10

A Monitoring and Testing Framework for Critical Off-the-Shelf Applications and Services

Nuno Antunes[1], Francesco Brancati[2], Andrea Ceccarelli[3,4], Andrea Bondavalli[3,4] and Marco Vieira[1]

[1]CISUC, Department of Informatics Engineering, University of Coimbra, Portugal
[2]Resiltech s.r.l., Pontedera (PI), Italy
[3]Department of Mathematics and Informatics, University of Florence, Florence, Italy
[4]CINI-Consorzio Interuniversitario Nazionale per l'Informatica-University of Florence

One of the biggest verification and validation challenges is the definition of approaches and tools to support systems assessment while minimizing costs and delivery time. Such tools reduce the time and cost of assessing Off-The-Shelf (OTS) software components that must undergo proper certification or approval processes to be used in critical scenarios. In the case of testing, due to the particularities of components, developers often build *ad hoc* and poorly-reusable testing tools, which results in increased time and costs. This chapter introduces a framework for testing and monitoring of critical OTS applications and services. The framework includes (i) a box instrumented for monitoring OS and application level variables, (ii) a toolset for testing the target components, and (iii) tools for data storing, retrieval and analysis. We present an implementation of the framework that allows applying, in a cost-effective fashion, functional testing, robustness testing and penetration testing to web services. Finally, the framework usability and utility is demonstrated based on two different case studies that also show its flexibility.

10.1 Introduction

Verification and Validation (V&V) has been largely applied in scenarios that involve life and mission critical embedded systems, and is dominantly used as a design-time quality control process for the purpose of evaluation of the compliance between of a product, service, or system [1]. Checking a system using traditional V&V methods frequently exceeds the effort needed for the core development time. In fact, rigorous V&V in on the fundaments of critical applications and has been applied in several domains as the railway [2] and space [3], and recently a strong effort has been made to standardize these practices for automotive [4].

Although the industry rapidly turns to system integration based on the reuse of hardware and software components, also known as off-the-shelf (OTS) components, it is still necessary to apply rigorous V&V techniques to assess the applications. While hardware OTS are nowadays widely accepted, and used (they have their own certification), software OTS still creates serious difficulties to companies, which are on one hand constrained to meet predefined quality goals, whereas, on the other hand, are required to deliver systems at acceptable cost and time to market. Large companies mainly follow a brute-force approach by focusing large volume investment into tooling and in-house training, but even high-tech SMEs are highly vulnerable to the new challenges.

In this context, one of the biggest challenges to the V&V community is to define **methods, strategies and tools able to validate a system adequately**, while simultaneously keeping the **cost and delivery time reasonably low**. The key part of the challenge is to establish a proper balance between achievable quality with a particular technique (in terms of RAMS attributes) and the costs required for achieving such quality. The problem grows when it is necessary to include COTS components in a critical system that must be certified. As a matter of fact, although modern standards consider the possibility of assessing products, which encompass COTS software, this is still considered a challenge [5].

In industrial practices, *integration* and *usage* of OTS software components in critical systems is generally supported by two different assessment processes, both to understand the behaviour of the component and to assess that it does not introduce hazards in the system. In the first, whenever applicable, the activity is limited to assess the integration, verifying that the OTS component is properly wrapped in the system without affecting system's safety. In the second, a complete assessment of the OTS component is performed; this may include activities as production of documentation, reverse engineering, and static analysis, among others. For companies, this usually means a reasonable amount of **effort in developing a specific tool** that can support the testing of a specific OTS component to be integrated in a certain critical system.

This chapter presents a framework for testing and monitoring critical applications and services. The framework monitors the variables of the system while applying diverse forms of testing over the applications. This way, it is possible to better detect problems in the applications as well as better diagnosing them, maximizing the effectiveness of the tests. The framework is based on an application independent and reusable core infrastructure, allowing the user to apply cost effective practices. The proposed framework consists of two main components, as follows.

The first, named **Instrumented System** is a monitoring environment where the applications or services can be executed and monitored. The kernel of the operating system is instrumented to monitor all the variables that are representative for V&V process. The environment also includes middleware that is also instrumented to provide values of the all the variables representative for V&V at this level. The second component named **Test and Collect** contains a set of tools for application testing and, data storage and analysis. The testing tools included should be able to generate different types of testing including functional testing, robustness testing, security testing, etc. For data storage the framework includes a database management system and tools to allow the user, in a semi-automated way, to generate a schema able to store the values of the monitored variables.

The use of this kind of analysis is essential for the conscious use of OTS components. By testing the OTS, it is also possible to use wrapping strategies [6, 7] around the identified problematic parts of the component. An important part of the implementation is that one instance of this component can be connected to multiple instrumented systems. This way, the framework is prepared to be extended for other purposes, as in the case of monitoring a large scale system with multiple nodes, as it is possible to correlate data from multiple sources and also analyse more complex systems.

Several works have shown the usefulness of system monitoring to detect anomalies in the system. Statistical analysis algorithms have been used in the past for on-line fault detection [8]. This technique overcomes some of the limitations of static threshold analysis, that for instance in [9] monitoring techniques are used to detect application hangs. Works towards certifying OTS components are also not new. The technique [10] tries to determine the quality of OTS components using black box and fault injection in two phases: first, the component is tested to make sure it works properly, and second, the system is tested to make sure that the system works even if the component presents an incorrect behaviour.

The chapter also presents a prototype implementation and demonstration of the framework. The implementation includes tools that allow the user to apply to the web services different types of testing: functional, stress, robustness and penetration testing. During the different testing processes, the system variables are monitored both at middleware level and at operating system level. Two different case studies were devised to demonstrate and evaluate the framework.

The first **case study is focused on the services of the Life ray Portal**, an enterprise web platform project that aims for immediate delivery of robust business solutions for organizations. This case study allowed us to demonstrate the flexibility, usability and utility of the framework. The results revealed the services under test performing quite well in the situations tested. Obviously, the quality of the tests performed depends on the testing tools used, but this discussion is out of the scope of this work, as the merits of each tool were evaluated and discussed in different works by their authors [11, 12].

The second **case study is focused on simulator of a railway** environment that includes a system that should detect anomalous and hazardous situations on the trains running on that line. The stringent requirements of the system that should be tested and validated exactly in the same setup as it will operate demonstrated the flexibility of the toolbox, which was able to be easily ported into such environment.

The chapter is organized as follows. Section 10.2 describes the concepts behind the framework, while Section 10.3 presents the implementation details. Section 10.4 presents the case studies used to demonstrate the framework and the respective details. Section 10.5 concludes the section and puts forward relevant future work.

10.2 Framework Architecture

Our proposal is an advanced framework for testing and monitoring critical applications and services. Despite the most common approach for testing OTS web services is the “black box”, the tool has been designed to take advantages of any piece of information available. The overall architecture of the proposed framework is depicted in Figure 10.1. As it is possible to observe, the framework is based on two main components: i) *Instrumented System*, which the system in which the web service is running, and ii) *Test and Collect*, which is used to stimulated the web service and to collect evidences

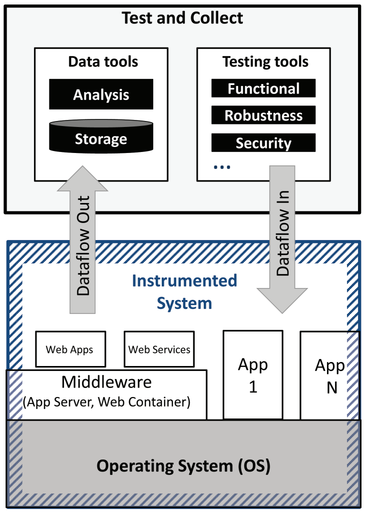


Figure 10.1 Framework architecture: overall view and interactions.

of its behaviour. Although the current implementation focuses on Java Web Services running over a Tomcat 7 Application Server (AS) and a Linux CentOS 6 Operative System (OS), the proposed solution can be evolved to different platforms and Web Services Middleware (AS).

As we can observe, Figure 10.1 also shows the interactions between the two systems of the framework: the testing tool invokes methods of the web service triggering specific functionalities, and at the same time the analysis tools read information on the overall status of the operative system and service middleware. The next sections present the concepts behind each component.

10.2.1 Instrumented System (IS)

The Instrumented System is a monitoring environment where the applications or services can be executed and tested. Considering that weaknesses can affect the middleware layer (e.g., depleting available free memory in the heap) and the operating system layer (e.g., exchanging a huge amount of data or delaying the overall system), both are object of monitoring.

Figure 10.2 represents the monitored components (Operating System, Middleware, Applications) and data flows. The key innovation is to monitor both the variables of the *OS* and of the *Middleware* (when applicable) at the same time the application is being tested. This provides detailed data on the behaviour of the OTS component, thus going beyond the mere collection

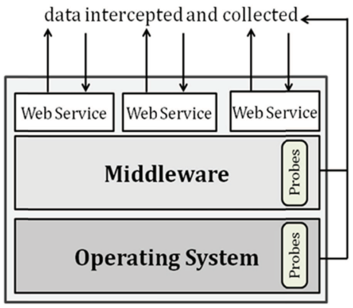


Figure 10.2 Detailed functioning of the Instrumented System.

of inputs and outputs or the monitoring of specific functions of the underlying system that the OTS component uses. To achieve this, it is necessary to instrument the kernel of the OS introducing monitoring probes that report the value of the selected variables per unit of time. These values should be stored in a standard format to later be externalized through the *Dataflow Out (DO)*.

As example of middleware, the environment may include an *application server* where the user can run the web applications and services that are necessary to be tested. Also, the application server includes monitoring of the values of relevant variables that are also stored in a standard format for later use of the *DO*. Due to the emerging role that web applications and services have in critical systems, the inclusion of a monitored application server is a very important requirement, as this allows gathering the values of variables that are closer to the applications under test.

The *Dataflow In (DI)* is necessary to perform the test in the applications. In the case of web applications and services, which have an interface available over the network, the DI In is constituted by the ports used to perform the tests together with OTS components that can execute the tests through these ports. The environment should also be ready to support the testing of other applications, with the Dataflow having the responsibility of translating the tests created by the testing tools in a form that can be executed in the target application. In practice, the Dataflow In represents the only part of Instrumented System that the user should implement in order to have his application tested.

10.2.2 Test and Collect

Test and Collect includes a set of extensible tools that should support the user in two main activities: (i) *testing*, and (ii) *storage and analysis*. The **testing** component controls the execution of the testing tools. Although the framework is designed to be fully automated, the human interaction cannot be completely avoided at least for the test execution.

The level of human interaction can vary from test to test; thus, each testing tool should provide its own interaction interface. It is mandatory that the testing tool communicates with the **storage and analysis** module to trace testing activity, providing information as test input/output, and executions results and durations that should be logged by the storage module to match the results provided by the IS during the execution of these tests. Figure 10.3 depicts this relationship, which is detailed below.

The **storage and analysis** module is also in charge of harvesting data from the IS probes and of structuring and storing them to facilitate the subsequent data analysis. The storage component is made up of three modules: (i) *Probes Collector* (PC), (ii) *Data Manager* (DM), and (iii) *Database* (DB).

The PC is responsible of reading data from IS probes and due to the different sources (middleware or OS) it needs to use different policies to respect the data availability and probe servers' constraints. Data read are then managed by the DM component that organizes the data coming from middleware structuring it to provide the state of the monitored system from a specific point of view.

Finally, data is stored in the underlying database. To provide more efficient data analysis capabilities, the template schema follows the model of a star schema from OLAP. In fact, a well-structured data repository and OLAP analysis can be very useful for analysis of results from dependability evaluation experiments [13]. Additionally, it makes possible to share and compare the results of multiple different experimental evaluations [13].

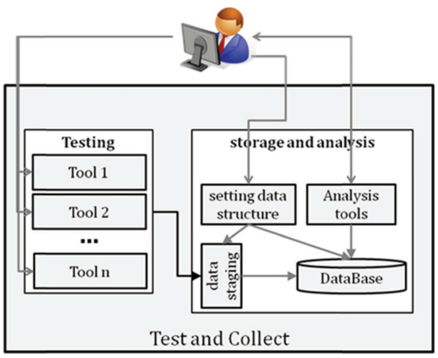


Figure 10.3 Detailed functioning of the Test and Collect.

The **testing tools** included should be able to generate different types of testing including functional testing, stress testing, robustness testing, penetration testing, security testing, etc. The definition of tests is always dependent on the type of application as well as specific to the domain of the application (e.g., testing requirements from standards). In the case of web applications and services, where the interfaces are usually well defined, the test generating tools usually require only minor configuration. However, in the case of other applications the user may be requested to configure or even extend the testing tools. To cope with this, the tools included are prepared to be easily extensible to accommodate the user needs. As the tools are easy to modify or replace, the framework provides high flexibility and makes it easier to test applications.

Functional testing is a black box testing technique that tries to find discrepancies between the program and the external specification [14] and it is based on a set of test cases derived from the analysis of the specification. Stress testing subjects the program to heavy loads or stresses [14]. In this case, the testing application must submit loads that match (or even surpass) the load that the application under test is specified to sustain over a period of time. This is particularly useful in web-based applications where you want to ensure that your application can handle a specific volume of concurrent users or requests. Robustness testing is a specific form of black-box testing. The goal is to characterize the behaviour of a system in presence of erroneous input conditions. Robustness testing stimulates the system in a way that triggers internal errors, exposing programming and design errors both in the error detection and recovery mechanisms. Penetration testing is a specialization of robustness testing that consists of the analysis of the program execution in the presence of malicious inputs, searching for potential vulnerabilities. Penetration testing tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type.

Finally, the toolset should be allowed to be used several data analysis algorithms; including fault detection mechanisms based in static threshold analysis algorithms and also statistical analysis algorithms. However, the main idea is to leave to the user the conditions to perform the analysis using the algorithms that he is more experienced with and, above of all, that are most adequate to his business domain. In fact, one strength of the use OLAP analysis techniques is the optimization of their schema for the use of *ad-hoc* queries [13].

10.3 Implementation Details

To show the applicability of the approach and perform an experimental evaluation, a prototype was designed and is currently under development. For cost reduction and to allow bigger flexibility, effort was made to use low licensing cost solutions resulting many times in preference for free or open source software. It is important that, in many cases, the selection of one technology to use impacts the technologies for other layers. The next sections detail the status of implementation of each one of the components of the framework as well as the technologies selected to use.

10.3.1 Instrumented System (IS) Implementation

The node component was implemented in a virtual machine. This option for virtualization provides flexibility as can be easily replicated and maintained. This will allow the deployment of the node ready to use in any number necessary for the system. The operating system selected to implement the prototype was CentOS [15]. First we narrowed our options to Linux based distributions due to the cost advantage and to the diversity of monitoring options to monitor the kernel events. From the multitude of options available, CentOS provides a free enterprise class OS.

The instrumentation of the operating system was implemented in the form of a loadable kernel module using the SystemTap tool [16]. This tool builds on and extends the capabilities of the *kprobes* [17] kernel debugging infrastructure and allows to program breakpoint handlers using a high-level scripting language that is later translated into C code. This way, SystemTap simplifies the development of system instrumentation and also improves the reuse of existing instrumentations, thus allowing building up on the expertise of others. The developers of SystemTap also took into consideration the portability and safety concerns, both of which have major importance in this work.

The prototype implementation of Instrumented System includes, as example of middleware, an application server with monitoring capabilities to allow testing the web applications and services. The selection of choice was JBoss Application Server (JBoss AS) [18], an application server that implements the Java Platform, Enterprise Edition (Java EE). It is free and open source software available under the terms of the GNU LGPL and it is written in Java and as such is cross-platform: usable on any operating system that supports Java. JBoss represents the industry *de facto* standard for deploying Java-based Web applications, it has a wide community acceptance, and support subscriptions can be purchased. For monitoring the values of the applications running inside the app server, it is used Java Management Extensions (JMX) Technology. JMX provides the tools for building distributed, modular and dynamic solutions for monitoring devices or applications. It is designed to provide high flexibility both for legacy systems and for the future. JMX is supported by the most relevant Java application servers. This will allow in the future, porting the monitoring solution to other application servers, and it was one of the requirements as it is planned to add other servers to the prototype to provide a broader range of options and compatibility to the user.

The implementation of the *Dataflow In* depends greatly on the applications to be tested. For the case of web applications and web services, OTS components together with the ports that allow the network traffic constitute the *Dataflow In*. For instance, in the case of web services, the toolset includes the open source tool *soapui* [19], that is the visible part of the Dataflow In for the user. This tool allows to easily executing user-defined tests in the web services under test. In other cases, where the test execution such be from inside the testing machine, it is under development a daemon to run in background accepting communication through TPC sockets and performing the required tests.

Finally, the dataflow out was implemented as folder where the monitoring systems can write the files and from where the Test and Collect can pool the files periodically. The data is split in chunks, each file containing the data respecting to a certain period of time that is identified in the filename. There are many ways to extract the files from the exterior, but the solution currently adopted consists of using *secure copy* (scp), is a protocol to securely transfer files between two hosts, based on the *SSH* protocol. This is a preliminary implementation that will be enough for experimental evaluation but as a more automated solution is under development and should replace it in a near future.

10.3.2 Test and Collect Implementation

The Test and Collect includes a set of testing tools ready to use. Most of these tools are black-box tools, and as the name shows, these tools view the program as a black box and are completely unconcerned about its internal behaviour. Our framework by analysing the values of the monitored variables uses information about the behaviour of the application in a transparent fashion to the testing tools. Most of the tools included also target web services, one of the main targets of the framework, as they are increasingly used in business-critical systems. They provide always a well-defined interface, allowing an easy use of automated tools. Other types of tools and also targeting other will be added in future versions of the framework. In very specific domains, the user will need to write the necessary tests and implement the necessary tools to use them.

The testing tools currently available allow performing functional and stress test, penetration test and robustness test. The testing framework has been developed minimizing the human interaction especially during the testing activities. With the present tools, the human interaction is indeed focused in the configuration phase, which must be performed one time for each tool. Such tools can provide common configurations as well as they can propose a configuration that suits the testing needs.

10.3.2.1 Functional and stress testing

Functional test is a quality assurance process based on black-box approach that aims to provide a proof of implementation correctness regarding the specifications of the software under test. The test is performed feeding the software under test with well-known values and examining the output produced.

Although common functional tests involve the test of single methods, within this context, it has been followed an approach which tackles high-level functionalities. The approach consists of a set of workflows that mimic the behaviour of a software user for executing specific high-level tasks, which in turn can comprise the invocation of a huge variety of methods [20].

The workflows definition is a cornerstone of this approach and it must be defined specifically for each service under test considering its interface and the software specification. Workflows define how and when the service interface of the software under test should be questioned and, also, they provide the information needed for the subsequent phase of result validation. Following the black-box approach the verification is done invoking specific methods of the service for checking its internal status.

The importance of these workflows is further emphasized since they can be used as bricks for compound and complex workflows for Stress testing. Stress testing is a form of deliberately intense testing used to determine the stability of a given system. The tool developed for functional test, properly configured with suitable workflows, can stimulate the system under test to provide evidence of stability. Workflows for Stress testing have been defined from the high-level tasks identified for the functional tests by parallelizing multiple high-level tasks invoked from a variety of users and abbreviating to the minimum the delay between sequential invocations.

10.3.2.2 Robustness testing and penetration testing

Robustness testing is a specific form of black-box testing that attempts to characterize the behaviour of a system in the presence of erroneous or unexpected input conditions [21]. The tool instrumented in the testing framework implements the technique proposed in [22]. The approach consists of a set of robustness tests that is applied during execution to disclose both programming and design problems.

The set of robustness tests is automatically generated by applying a set of predefined rules (see detailed list in [22]) to the parameters of each operation of the web service during the workload execution. An important aspect is that rules focus difficult input validation aspects, such as: null and empty values, valid values with special characteristics, invalid values with special characteristics, maximum and minimum valid values in the domain, values exceeding the maximum and minimum valid values in the domain, and values that cause data type overflow. The robustness of the web services is characterized according to the failure modes adapted from the CRASH scale.

Penetration testing is nowadays one of the most used techniques by web developers to detect vulnerabilities in their applications and services. This technique assumes particular relevance in the web services environment, as many times clients and providers need to test services without having access to the source code (e.g. when testing third-party services), which prevents the use of more effective techniques that require that access. The tool instrumented in the testing framework implements a technique targeting the detection of SQL Injection vulnerabilities in web services. The tool was originally presented in [11].

10.3.2.3 Data storage and analysis tools

A tool is under development for the generation of the star schema according to the needs of the user. This tool, based on the template star schema provided, and after some configuration by the user, generates the schema that will store the monitored data. This tool will also include capabilities to perform the extraction, transformation and load (ETL) of the data. It will allow the user to retrieve the data from the Instrumented Systems using the *Dataflow Out* channel and insert the data in the schema. With a wide range of options for ETL software available, the option for developing a new tool comes from simplicity reasons: it would be an exaggeration to use a heavy ETL tool while our toolset only needs for a very specific and simple tool that designed to work based on a the configuration that the user provides when creating the star schema for the DBMS. Also, the use of third party ETL tools would most probably require the user to have knowledge on how to operate them.

A myriad of solutions are available to implement the data storage. PostgreSQL[1] is an open source solution with a long history of development a proven architecture with recognized reputation for reliability, data integrity, and correctness. It gives to the framework great flexibility as it runs on all major operating systems, including Linux, UNIX, and Windows. A lot of tools from the community support PostgreSQL, and it is also widely supported by open source business intelligence tools as SpagoBI[2] and Pentaho community edition[3].

As aforementioned, in terms of data analysis, the main goal is to provide the user the best conditions for the execution of the analysis of his preference. With this goal, the tool set includes basis tools for data visualization and query execution. The toolset will also include the more advanced tools as the mentioned BI tools (SpagoBI and Pentaho CE) as well as other options that are also considered to be included in the toolset [23]. For better analysis, it is necessary that the data is correlated to the tests executed, and this is a very important part of the ETL process. Finally, the toolset will include ready to use tools that use some more specific algorithms targeting fault detection, proposed by research community. Examples of these works are static threshold analysis [9] and statistical analysis algorithms for on-line fault detection [8].

10.4 Demonstration

Two case studies were devised to demonstrate the applicability and feasibility of the approach.

The first case study is presented in Section 10.4.1 and uses the Life ray Portal [24], which is an enterprise web platform project that aims for immediate delivery of robust business solutions for organizations. An API based on SOAP web services is provided, containing a diverse range of functionalities. These services are an interesting case for testing the framework, after the framework is deployed on top of the instrumented JBoss AS middleware.

[1]www.postgresql.org

[2]www.spagobi.org

[3]http://community.pentaho.com

The second case study, presented in Section 10.4.2, is based on the PMF simulator. SHAPE is a system installed along a specific railway line. The main purpose of the system is to automatically detect anomalous and hazardous situations on the trains running on that line. SHAPE aims at detecting two specific situations: i) SHAPE can detect fires on board a train, through reading at a distance of the temperature of the external surface of the trains; ii) it is able to detect possible violations of the reference shape, through specific laser scanners, in order to identify any dangerous protruding part of the train.

10.4.1 Case Study: Life Ray Web Services

Life ray is free and open sourced Java software that was initially developed to provide an open source enterprise quality portal. Since the early stages of development, Life ray has been widely adopted for intranet as well as extranet enterprise solution. Eventually, it brought Life ray to have a big supporting community, which, together with the Life ray foundation, contributed to define a generic and extendible product.

Life ray Portal is an enterprise web platform project that aims for immediate delivery of robust business solutions for organizations. It includes features that are usually necessary for the development of websites and portals, as built-in web content and document management system. Life ray is developed using Java technologies and it is ready to work in a large set of web/application servers. In fact, the community edition is free and open source software available under Licensed under the terms of the GNU LGPL. It follows an extendible architecture by plugins that, in turn, encompass collaboration, social networking, and single sign on, per-component privileges policy as well as e-commerce tools. Third-party plugins are also available to provide more advanced feature such as Microsoft office integration. A plugin can be seen as a J2EE-Servlet and is referred to as a portlet. Portlets communicate with each other using the services that each one exposes that, in turn perform the portlets business logic. Our installation of Life ray includes the version 6.0 of the portal and 83 deployed SOAP (Simple Object Access Protocol) web services. More details on Life ray can be found in [24].

10.4.1.1 Tests performed

During this case study, different types of tests were applied: Functional, Stress, Penetration and Robustness tests. To verify the correctness of Liferay services, a study on its plugins interaction has been conducted. The necessity for the preliminary study has been felt because of the strictly correlated invocations among methods exposed by web services.

The preliminary study has been exploited to define workloads that could mimic the behavior of Life ray internal interactions, which correspond to **functional tests**. Even simple activity, like posting a message on the blog by UI interface, could involve a plethora of plugins including authentication, user information retrieving, permission checking and finally messaging service. To stimulate Life ray in a way that could resemble human activities, many Workloads have been defined to cover Life ray functionalities by mimicking the behavior of human interaction. Mimicked actions encompass posting a message in the blog and in the forum, creating an event in the calendar, creating a directory-tree in the file repository and uploading a file in it.

These workloads have been used to define other workloads for **stress tests,** to highlight possible weakness in terms of concurrency management. Those tests have been designed from the workloads defined for functional tests to evaluate Life ray behavior under a heavy load. For each workload, that mimics a specific action, a new one is defined as a composition of many copies of the same workload; these copies differ just for the user. The purpose of this approach is to simulate multiple users' activities on Life ray, that stimulate the same services and some shared data. The stress test, as it is designed, suits especially well when there isn't a sound knowledge of web services internal mechanism; the deeper is the knowledge of web service internals the more effective workloads can be designed.

Regarding **robustness testing**, due to the preliminary study performed for the functional test, a generic knowledge of methods invocation was available to configure the tool to generate better tests. This knowledge was especially important for the tool to use values that exercise the code of the web service under test in a more complete way. After the configuration of the tool it submits the robustness tests in an automated way and reports the robustness problems found.

As for **penetration testing**, the available knowledge of methods invocation was a key to configure the tool to generate better workloads and attack loads. This is especially relevant for this tool as its effectiveness is depending on the completeness of its workloads. After this configuration, the test execution is a straightforward process in which the tool submits its workload and attack load to the web service and then reports the vulnerabilities found.

10.4.1.2 Tests results

Experiment execution is made up of three phases, where just the final one is specific for the kind of test to be conducted. The phases are:

1. Set up Service Under Test (SUT),
2. Data Logger execution,
3. Testing tool execution.

On the first phase the service under test is started up. This phase includes also the startup of middleware and OS probes. The second phase can be launched simultaneously, as it does not read information from OS or middleware probes: it just prepares the structures needed for logging. During test execution, the testing framework logs raw tests results and prints on the console information on the tests execution (test currently running, test duration, etc.). This information is useful to monitoring the tests execution. Test results are collected during test execution; at the tests termination, collected data are flushed into a database.

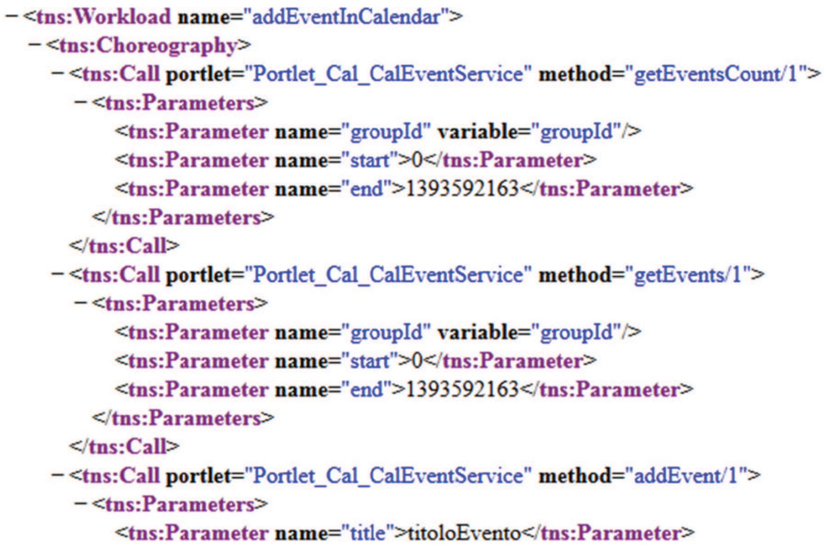
Different tools that range from very specific tools such as R or MatLab to commonly available and general-purpose tool such as OpenOffice-Calc are installed on the Test and Collect system, connected to the database, and can be used to retrieve and analyze data.

Due to the wide usage and the extended support that Life ray received from its community since its development started, it was expected that Life ray passed all the functional tests defined.

Figure 10.4 shows an extract of the workload (set of services invocations) used for creating a new event on a calendar (it includes logging in to the system, listing the available/subscribed calendars, choose to first one and listing all the events of February, adding the new item then logging out; subsequently further invocations checks that the event was correctly recorded). The invocation correctness is verified by a visual inspection of Life ray services.

The output produced during the execution of the test is displayed on the screen of the Test and Collect system and consists of a sequence of services methods invocations. For each one of these, the HTTP response code is printed out. Being a functional test is mandatory that the HTTP response code would match with the expected one.

The aim of this **stress testing** is to assess the ability to resist against a workload which leverage on high frequency of requests, and the results can be evaluated in term of system loads and resource usage. Table 10.1 shows the average CPU usage and memory usage; the former one is furthermore



```
- <tns:Workload name="addEventInCalendar">
  - <tns:Choreography>
    - <tns:Call portlet="Portlet_Cal_CalEventService" method="getEventsCount/1">
      - <tns:Parameters>
          <tns:Parameter name="groupId" variable="groupId"/>
          <tns:Parameter name="start">0</tns:Parameter>
          <tns:Parameter name="end">1393592163</tns:Parameter>
        </tns:Parameters>
      </tns:Call>
    - <tns:Call portlet="Portlet_Cal_CalEventService" method="getEvents/1">
      - <tns:Parameters>
          <tns:Parameter name="groupId" variable="groupId"/>
          <tns:Parameter name="start">0</tns:Parameter>
          <tns:Parameter name="end">1393592163</tns:Parameter>
        </tns:Parameters>
      </tns:Call>
    - <tns:Call portlet="Portlet_Cal_CalEventService" method="addEvent/1">
      - <tns:Parameters>
          <tns:Parameter name="title">titoloEvento</tns:Parameter>
```

Figure 10.4 An extract of the workload to set a New Calendar Event.

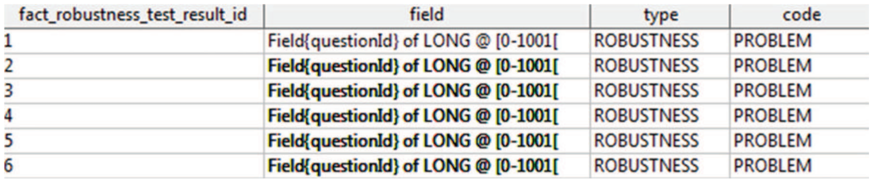
Table 10.1 Extract test results for New Calendar Event

Parallel Requests (n_r)	Process CPU Load (%)	System CPU Load (%)	System Load Average	Free Heap Memory (B)	Free Non Heap Mem (B)	IO Written/ Read Data (B)
5	0.3031	0.38245	1.54	100128920	24899588	5959
10	0.1254	0.651	1.195	86411094	22482534	77344
100	0.1342	0.999	2.34	84833658	21993838	184244

detailed distinguishing between process CPU load and system CPU load, with system CPU load that encompasses any task running on the system. Memory usage is furthermore detailed as well distinguishing between heap memory, used for java objects and non-heap memory.

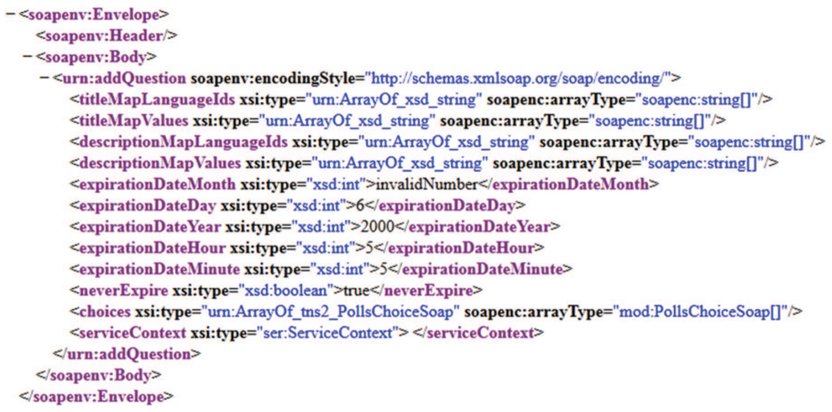
The table encompasses the experiments of 5, 10 and 100 simultaneous execution of the “New Calendar Event” workload. Data are collected 1 times per second. Table 10.1 shows that the CPU usage of service process remains quite stable despite the increase of the number of requests. Process CPU load, the system load as well as memory usage vary as the number of parallel requests increase. The table shows that system resources usage clearly increases due to the waits for Disk Output activities, which rise.

Figure 10.5 shows an extract of the **robustness test** report, in which all the tests reported robustness problems. This would suggest weakness in the services, but a manual inspection revealed that while tool reports "PROBLEM", the service correctly identify and discard the invalid request. We explain this with the help of Figure 10.6.



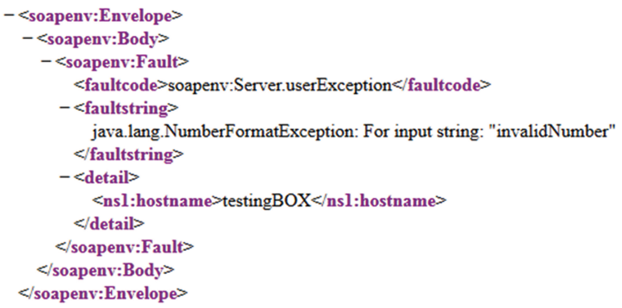
fact_robustness_test_result_id	field	type	code
1	Field{questionId} of LONG @ [0-1001[	ROBUSTNESS	PROBLEM
2	**Field{questionId} of LONG @ [0-1001[**	ROBUSTNESS	PROBLEM
3	**Field{questionId} of LONG @ [0-1001[**	ROBUSTNESS	PROBLEM
4	**Field{questionId} of LONG @ [0-1001[**	ROBUSTNESS	PROBLEM
5	**Field{questionId} of LONG @ [0-1001[**	ROBUSTNESS	PROBLEM
6	**Field{questionId} of LONG @ [0-1001[**	ROBUSTNESS	PROBLEM

Figure 10.5 Extract from robustness test results.



```
- <soapenv:Envelope>
    <soapenv:Header/>
  - <soapenv:Body>
    - <urn:addQuestion soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <titleMapLanguageIds xsi:type="urn:ArrayOf_xsd_string" soapenc:arrayType="soapenc:string[]"/>
        <titleMapValues xsi:type="urn:ArrayOf_xsd_string" soapenc:arrayType="soapenc:string[]"/>
        <descriptionMapLanguageIds xsi:type="urn:ArrayOf_xsd_string" soapenc:arrayType="soapenc:string[]"/>
        <descriptionMapValues xsi:type="urn:ArrayOf_xsd_string" soapenc:arrayType="soapenc:string[]"/>
        <expirationDateMonth xsi:type="xsd:int">invalidNumber</expirationDateMonth>
        <expirationDateDay xsi:type="xsd:int">6</expirationDateDay>
        <expirationDateYear xsi:type="xsd:int">2000</expirationDateYear>
        <expirationDateHour xsi:type="xsd:int">5</expirationDateHour>
        <expirationDateMinute xsi:type="xsd:int">5</expirationDateMinute>
        <neverExpire xsi:type="xsd:boolean">true</neverExpire>
        <choices xsi:type="urn:ArrayOf_tns2_PollsChoiceSoap" soapenc:arrayType="mod:PollsChoiceSoap[]"/>
        <serviceContext xsi:type="ser:ServiceContext"> </serviceContext>
      </urn:addQuestion>
    </soapenv:Body>
  </soapenv:Envelope>
```

(a)



```
- <soapenv:Envelope>
  - <soapenv:Body>
    - <soapenv:Fault>
        <faultcode>soapenv:Server.userException</faultcode>
      - <faultstring>
          java.lang.NumberFormatException: For input string: "invalidNumber"
        </faultstring>
      - <detail>
          <ns1:hostname>testingBOX</ns1:hostname>
        </detail>
      </soapenv:Fault>
    </soapenv:Body>
  </soapenv:Envelope>
```

(b)

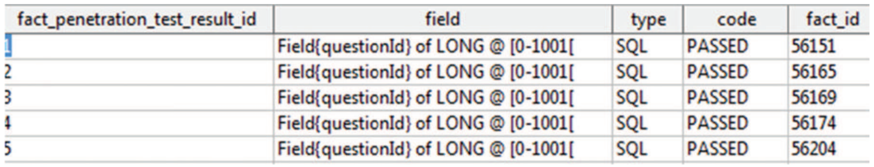
Figure 10.6 Example of robustness test: (a) request; (b) response.

Figure 10.6(a) shows an extract of a robustness test involving the Poll Service, in particular the "add Question" method. Life ray, relying on Axis2 for parsing values, automatically manages the invalid value for the parameter "expiration DateMonth" rejecting the request and without passing it to the "actual" service. The rejection causes an HTTP 533 (which belongs to the "internal error" family): the tool used for robustness testing, operating at black-box, can't distinguished this answer from any other internal error, and consequently the "PROBLEM" code is displayed in Figure 10.6(b) which shows the response that Life ray produces for the request.

Life ray uses Axis2 for service publishing and interface, Axis2 is responsible for parsing values passed by SOAP as well as for invoking the actual Java method which was remotely requested. The parsing phase consists also of a validation phase in which the parsed values are validate against their destination types constraints. The failure of this phase implies the subsequent rejection of the request and thus the generation of a response with HTTP code 500.

Figure 10.7 shows an extract of the results of the **penetration tests** applied to Life ray Calendar Service. The extracted data, as well as the entire test results, show the robustness of Life ray against penetration attacks. All the potentially risky requests are identified and discarded by the Axis2 Layer for services interface, by the Object Relational Mapping (ORM) layer for objects persistency and by the permission checking mechanism, which constitute a cornerstone for Life ray services interoperability.

In fact, Life ray exposes its services using Axis2, which validates the invocation parameters before passing the request to the "actual" service. Additionally, Life ray relays upon Hibernate (the ORM used) which provides an SQL parameter sanitizing service, which in turn it uses named queries that work on top of statements of the JDBC API; all those layers operate the necessary actions to avoid risks from malicious requests. Finally, the invocations that include items the user is not authorized to use are identified by the Life ray Permission Service.



fact_penetration_test_result_id	field	type	code	fact_id
1	Field{questionId} of LONG @ [0-1001[	SQL	PASSED	56151
2	Field{questionId} of LONG @ [0-1001[	SQL	PASSED	56165
3	Field{questionId} of LONG @ [0-1001[	SQL	PASSED	56169
4	Field{questionId} of LONG @ [0-1001[	SQL	PASSED	56174
5	Field{questionId} of LONG @ [0-1001[	SQL	PASSED	56204

Figure 10.7 Calendar Service penetration tests result.

10.4.2 Case Study: SHAPE

We used a second use case to show the flexibility of the approach and also to demonstrate that the approach is not depending on the concrete technological implementation. This use case was based on SHAPE, which is a system installed along a specific railway line. SHAPE has the requirement that the all the tests to be performed must be done in an environment certified as equivalent to the target environment.

The main purpose of the system is to automatically detect anomalous and hazardous situations on the trains running on that line. In particular, SHAPE aims at detecting two specific situations: i) SHAPE is able to detect fires on board a train, through reading at a distance of the temperature of the external surface of the trains; ii) it is able to detect possible violations of the reference shape, through specific laser scanners, in order to identify any dangerous protruding part of the train.

SHAPE was designed to be suitable for interfacing with existing signalling systems, thus to send possible alarms useful to safely stop the train and to properly manage the critical detected event, according to the foreseen recovery actions. SHAPE is composed by the components: Scanner, Init & Diagnosis, Data Acquisition, Data Aggregator, Data Analyser and Monitor, System State.

- Init & Diagnosis – communicates with the scanner in order to collect diagnostic data and to trigger scanner activation.
- Data Acquisition – receives raw data from scanners.
- Data Aggregator – receives train data from Data Acquisition (e.g. images produced by the scanner) and aggregates such information, to be sent to the Monitor component.
- Data Analyser – receives aggregated data from the Data Aggregator and send analysis results to the Monitor component.
- Monitor – manages all the system states phases according to the data received from the Shape Component.
- System State – acquires information regarding the system state from each component and sends them to the WaySide component.

10.4.2.1 Monitoring environment adaptation

SHAPE has stringent requirements whichneeds to be tested in the same operating system, configuration, and equivalent hardware that it is supposed to be used in the future. Therefore, to be able to use the monitoring and testing approach in together with SHAPE, it was necessary to port the monitoring facilities to the target system and configuration.

The new system uses a different operating system and due to criticality restrictions, it cannot have new packages installed, as the system monitoring tools (SystemTAP) required by used in the implementation of the Instrumented System. Therefore, the challenge was to implement similar monitoring functionalities with less intrusive solutions.

The solution used the following tools, which are present in most of unix and linux distributions:

- top – provides data about cpu and memory usage;
- mpstat – provides data about system load;
- iostat – provides data about I/O usage;

Another tool was necessary because the SHAPE simulator involves a set of processes and subprocesses that are continuously evolving.

- pstree – allows to track the processes and the respective process tree, so it is possible to gather data about all the processes relevant for the monitoring system.

The downside of this solution is the performance. In practice, although less intrusive than the SystemTAP solution, it takes much more time to obtain data, and therefore it does not allow small gathering windows. However, we believe that the window is still small enough to do fine grained analysis of the system behavior.

10.4.2.2 Tests performed

To demonstrate the solution, we executed the SHAPE simulator during 24h while monitoring the relevant variables of the system. During this period, the simulator was exercised using the test cases available to test the correctness of his responses. At the same time, the newly included probes seamlessly monitored the variables of interest. Table 10.2 contains the summary of the most relevant variables monitored during the period.

All the variables were analyzed are stored for each sampling instance. This allows us to do temporal analysis of the variables. Figure 10.8 presents the evolution of one specific variable over time, in this case the “Number of SHAPE processes”. As we can observe, the amount of CPU usage keeps increasing throughout the collection period, but still in relatively short values.

Table 10.2 Summary of the variables monitored

Variable	AVG	STDV	MIN	MAX
Total User CPU	0.14	0.08	0.00	0.30
Total System CPU	0.24	0.12	0.00	0.50
Average User CPU	0.14	0.07	0.02	0.26
Average System CPU	0.25	0.12	0.03	0.45
Average IO Wait CPU	0.11	0.00	0.11	0.12
Memory Used	837945	32356	782680	903032
Memory Free	1031399	32356	966312	1086664
Memory Cached	188067	10077	168972	205268
Swap Used	0.00	0.00	0.00	0.00
Swap Cached	582445	20468	547568	618732
IO Disk Read Per Sec	1.78	0.03	1.74	1.83
IO Disk Write Per Sec	7.15	0.22	6.50	7.52
IO Disk Read	2913344	30	2913256	2913424
IO Disk Write	11674444	527703	10347224	12589656
# of SHAPE Processes	2.98	0.17	1.00	4.00
Number of Samples	*151146*			

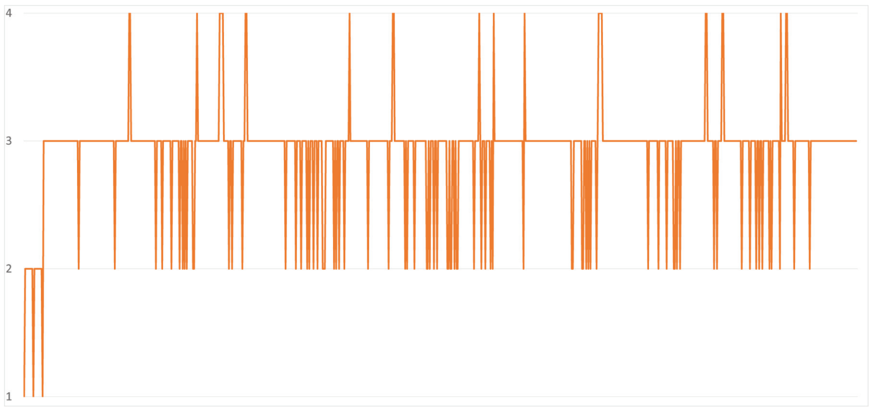


Figure 10.8 Evolution of Number of working processes in SHAPE.

10.5 Conclusion

In a context where OTS components are increasingly used on critical scenarios, companies need tools that help them to understand the quality of these components. In specific cases of testing, rather than using their own developed ad-hoc and poorly-reusable testing tools, these companies can benefit from using cost effective techniques and tools.

This chapter presented a reusable and adaptable framework for testing and monitoring of critical OTS applications and services that includes an instrumented box for monitoring OS and application level variables, a testing toolset that is adaptable for testing the target components, and tools for data storage and analysis. The architecture of the framework was described as well as the status of its implementation.

The framework allows users to easily apply functional testing, stress testing, robustness testing and penetration testing to their web services. The procedure to use the framework is described and its usability is illustrated with a case study that uses the Life ray platform, composed of several web services. The case study shows how flexible is the framework, allowing integration of multiple third party tools seamlessly. Obviously, in the case of functional testing, it is necessary to conduct some preliminary study to emulate its use cases, but this is expected due to the nature of the tests. The framework can orchestrate the use of the tools and reduce the human effort by reutilizing the information provided at configuration time within multiple tools.

The concepts behind the framework can also be extended to setups that differ from the ones defined in the framework implementation. This was demonstrated in the second use case, in which the concepts.

Future work includes the integration of failure detection and prediction algorithms in the box. Additionally, the framework can be modified to use more than one Instrumented System at the same time, allows testing more complex systems. Finally, it can be extended to take advantage of other kinds of information monitoring.

References

[1] Tran, E. (1999). "Verification/Validation/Certification," in: *Topics in Dependable Embedded Systems*, ed. P. Koopman. Carnegie Mellon University, Pittsburgh, PA.

[2] IEC. (1998). *IEC 61508 TC: IEC 61508, Functional Safety of Electrical/ Electronic/Programmable Electronic (E/E/PE) Safety Related Systems, Part 3: Software Requirements*. IEC, Geneva, Swiss (1998).

[3] RTCA. (2011). *RTCA: RTCA DO-178C/EUROCAE ED-12C – Software Considerations in Airborne Systems and Equipment Certification.*

[4] ISO. (2011). *ISO: Road vehicles – Functional safety – Part 6: Product development at the software level.*

[5] IEEE Computer Society. (2012). *Software & Systems Engineering Standards Committee: 1012–2012 – IEEE Standard for System and Software Verification and Validation.*

[6] Ghosh, A. K., Schmid, M., and Hill, F. (1999). "Wrapping Windows NT software for robustness. In: Fault-Tolerant Computing," in *Twenty-Ninth Annual International Symposium on Digest of Papers* (New York, NY: IEEE), 344–347.

[7] Popov, P., Strigini, L., Riddle, S., and Romanovsky, A. (2001). "Protective Wrapping of OTS components," in *Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, Toronto.

[8] Brancati, F. (2012). Adaptive and Safe Estimation of Different Sources of Uncertainty to Improve Dependability of Highly Dynamic Systems Through Online Monitoring Analysis (New York, NY: IEEE).

[9] Carrozza, G., Cinque, M., Cotroneo, D., and Natella, R. (2008). "Operating system support to detect application hangs," in *International Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS*, Leeds, UK.

[10] Voas, J. M. (1998). Certifying off-the-shelf software components. *Computer* 31, 53–59.

[11] Antunes, N., and Vieira, M. (2009). "Detecting SQL Injection Vulnerabilities in Web Services," in *Fourth Latin-American Symposium on Dependable Computing (LADC '09)*, 17–24. IEEE Computer Society, Joao Pessoa, Brazil.

[12] Laranjeiro, N., Canelas, S., and Vieira, M. (2008). "wsrbench: An On-Line Tool for Robustness Benchmarking," in *IEEE International Conference on Services Computing, 2008* SCC '08 (New York, NY: IEEE), 187–194.

[13] Madeira, H., Costa, J., and Vieira, M. (2003). "The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments," in *Proc. of 2003 International Conference on Dependable Systems and Networks (DSN 2003)* (New York, NY: IEEE), 86–91.

[14] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing.* Hoboken, NJ: John Wiley & Sons.

[15] CentOS Project. *The Community ENTerprise Operating System.* Available at: http://www.centos.org/

[16] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J., and Chen, B. (2005). "Locating system problems using dynamic instrumentation," in *2005 Ottawa Linux Symposium* (New York, NY: IEEE), 49–64 (2005).
[17] Moore, R. J. (2001). "A Universal Dynamic Trace for Linux and Other Operating Systems," in *USENIX Annual Technical Conference, FREENIX Track*, Boston, MA, USA, 297–308.
[18] Red Hat. *JBoss Application Server.* Available at: https://www.jboss.org/jbossas/
[19] eviware: *soapUI.* Available at: http://www.soapui.org/
[20] Ceccarelli, A., Zoppi, T., Bondavalli, A., Duchi, F., and Vella, G. (2014). "A testbed for evaluating anomaly detection monitors through fault injection," in *5th IEEE Workshop on self-organizing real-time systems (SORT 2014)*, Reno, Nevada, USA.
[21] Koopman, P., and DeVale, J. (1999). "Comparing the robustness of POSIX operating systems," in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing Digest of Papers* (New York, NY: IEEE), 30–37.
[22] Vieira, M., Laranjeiro, N., and Madeira, H. (2007). "Benchmarking the Robustness of Web Services," in *13th Pacific Rim International Symposium on Dependable Computing, 2007 PRDC 2007* (New York, NY: IEEE), 322–329.
[23] Golfarelli, M. (2009). "Open source BI platforms: a functional and architectural comparison," in *Data Warehousing and Knowledge Discovery* (Berlin: Springer), 287–297.
[24] Liferay, Inc. *Liferay Portal.* Available at: http://www.liferay.com/