An On-line Test Platform for Component-based Systems[1]

Peter H. Deussen, George Din, Ina Schieferdecker
Fraunhofer Research Institute for Open Communication Systems (FOKUS)
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
E-mail: {deussen, din, schieferdecker}@fokus.fhg.de

Abstract

One of the most provocative research areas in software engineering field is the testing of modern component based distributed applications in order to assure required quality parameters. Dynamic interactions and structural embedding, run-time loadable configurations, and services that can be deployed in arbitrary executions environments results in an increased complexity. Moreover, that the variety of possible states and behaviors becomes unpredictable. Thus, since testing during the development phase is always applied in simulated environments, it is almost impossible to detect faults, which appear under real condition, during production phase of a system.

In this paper, we therefore aim at concepts and methodologies that achieve on-line testing of distributed component based systems in their production phase. In comparison with off-line testing (i.e. testing that takes place during system development), on-line testing addresses particular aspects of the behavior of distributed systems, such as: functionality under limited time and resources available, complex transactions that are performed between components provided by different vendors, deployment, and composition of different services.

1. Introduction

Nowadays, the increased complexity of distributed systems accentuate the need to use modern component based technologies like Java Beans, .Net, or Corba based component implementations, as well as emerging active network architecture technologies. On top of these technologies, diverse services are developed and deployed into execution environments scattered on a network. The matter with that is, on one side the emerging difficulty on managing and controlling of services developed by different service providers and, on the other side, the problems generated by deployment of services onto environments installed on diverse networks and provided by different vendors.

The complex interactions between components can obviously lead to an unpredictable weakness in the performance of the system. Malfunctions or failures inside of any of these components can always occur, fact that may affect the system activity. Testing during the development phase only demonstrate the conformance to the system specification. A challenge is the testing of the whole system in the production phase with the aim to monitor the system in condition of dynamically changes.

We have developed a methodology, called *Online Testing,* for testing under realistic operating conditions, the behaviour of all elements involved in the distributed system. This helps us to determine and control exactly what is happening within the system at any time during its operation.

Distributed application components of the tested system are modified to produce events of interests at the critical points considered at different levels of inspection. The test system collects, manages and interprets the events according to predefined patterns of events derived from the specifications of the system that is being tested. The patterns of the events are described by means of test cases.

To demonstrate experimentally the applicability of our concepts we focused on a case study in which an active network is tested. To implement our online test system some technologies were selected. In this respect, a challenge was to find an adequate technology to implement the online test cases. Even though, the TTCN3 [24] testing language was designed for off-line testing, we found that many concepts and features could be adapted for on-line testing.

In this paper we present our work on defining the concept framework for online testing and a prototype of an online test system able to test online an active network. Section 2 presents other works in this direction. In section 3 our architecture and concepts for online testing are revealed. Section 4 gives details about the data flow between the components of the test system and their interfaces. Section 5 discusses about the case study we defined to evaluate the usefulness, possibilities and limitations of our prototype. Section 6 deals with the technologies we used to enable the functionality of our

[1] This work was partially supported by the EURESCOM project P1108 (Olives).

system. Section 7 concludes with a summary of this work and proposes some new research directions.

2. Related Works

At network level, monitoring systems are widely used to validate different aspects: security, connectivity, vulnerabilities, authenticity, resource availability etc. A large number of tools for network measurements and monitoring are available, but the general problem is that each one instantiates a proprietary architecture and some protocols. In this section, we are interested only in projects, which propose generic architectures and solutions for monitoring and testing able to integrate miscellaneous components communicating over standard interfaces and protocols.

Several projects address the online testing area, at least particularly, for some components of the architecture. In general, other works focused on network level (e.g. monitoring tools), but some attempts were made also at the application level [14] where a flexible and generic approach for a fault management system driven by policies is presented. For Grid environments, the DMF [8] work aims at designing a Grid monitoring system able to do performance analysis and fault detection. They elaborated a general architecture for monitoring and identifying the main components piecing the monitoring system. In [23] [21] the JAMM monitoring system is described, which is an agent-based system that automates the execution of monitoring sensors and collection of event data. Another important project, Dasada [6] proposes an integrated infrastructure to provide assurance, dependability, and adaptability for critical mission systems. The Cadenus [3] project envisions an integrated solution for the creation, provisioning composition and validation of services. The scalability and performance problems are addressed in [21] where an SNMP-based distributed monitoring system for network area is described.

3. Online Testing Concepts and Platform Architecture

When considering the task of testing a distributed system it is necessary to provide a model of the tested system and a definition of the aspects to be tested. Any meaningful definition of *conformance testing* requires a specification of the behavior (or performance) of the *system under test* (SUT), given by a complete system description or — more likely for the particular type of systems considered in this paper — by an (incomplete) set of *use cases* or *test purposes*. In either case, a notion of an entity that is subject to testing activities has to be included in the system specification, usually by defining its interfaces. In classical non-distributed testing, this *black box* is just the SUT itself, but for distributed systems it is appropriate to consider a set of *components under test* (CUTs) interacting by asynchronous or synchronous exchange of messages (communication protocols or e. g. RPI), and run in specified *execution environment*s (EEs).

Conceptually, Online Testing method involves identifying and testing the behavior of the SUT at runtime, behavior that is defined within its requirements. Online testing is an approach for "gray-box" testing since some knowledge about the SUT is obviously required.

One of the most important conceptional needs is to capture the subsequent changes in the state of the SUT. We define online testing in the context of a SUT composed of many execution environments (EE) where services or components (C) are deployed. The life of a component consists of several *phases*. Basic examples of those phases are *deployment*, *operation*, and *removal.* Each phase consists of a number of *steps.* With each step, an event is associated that indicates that the particular step has been performed successfully or not. We call a description of phases and steps of the life cycle of a service a *mission*. Testing of a component means the observation and (in case of unsuccessfully performed steps), the control of missions.

Online testing supposes collecting of a large amount of information about the tested system by means of events. Events are observed by interacting with the system under test in several ways; how this interaction is performed can be characterized by so-called *points of control and observation* (PCO). A PCO is an interface that allows the monitoring of a certain CUT. PCOs can be classified as follows: (1) *interface points,* the communication interfaces of components, (2) *link points*, located at inter-component links to capture message exchange, (3) *test interface points*, special interfaces designed for test (or management) purposes, (4) *environment points*, to monitor the provision of the execution environment for some components (i.e. deployment, communication set-up and removal), and (5) *external points*, to extract information about the executing environment itself.

In classical non-distributed off-line testing, PCOs of type (1) and (3) are used, PCOs of type (2) are used in dedicated set-ups for interoperability testing e. g. for network component evaluation, but since the execution environment of CUT are subject to test activities only in the on-line case, PCOs of type (4) and (5) are novel in our approach.

A conceptual framework for online testing includes: (1) *instrumentation and probing system, (2) online test cases (OTC), (3) system model, (4) event publication and transport* and *(5) configuration controller.* Additional*, (6) gauging mechanism* is used which transforms the collected data into visual data.

In Figure 1 shows the architecture model derived from online testing concepts. Any OTS must instantiate this architecture.

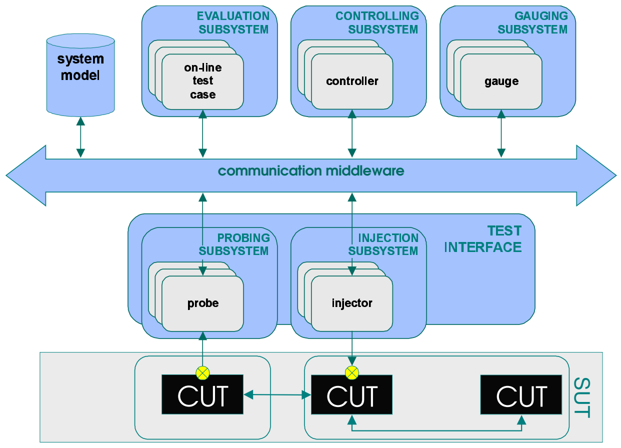

Figure 1. The Architecture of Online Testing Platform. Here are presented the elements of online testing architecture, derived from the concepts framework. Probes interact with SUT and extract informational units relevant for testing purposes. These units are distributed over the event middleware to the other components.

Next, we present the components of the Online Testing Platform. This enumeration is made from the conceptual point of view, since only their specification and functionality is presented and since we do not refer to any underlying technology.

Probes. The observation of a distributed system is one of the main requirements for online testing. Detection of faults, low performance or malfunctions in a large distributed system depends on the quantity and quality of the collected information about the running system. The monitored system must be carefully instrumented with probes in the points where the observations should be made, with code able to extract such information. This task can be done either manually by extending the SUT or by using specialized instrumentation tools.

Probes are, basically, pieces of software dedicated to collect events and information from the SUT about the functional aspects of the components within the execution environment. To improve the performance of the OTS, probes may perform computations on the data collected within the SUT and report only the results of the computation.

There are two types of probes:

Component independent probes which are deployed for monitoring of any component. They address general aspects in the behaviour of the components (i.e. changing of functional phases: deployment, operation etc.).

Component dependent probes which are deployed to monitor aspects particular to a component with a certain functionality (i.e. firewalls).

Injectors. Online testing can be performed *passively* or *actively*. *Passive* stands for testing without affecting the SUT. The data that we evaluate is that already existing in the SUT. In contrast, *active* testing supposes involvement of the Test System in the SUT activity and implies generation of relevant testing data.

Injectors are specialized instruments able to stimulate the SUT so that different aspects of its functionality can be tested. In their absence, any particular functionality aspects cannot be tested in conditions of normal execution. However, introducing test data into the SUT influences its performance and therefore it should be applied carefully.

The most useful way to use injectors is to inject faults in the distributed system. This technique allows the test system to detect such errors under real operating conditions (i.e. services are not available, software components are wrongly deployed etc.).

Online Test Cases. The on-line test cases (OTC) describe behavioral aspects of the SUT by means of certain patterns of events. OTCs validate and control the predefined and expected sequence of events and notify errors in the case of not corresponding behavior.

Elaborating of online test cases for distributed system faces the problems generated by the parallel activities performed in the tested system. In large the behavior of a system is a collection of states and a sequential representation would require all possible interleaving threads, which is practically impossible. Therefore, we aim at identifying only particular behaviors, which are in general considered critical.

Configuration Controller. Some coordination inside the OTS is needed. A controller, that on top of administrator-defined policies manages the activity in the test system, provides this functionality. Technologically, this component is considered a workflow manager, which enforces in the test system the administrator management decisions. It controls also the creation and removal of the other components created on the fly during execution (probes, online test cases)

Communication Bus. A communication bus realizes all the distributed communication functions. It must be able to handle the large amount of events circulating between subscribing components. A few requirements for the communication bus are:

It must provide an easy *subscription mechanism*. The test system consists of a great number of components, which subscribe to the communication bus from different

machines. A component joining the bus is interested only in particular events depending on their content. The bus must be able to distribute all the events of interest to a component.
A component can send a message to a group of components. The communication middleware must provide the concept of group of listeners (e.g. probes, OTSs). In particular, one component may be present by a group containing just this component.

Besides these requirements, one more demand is to have a standard communication protocol. It must consist of flexible and generic communication messages able to transport any type of data.

System Model. Within the OTS some knowledge about the SUT is required. This information is specified statically in a database and is used to configure the OTS. Any changes in the structure of the SUT (either new components are deployed into a node or new execution environments are going to be tested) are updated also in this base. The information stored within this base must cover also configuration parameters of the gauges, test cases and probes.

Another task of the system model is to store information regarding the dynamic behavior of the SUT. This repository manages all the events that are monitored by probes and collected by OTCs. It is very important to store event data, since OTS must be able to make complex historical analysis, evaluate SUT's performance or perform statistical analysis. Sometimes it may be necessary to make predictions against the functionality of SUT in order to deploy new probes.

Gauges. The gauges are software components, which hook into the communication bus and listen to the circulated events, with the main goal to represent and keep up to date an abstract view of the tested system, which may be reported to the service administrators. Since any visualization of the status of the SUT requires also information about its structure, transactions with the event archive and system model bases may also be possible.

4. Interfaces and Data Flow

Having presented the architecture and the conceptual framework, we now present the interaction and data flow among the components.

The flow of the operations made in the OTS starts with the *configuration requests* that are created by the configuration controllers. The configuration controllers permanently interact with the system model base where the management policies and the static configuration of the SUT (i.e. node IP addresses, communication ports of the probe managers of the nodes) are stored. Any changes (i.e. adding of a new node, decision to monitor only certain components or only the components deployed on a particular node) are updated in the whole system by the controller. The configuration of the system is achieved via *configuration requests.* For instance, if the system administrator decides to test a new node, the configuration controller sends a request to the probing subsystem, which deploys the probes necessary to monitor that node. Another request is sent to the OTC component, which starts the test cases meant to test the activity in that node.

Next, the probing system begins to collect data from the probes deployed in the SUT's observation points. Data is delivered to the OTSs in the form of informational units encoded into the internal communication protocol.

The OTSs vary from something simple (evaluation of a few informational units) to complex behavior descriptions with a great number of states. The complex OTSs may need to backup information into the system model component.

The communication function between the Test System components is realized via an event middleware. The components communicate asynchronously by sending and receiving *events,* therefore we need an event middleware, which implements a subscribing mechanism and is capable of distributing the events. Within the system, the same event can be received and processed by more than one component. Hence, the communication middleware, must support message delivery mechanism based on the content of the circulated messages. This way we can group together several components, which are interested in events coming from a certain component.

Since a lot of information must be visualized for the interaction with the administrator, visualization components must subscribe also to the communication bus. These special components are called *gauges* and they continuously retrieve information about the status of the SUT (probed execution environments, probed components, deployment steps of a component, statistically values etc.).

4.1. Probing System

The performance of the test system depends on its ability to monitor the SUT and on the quality of the monitored data. The SUT has a dynamical structure: new execution environments are plugged-in; the components are continuously deployed, started and removed. Consequently, an automated solution is needed in the probing system. Our solution is especially designed for component based SUTs and permits dynamical deployment and removal of probes and their reconfiguration on the fly. It must be also possible to dynamically activate/deactivate probes so that the level of granularity and detail of the probing can be tuned from the configuration controller by using operation policies or even from the test cases, which can solicit more refined data.

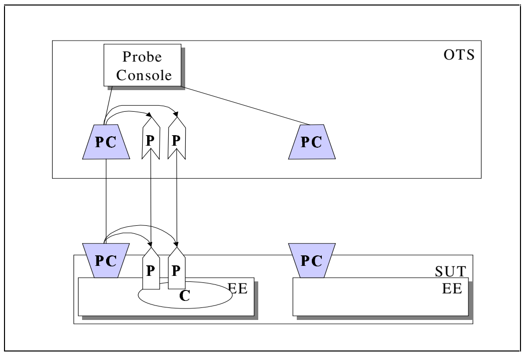

Figure 2. Probe deployment and communication with OTS. The Probe Console starts a Probe Controller for each Execution Environment. Further, whenever a new component is deployed a set of probes is deployed.

The probing subsystem is divided into two parts: OTS part and SUT part. A probe console controls the overall activity in the OTS part of the Probes. To coordinate probing related to components running at one EE, a probe controller (PC) is deployed for this EE. It communicates with its counterpart in the SUT. Whenever a new component is deployed in the EE, these controllers interchange management information that allows the OTS to decide which probes to be deployed in order to monitor this component. The deployment and removal of probes are coordinated by controllers. The probes (P) are also deployed in both sides. Generic messages were defined to provide the transportation of any type of probed data; we were able to define also a standard interface, which is implemented by any probe.

Functionally, probes manage two classes of events: At the interface to the SUT, *low-level events* generated by probe components that instrument the SUT. They can be either general events (emitted by all components or EEs) or specific events (emitted by certain types of components). In opposite to that, *high-level events* are created by the probes by performing pre-evaluations on the probed information, and capsulate those derived data into the internal communication protocol. These data are sorted, stored and interpreted by online test cases.

Probes need to be automatically deployed and removed. The management is a complex task, since one has to deploy probes for many different observation points. Probes can be deployed at the start-up of a component or at request, on the base of some administration policies. Moreover, when the results of a certain probe do not lead to a clear conclusion, the OTS may start other probes to perform tests that are more refined. All these problems related to the probe management are supported by the probe controllers, which continuously retrieve information about the status of the deployed probes and about the changes in the SUT.

4.2. Online Test Cases

The OTCs capture the essence of the information received from probes. They address in general conformance aspects, but also performance of interoperability issues can be considered. OTCs are instantiated for each deployed component within the SUT. Another controller is responsible for their coordination (i.e. start-up, execution or removal) and it is created whenever OTS registers a new component deployed into an EE. Similar to the probing subsystem, an OTC console manages all these controllers. Any test case subscribes to the event bus by creating a filter for the events of interest. From the large amount of events circulated via communication bus, there are selected only those relevant for the testing purpose.

As example, we present a very simple test case written in TTCN3. It requests each 20 seconds the number of packets blocked or passed through a certain firewall (see Section 5 for more information) identified by _active_service_id. The same numbers are requested also for the overall traffic. When both numbers are received then some simple analysis is made. If a certain condition is not fulfilled the configuration controller is notified. For simplification we eliminated some the code that is responsible for connection set-up and internal coordination.

```
// OTC for Operational

testcase OTCO (inout OTCControllerComponent _c,
               inout integer _mission_id,
               inout integer _active_node_id,
               integer _active_service_id)
runs on OTCOComponent system SystemComponent {
    var OTCControllerComponent c := _c;
    var integer mission_id := _mission_id;
    var integer active_node_id := _active_node_id;
    var integer active_service_id := _active_service_id;

    // … connect to communication bus

    // receive data and update the database
    // from time to time make requests for the data
    while (true) {
        var integer blockedTMP := 0;
        var integer timeTMP := 0;
        var integer passedTMP := 0;
        var integer blockedAllTMP := 0;
        var integer timeAllTMP := 0;
        var integer passedAllTMP := 0;

        timer TWait := 20.0;
        TWait.start;
        alt { [] TWait.timeout; }


        var E_ICP_req V_E_ICP_req;
        V_E_ICP_req.id := mission_id;
        V_E_ICP_req.mes.id := mission_id;
        V_E_ICP_req.mes.res_group :=
            "OTCO[" & int2str(mission_id) & "]";
        V_E_ICP_req.mes.rec_group := "Probe";
        V_E_ICP_req.mes.request := "OPERATION_STATUS";
        V_E_ICP_req.mes.argument.tag := "OPERATION_STATUS";
        V_E_ICP_req.mes.argument.req.os_req.mission_id :=
            mission_id;
        PortToSIENA.send(V_E_ICP_req);

        // for all packects now
        V_E_ICP_req.id := mission_id;
        V_E_ICP_req.mes.id := mission_id;
        V_E_ICP_req.mes.res_group :=
            "OTCO[" & int2str(mission_id) & "]";
        V_E_ICP_req.mes.rec_group := "Probe";
        V_E_ICP_req.mes.request := "OPERATION_STATUS_ALL";
        V_E_ICP_req.mes.argument.tag := "OPERATION_STATUS_ALL";
        V_E_ICP_req.mes.argument.req.os_req.mission_id :=
            mission_id;
        PortToSIENA.send(V_E_ICP_req);

        var ICP_res V_ICP_res;
        var ICP_res V_ICP_resAll;
        PortToSIENA.receive(T_ICP_res(T_responseType_os)) ->
            value V_ICP_res;
        PortToSIENA.receive(T_ICP_res(T_responseType_os_all)) ->
            value V_ICP_resAll;

        var integer blocked :=
            V_ICP_res.argument.res.os_res.blocked;
        var integer time := V_ICP_res.argument.res.os_res.time;
        var integer passed :=
            V_ICP_res.argument.res.os_res.passed;

        var integer blockedAll :=
            V_ICP_resAll.argument.res.os_res.blocked;
        var integer timeAll :=
            V_ICP_resAll.argument.res.os_res.time;
        var integer passedAll :=
            V_ICP_resAll.argument.res.os_res.passed;

        // update the DB - with this data; other test cases may
        // use it

        // make some analysis of the data;
        var integer cond :=
            EvaluateData(blocked, passed, blockedAll, passedAll,
                         time, timeAll,  blockedTMP, passedTMP,
                         blockedAllTMP, passedAllTMP)

        if (cond  < 1) {
            blockedAllTMP := blockedAll;
            passedAllTMP := passedAll;
            blockedTMP := blocked;
            passedTMP := passed;
            timeTMP := time;

            // notify Configuration Controller about this error
            // these operations are not included
        }
    }
}
```

4.3. Internal communication protocol

Within the test system the components communicate via events sent or received through the communication bus. The communication is achieved on top of a generic protocol and follows a simple request/response scheme or comprises of a single, unacknowledged indication. To avoid confusion between the events received from probes (high level events) and the events generated by test components, each event has a type. Each request/response pair is associated with a unique identifier, which can be obtained from a number server running within the configuration controller.

The concrete data representation and the processing mechanisms within the probes and test cases are very much dependent on the target system. One important contribution at the content of the messages send/received via this communication protocol bring the selected validation aspects. The protocol we designed is, in large, more like a generic message-based interface since any type of information can be embedded into the transported messages.

5. Case Study

To determine experimentally the validity of the defined concepts, a case study is performed. We experimented our method on testing the deployment and operation of services in an active network architecture. A key feature of active networking is to enable network components to be designed and deployed by third parties. The Caspian active network architecture developed in the EURESCOM project P926 and used in our project as target system aims to produce a proof of the suitability of active networking technology for mobile Internet environments.

The active network model is adequate to our conceptual SUT model. In [22] a conceptual architecture for active networks is defined which may be used here as a reference. An active network consists of active nodes. An active node (AN) comprises of a node operating system and one or more execution environments providing a programming interface or virtual machine that can be programmed or controlled by the active packets. We refer to a process that is deployed and running in the execution environment of an active node as an (active) service. Once deployed, a service is able to perform interactions with other net components (not necessarily other (active) services, if inter-domain scenarios or end systems are considered).

Caspian's architecture is presented in Figure 3. In Caspian an active node consists of active router and active server. The active router receives IP packets from the network and fetches them according to predefined filtering rules. The packets, which pass through the filters, are sent to the active server. Further the packets are delivered to the active services running on the Server (these active services can be started either at boot or at runtime).

The tested component is an active firewall service component, which filters IP packets based on source/destination addresses. Even this simple scenario allows experimenting with on-line testing concepts, including surveillance of the service and logging of success/fault conditions.

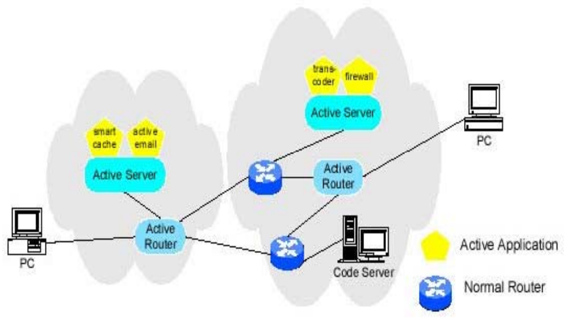

Figure 3. Caspian Active Network Architecture.

In order to couple the test system with Caspian some modifications in the core and non-core functionality were implemented. Nevertheless, the most important enhancement was to provide the probing functionality, which – consequently - in this case was implemented itself by active services. A main service is the probe manager, which communicate with its counterpart in the OTS to report all the changes that occur within the Caspian active node. The probes coordinated by this probe manager are also implemented as services and are dynamically loaded on demand at deployment phase. Both, the probe manager and the probes reside on a code server and can be activated or removed like ordinary services.

The case study instantiates the on-line testing platform conceptual architecture described above by the adoption and integration of several technologies. For online testing of the concrete component, an active firewall, we considered several appropriate test cases:

- *Service not available*. This fault always may occur since the services are usually stored on a web server. Either the web server is not available or the connection to the server is down, are possible causes of this fault. Even these simple scenarios determined us to experiment with testing of availability of invocated services.
- *Not corresponding traffic*: Based on knowledge about the network one can establish particular characteristic values of the circulated traffic. Whenever the traffic does not conform to these values something wrong could be within the SUT (i.e. a firewall blocks or let go through too many packets)
- *Error on loading a service*. Usually, errors on loading a service appear when different resources are not available. This was another affordable case where online testing makes sense. We experimented with the situation where a kernel module is not loaded and consequently a service does not work properly.
- *Errors in the configuration file*: To deploy a service into the Caspian platform, a configuration file is required. It includes information about the location of the service to be deployed and several initialization parameters. Such an error can occur when, for instance, a not available URL is specified. This fault was interesting for us to test particular processing steps of start-up of a service.

6. Prototype Implementation

The software enabling the OTS is structured on three levels:

- *Abstract specification level*: On this level, the behavior of the test system is described in an abstract way by using TTCN3 notation. The complex behaviour and communication operations of the probes and OTCs are specified at this level.
- *Adaptation and execution level*: At this level the abstract behavior is compiled into concrete executable code. This code is deployed into an execution environment, which performs the operations described in the abstract notation. We also implemented an encoder and a decoder for the translation of the data from XML to TTCN3. On this level reside also the components, which enable gauging, controlling or storage capabilities.
- *Connectors level*: Two types of connectors are used: *low level connectors* implemented only in the probe components to communicate with the SUT and *high level connectors* to communicate with the event middleware.

Except for the abstract specifications, all the components are implemented by using java-based technologies. The interchanged messages are describes in XML. As it follows, we describe the technology mapping.

For the system model we used XML based technologies. To parse XML messages we used JAXP (Java APIs for XML Processing) implementation of DOM [7] and JAXEN [13] was chosen to perform Xpath [25] queries for data selection.

The TTCN3 [24] (test and testing control notation, 3rd edition) language is designed to support conformance testing, i.e. testing of a concrete implementation against a (standardized) specification. TTCN3 therefore provides abstraction from concrete data representation by incorporating various APIs to translate from concrete to abstract test data. Powerful pattern matching allows for easy data interpretation at an abstract semantic description level.

In the probing components, we used TTCN3 to describe the data structure for the low and high-level messages, abstract behavior and the interaction with the SUT. TTCN3 was also used for the purpose of data analysis and correlation inside the OTCs. Since TTCN-3 is dedicated to describe test cases and distributed test systems, a system of concurrently running test components is used for analysis purposes.

The communication among different software components enabling the test system is implemented on top of Siena [18] a stand-alone java process that manages the distribution of events to interested parties. Siena satisfy all the design conditions presented in section 4.

With the help of Cougaar [5] an open-source Java-based decentralized coordination infrastructure, we implemented the Workflow management by using the plug-in mechanisms. All important decisions are met in this component configurable with user defined policies.

Gauging is achieved by using the Jahia WEB Portal [12] Several small JSP pages inside the portal help to visualize the most important events related to the activity in the active node. The data is requested from the system model base by using servlets and Java Beans.

7. Summary and Future Work

This paper describes the concepts and mechanisms for online testing of component based distributed systems. At the present time all basic functionality of the OTS components (probes, event archive, system model, otc, configuration controller and gauges) is complete. Our solution proposed and implemented is based on TTCN3, a language dedicated to definition of test cases for off-line testing, and ensure that it can be easily adopted also for online testing.

We are currently working on adding new probing and visualization capabilities for the operational phase. We also intend to improve the performance of our system, a requirement that was not yet considered, by experimenting other available technologies.

The case study on active network services shown that, the on-line testing methods are applicable for validation of several behavioral aspects directly in the production phase of those services. One essential requirement to enable online testing mechanisms is the instrumentation of the target system. Special interfaces must be defined between the SUT and OTS to make possible probing of the data. One step forward in this direction would be to impose to define and implement such interfaces at the pre-deployment phase of the distributed system (if possible to standardize). This will avoid the instrumentation phase, which obviously requires manually intervention when the test purposes regard a deeper knowledge about the SUT.

References

[1] Aide – Active Interface Development Environment, Webside http://www.cs.wpi.edu/~heineman/dasada/

[2] G. Alonso, "Workflow Assessment and Perspective", in International Process Technology Workshop, 1999.

[3] CADENUS: Creation and Development of End-User Services in Premium IP Networks, Website http://www.cadenus.org/

[4] J.M. Cobleigh, L.J. Osterweil, A. Wise, B. Staudt Lerner, "Containment Units: A Hierarchically Composable Architecture for Adaptive Systems", in 10th International Symposium on the Foundations of Software Engineering, 2001.

[5] Cognitive Agent Architecture (Cougaar) Open Source Website, http://www.cougaar.org/index.html

[6] Dasada Home Page. http://www.if.afrl.af.mil/tech/programs/dasada/

[7] Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1, 1998, avail at. http://www.w3.org/TR/REC-DOM-Level-1/

[8] Distributed Monitoring Framework (DMF), Website http://www-didc.lbl.gov/DMF/

[9] N. G. Duffield, M. Grossglauser, "Trajectory sampling for direct traffic observation", SIGCOMM, pp. 271-282, 2000.

[10] Dutton, N.W., C. Wentworth, Ovum Evaluates: Service Management for E-business Applications, Ovum Ltd, February 2001.

[11] Dynamic Assembly for System Adaptability, Dependability, and Assurance (Dasada), Website http://www.if.afrl.af.mil/tech/programs/dasada/

[12] Jahia Portal Server 3.0, Website http://www.jahia.org

[13] Jaxen - Java Xpath Engine, Website http://jaxen.org

[14] M.J. Katchabaw, H.L. Lutfiyya, D. Marshall, and M.A. Bauer, "Policy-Driven Fault Management in Distributed Systems", Proc. of the 7th Int. Sym. on Software Reliability Engineering (ISSRE '96)

[15] G. Kaiser, P. Gross, G. Kc, J. Parekh, G. Valetto, "An Approach to Autonomizing Legacy Systems", in Workshop on Self-Healing, Adaptive and Self-MANaged Systems, 2002.

[16] P926 Project Page at Eurescom, Website http://www.eurescom.de/public/projects/P900-series/P926/default.asp

[17] T.H. Ptacek, T.N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection", 1998, avail. at http://citeseer.nj.nec.com/ptacek98insertion.html

[18] Scalable Internet Event Notification Architectures (Siena) Website, http://www.cs.colorado.edu/users/carzanig/siena/

[19] S.K Shirvastava, L. Bellissard, D. Feliot, M. Herrmann, N. De Palma, S.M. Wheater, "A Workflow and Agent based Platform for Service Provisioning", in 4th IEEE/OMG Int. Enterprise Distributed Object Computing Conference, 2000.

[20] Stump, R., W. Bumpus, Foundations of Application Management, J. Wiley & Sons, 1998.

[21] R. Subramanyan, J. Miguel-Alonso and J.A.B Fortes, "A scalable SNMP-based distributed monitoring system for heterogeneous network computing", IEEE 2000.

[22] D.L Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, "A survey of active network research", IEEE Communications Magazine, vol. 35, no. 1, pp. 80-86, 1997.

[23] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, M. Thompson, "A Monitoring Sensor Management System for Grid Environments", Proc. of the 9th IEEE Int. Sym. on High Performance Distributed Computing (HPDC'00)

[24] Tree and Tabular Combined Notation, version 3, ITU-T Recommendation Z.140, 2001 avail. at http://www.itu.int/ITU-T/studygroups/com10/languages

[25] XML Path Language (XPath), Version 1.0, W3C Recommendation 16, 1999, avail. at http://www.w3.org/TR/xpath