systems testing tools have been applied to the standard CCM examples supplied by the OMG (e.g. "dining philosophers"). It has been already demonstrated within the project how proxy components are suitable for feeding there observations to the COACH TraceServer (see section 4) and also in combination with a TTCN-3 based online monitoring system that compares observed interactions from the SUT based on abstract TTCN-3 test definitions. In the latter case the proxy components which had been generated once have been applied in multiple test cases and address various test purposes according to the selected test configurations.

6 Summary and Conclusions

The area of component based testing is a rather new area in the sense that there are almost no specialized tools in testing the development or acceptance of CCM components. The work presented in this paper is a first step to realize a test and acceptance framework for these components. A comprehensive set of testing tools have been described and implemented which can be used at an early stage of the production of CCM based applications as well as during the system acceptance testing because its scope covers the needs of component software implementers and the requirements of system test engineers.

Test concepts like the Actor and Reactor have already proven their usefulness. Visualization techniques as implemented in the trace viewer provide useful feedback

for verifying the correctness of the implementation. Although two distinct user groups have been identified the test framework is flexible enough to allow other usage as well. It is at the discretion of the user whether to use the proxy based approach that can be used without modifying the component under test or the interceptor approach that allows for a rich choice in interaction points in the component under test at any stage of the development or acceptance.

The use of test scripts allows for test automation and a reproducible way of testing the components. Different needs such as monitoring, trace visualization and adaptation of well accepted (standardized) test notations have been considered and satisfied. The availability and active usage of such test tools in the CCM based software production shall support the quality, confidence and acceptance of CORBA components applications.

The Open Source implementations developed in the COACH project have been adopted and extended to create a meaningful test environment. The test facilities will be available with the CCM platforms OpenCCM [20] and Qedo [21].

References

1. H.J. Batteram and W.A. Romijn (editors), “Telecom domain requirements upon component architectures”. COACH deliverable D1.1.
2. D. Beazley, “Python Essential Reference”, ISBN: 0735710910, New Riders, 416 pages (June 2001)
3. K. Beck. Test-Driven Development by Example. Addison Wesley, 2003.
4. A. Bertolino, A. Polini: A Framework for Component Deployment Testing. Proceedings of the 25th International Conference on Software engineering, Portland, Oregon, Oct. 2003.
5. G. Bundell et al.: A Software component Verification Tool. SMT’00. Wollongong (AUS) Nov. 2000.
6. J. Hartmann et al.: UML-Based Integration Testing. ISSTA’00. Portland, Oregon, Aug. 2000.
7. A. Hoffmann et al.: CCM testing environment. ICSSEA’2002, Paris, Dec.2002.
8. R. Johnson, “Tcl and Java Integration”, Sun Microsystems Laboratories, February 3, 1998 http://www.tcl.tk/software/java/tcljava.pdf
9. J. Reznik (editor), “Requirements for the component tool chain and the component architecture”, COACH deliverable D1.3.
10. M. Schünemann et al.: Improving test software using TTCN-3, GMD Report No. 153, Dec. 2001. http://www.gmd.de/publications/report/0153/
11. A. Yin et al.: Operation-based interface testing on different abstraction levels. ICSSEA’2001, Paris, Dec.2001.
12. CCMtools project: http://sourceforge.net/projects/ccmtools
13. ETSI: Testing and Test Control Notation (TTCN-3). http://www.etsi.org/ptcc/ptccttcn3.htm
14. ETSI TS 102 219: Methods for Testing and Specification (MTS): The IDL to TTCN-3 Mapping, V1.1.1, 2003-06.
15. ETSI: TTCN-3 Runtime Interface (TRI). ES 201 873-5, Feb. 2003.
16. IST Project COACH web site, http://www.ist-COACH.org/
17. OMG, formal/02-06-65: CORBA Components, v3.0 full specification
18. OMG ADTF: UML testing profile. http://www.fokus.fhg.de/tip/u2tp/
19. OMG: Portable interceptors. TC Document orbos/99-12-02, December 1999.

20. Open CORBA Component Model Platform (OpenCCM) http://corbaweb.lifl.fr/OpenCCM
21. The QEDO project http://www.qedo.org/
22. W3C Note on VML http://www.w3.org/TR/NOTE-VML
23. WDM and IP Network Management (WINMAN) project: http://www.winman.org/
24. Mozilla SVG project, http://www.mozilla.org/projects/svg/