A Testbed for Evaluating Anomaly Detection Monitors Through Fault Injection

Andrea Ceccarelli, Tommaso Zoppi, Andrea Bondavalli
Dept. of Mathematics and Informatics
University of Florence, Florence, Italy
{andrea.ceccarelli, bondavalli}@unifi.it

Fabio Duchi
Resiltech S.r.l
Piazza Nilde Iotti 25, Pisa, Italy
fabio.duchi@resiltech.com

Giuseppe Vella
Engineering Ing. Informatica S.p.A.
Viale Reg. Siciliana 7275, Palermo, Italy
giuseppe.vella@eng.it

***Abstract*—Amongst the features of Service Oriented Architectures (SOAs), their flexibility, dynamicity, and scalability make them particularly attractive for adoption in the ICT infrastructure of organizations. Such features come at the cost of improved difficulty in monitoring the SOA for error detection: i) faults may manifest themselves differently due to services and SOA evolution, and ii) interactions between a service and its monitors may need reconfiguration at each service update. This calls for monitoring solutions that operate at different layers than the application layer (services layer). In this paper we present our ongoing work towards the definition of a monitoring framework for SOAs and services, which relies on anomaly detection performed at the Application Server (AS) and the Operating System (OS) layers to identify events whose manifestation or effect is not adequately described a-priori. Specifically the paper introduces the key concepts of our work and presents the case study built to exercise and set-up our monitor. The case study uses Liferay as application layer and it includes fault injection and data collection instruments to perform extended testing campaigns.**

Keywords - monitoring, SOA, anomaly detection, application server, testing, fault injection

I. INTRODUCTION

In Service Oriented Architectures (SOAs, [5]), a *service* is a self-contained software module that provides defined functionalities and that is independent from the state or context of other services; each service has a published interface and communicates by exchanging messages. The communication between services can involve either simple data passing or two or more services coordinating activities. A SOA is essentially a collection of interacting *services*. SOA is a fundamental support for *business-critical* processes; this is in large part due to the SOA characteristics which comprise, amongst many, software interoperability and reuse, adaption (modification, reconfiguration and flexibility), and dynamicity. In fact, services may be updated, new services may join the SOA, and other services may be removed from the SOAs without the need of notifying to other services [5].

It results evident that a SOA can be a complex software system, with several requirements and even millions of lines of code. The complexity of the code, the services management and their evolution, make SOAs exposed to residual software faults i.e., software faults that escape testing and get activated only during operation. A related issue, although not addressed in this paper, is represented by intentional faults, such as malicious activities, intrusions and cyber attacks [15]. The activation of residual faults, overload conditions and intentional faults can result in failures and services downtime which ultimately lead to huge financial losses and consumer dissatisfaction. It is thus mandatory to monitor a SOA and its services to continuously check their status and timely detect malfunctions [4], [1].

The evolutionary characteristics of SOAs call for monitoring solutions which are as much independent as possible from the services, in order to not require re-configuration each time the SOA is updated, and include the detection of those errors whose manifestation or effect is not adequately described a-priori (i.e., at pre-deployment time).

Trying to address these challenges, our approach is based on shifting the observation perspective from the services to the underlying application server (AS) and operating system (OS), through monitoring OS and AS attributes (parameters) to understand the status and health of the whole system. Thus our objective is to build a monitoring framework for services and SOAs that detects anomalies (by *anomaly* we refer to changes in the variable characterizing the behavior of the system caused by specific and non-random factors e.g., overload, the activation of faults, malicious attacks, etc. [2]) in the OS and the AS. With respect to services, AS and OS are characterized by a reduced flexibility and less subject to evolution, thus requiring only one instrumentation and configuration at the time the system is deployed. Also anomalies reflect the effects of fault in the services, thus such effects can be observed, and anomalies detected, by monitoring OS and AS attributes [6], [20]. The monitor observes a selection of OS and AS attributes, and includes cross-layer detectors to improve detection accuracy and coverage. Amongst the possible error detection approaches, our framework focuses on anomaly detection, as it refers to the problem of finding patterns in data that do not conform to the expected behavior.

Without requiring instrumentation of the services, the monitoring framework will observe only AS and OS attributes; self-configuring detection mechanisms will then be used [6] to adapt to possible changes in the workload. This paper presents an overview of the anomaly detection monitor

and our ongoing activities towards its realization. In particular the paper describes the design and realization of a testbed that allows to perform code mutation experiments and collect data for further analysis.

The rest of this paper is organized as follows. Section II presents the topic of anomaly detection monitoring. Section III discusses our concept for the monitoring and anomaly detection framework. Section IV presents the testbed that we are building to exercise, tune and assess the framework. Section V to Section VIII present details on the testbed, in particular the workload generator (Section V), the monitor (Section VI), the fault injection tool (Section VII) and the data logging and analysis support (Section VIII). Conclusions and future works are in Section IX.

II. MONITORING FOR ANOMALY DETECTION

Several approaches can be identified in the state of the art for anomaly detection in computer systems. To mention a few, in [8] temporal and combinatorial rules, obtained from protocol specifications and system administrators, are used to reveal anomalies in distributed applications. In [9] the authors use mathematical models to define invariants between metrics, which are used to detect the activation of a fault. In [3] the CPU consumption of transactions in web application is modeled by means of statistical linear regression. The model is then used to detect performance anomalies, namely those changes in CPU usage not explained by actual workload.

The work in [6] proposes a configurable detection framework to reveal anomalies in the OS behavior, related to system misbehaviors. The detector is based on online statistical analysis techniques, and it is designed for systems that operate under variable and non-stationary conditions. The framework is evaluated to detect the activation of software faults in a distributed system for Air Traffic Management (ATM). For each observed OS-level attribute, the framework computes lower and upper adaptive thresholds in order to take into account the dynamic behavior of the system. If the value of a specific attribute does not fall within the estimated interval, it is judged to be suspicious. All suspicious attributes are combined to reveal an anomaly.

Anomaly detection techniques need information on the system behavior, obtained by monitoring the system during its operational life. Monitoring technology is widely used in many kinds of software. Recent taxonomy works show that runtime software monitoring has been used for profiling, performance analysis, software optimization as well as software fault detection, diagnosis and recovery [21]. Web services and SOA monitoring is executed in parallel with the normal executing processes without interrupting them. Several research works on web service monitoring exist with different motivations and frameworks e.g., [22], [24], [1].

The approach we are planning for anomaly detection is built on [6], but we are also considering the Application Server layer. While monitoring solutions exit in the state of the art to collect the quantities of interests, given the potentially vast number of attributes observable, an accurate selection of the attributes to monitor is needed.

III. THE ANOMALY DETECTION FRAMEWORK

The ultimate objective of our work is to design a monitoring framework for anomaly detection whose requirements are: i) integrate with OTS (Off-The-Shelf) and legacy subsystems; ii) operate under non-stationary, unpredictable and variable conditions; iii) reveal anomalies related to unintentional or malicious faults, and to performance or overload issues; iv) be independent from the service layer to not require updates for each service evolution, and enhance portability.

The monitoring and detection framework consists of two separate components: an observation component, and a detection component. We present the framework with the help of Fig. 1.

The services layered on the AS interact with other SOA services (*external services* in Fig. 1) and users to perform their tasks; these are reported in grey in Fig. 1. The AS and the OS are the target of the monitoring activity: the probes of the observation component monitors OS and AS attributes. OS probes are inserted in the OS kernel to monitor attributes of operating system. Example of OS and AS attributes that we consider are reported in Section VI.A and Section VI.B.

The data obtained from the OS probes and AS probes are sent to the detection component which is responsible of the anomaly detection task: it merges and weights the data received from AS and OS probes. We propose to use adaptive thresholds to detect software-related anomalies and evolve thresholds through time, following the approach in [6] where different OS attributes are monitored and the data achieved from them is used to i) detect anomalies and ii) adapt detection thresholds through time. Our objective is to extend the monitor to include AS attributes, and define weighting functions that include results both at the AS and OS levels.

The output of the detection framework can be ultimately used for diagnosis and remediation actions.

IV. TESTBED PRESENTATION: HIGH LEVEL VIEW

We present the testbed and the related case study that we developed to test our monitoring solution. The general idea is to monitor attributes while exercising the system with different workload and faultload.

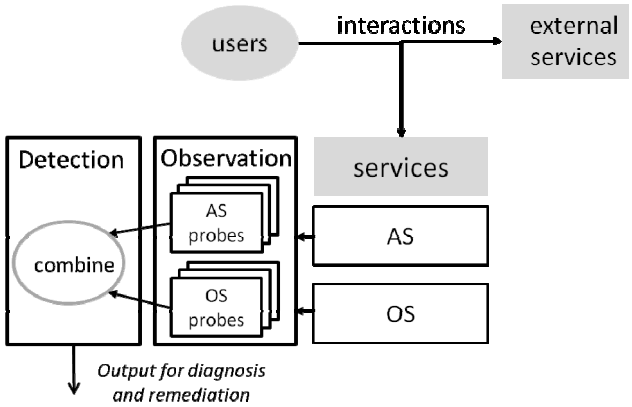

Figure 1. Overview of the anomaly detection framework.

For this scope, our testbed uses as target system the Liferay Portal 6.1 Community Edition [10] SOA running on an instance of Tomcat 7.0.42 AS and CentOS 6 with kernel 2.6.32. The rest of this section presents the testbed, with the help of Fig. 2. The *system under test* is deployed on a monitored machine, while data collection, inputs and outputs of the application layer, and the values collected by the monitoring probes are collected and stored in a database on a separate machine.

A. Liferay Portal 6.1

Liferay is a free and open source Java software that was initially developed to provide a open source enterprise quality portal. Since the early stages of development, Liferay has been widely adopted for intranet as well as extranet enterprise solutions. Our installation of Liferay comprises the version 6.0 of the portal and includes 83 deployed SOAP (Simple Object Access Protocol) web services.

B. Workload Generator

To generate and execute the workload we developed a Java application that accesses different services. This application simulates the behavior of different users of Liferay, invoking the services to perform common operations such as logging, adding a user group, updating organization information or blog, etc. Different combinations of users can simultaneously execute.

C. Fault Injection Tool

Fault injection [16] is performed by a tool we developed to insert software faults in the Liferay source code. Together with the workload execution, this will allow to exercise the attributes of the AS and OS both in presence/absence of faults.

D. The Monitor

The probes that monitor attributes of the Tomcat AS are realized using the JMX (Java Management eXtension, [11]) technology. JMX defines an architecture, the design patterns, the APIs, and the services for application and network management and monitoring in Java.

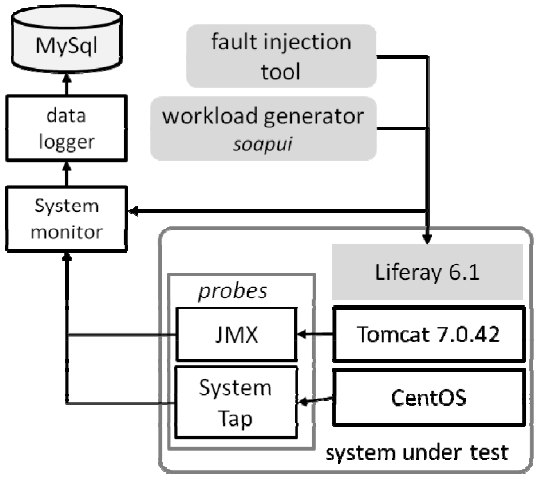

Figure 2. Overview of the testbed and main technologies applied.

For the purpose of our work, we use the Tomcat Mbean server, that gives information about metrics related to Tomcat and JVM.

Probes that monitor the OS attributes are located in kernel modules which in turn are produced using System Tap. System Tap provides a software infrastructure that simplifies the collection of information about the running Linux system. Among its features, it eliminates the need for the developer to go through the sequence of instrumenting, recompiling, and installing that may be otherwise required to collect data.

System Tap exploits the modularity of the Linux kernels design to produce a kernel modules that once loaded has visibility on kernel structures. The System Tap compiler produces a kernel module which accesses directly the kernel internal data.

E. Data Logger and Analysis

Data collected by the AS and OS probes are stored in a database. The probes are organized in such a way that an analysis with the OLAP (On-Line Analytical Processing, [12]) methodology can be conducted.

The underlying database can interface to analysis tools in order to speed up the offline data retrieval and analysis.

V. Workload Generation and Execution

A. Description of the Workload Generator

The objective of automatic tests execution implies the need of a tool capable of generating a workload on the target system. Such a tool has been called Workload Generator. Key requirement of the Workload Generator is flexibility, both in terms of ability to invoke any SOAP method and in terms of sequence and complexity of invocations. The tool must be able to manage complex interactions, thus being able to handle the results of each invocation and, when it is needed, to re-use it appropriately for the subsequent invocations. Finally, the Workload Generator must be able to provide information on the execution data to the data logger and analysis tools, such as execution duration, exception reporting and environment data; examples of the latter data type are *AS name*, *tested services URI*.

Our tool is build on SoapUI [23], a free and open source cross-platform functional testing solution. SoapUI is designed to interact with SOAP services amongst others. It has a rich set of features, ranging from functional test to scripting support. SoapUI covers most of the requirements for our tool, thus the library upon which SoapUI is built has been used as base for the development of our Workload Generator. On the other hand, SoapUI was not fulfilling our purposes because although Groovy scripting is supported, libraries for interacting and exchanging data with the necessary complexity are not provided.

Each method that can be invoked by the Workload Generator must have a description of its attributes and must be classified into one of the following categories: i) attributes with fixed values (*fixed attributes*), whose values are defined at the beginning of the execution, and ii) attributes that can be defined or changed during the execution (*variable attributes*).

For each method that is part of the workload, it is required to understand the meaning of its attributes. Then the values for the fixed attributes are decided on the basis of the workload purpose. For variable attributes, the values are set during the execution of the workload.

The set of methods which are exposed by a SOAP service are described in a XML file; each method is defined by its invocation string, the value of fixed attributes, and the name of variable attributes. These methods constitute the building blocks for the workload.

A workload is made up of a sequence of invocation of methods and flow controlling structures. In order to simulate any kind of human interactions, the workload execution engine supports parametric method invocation; based on a method response, as a human would do, the workload changes its behavior and the subsequent methods invocations.

Workload execution is made up of a sequence of statements that are executed sequentially. Three types of statements are supported:

- *If*,
- *While*,
- *Call*.

If and *While* are compound statements, thus they can contain a subsequence of statements made up of statements of the three types. The *if* statements can change the execution flow, depending on the dynamic evolution, executing an optional block. *If* statements can be followed by *else*, acting likewise any other programming language.

While statements execute the same subsequence of statements until a counter, which can be initialized by a variable or by a constant, reaches 0.

Call statements invoke a method of a SOAP-service; the method to be called as well as its attributes are defined in the statements. Optionally, the *Call* statements can assign results from the response content to a variable.

Invocation attributes of *Call* statements can be assigned to a fixed or variable attribute, in the latter case the variable must be defined before invoking the method. The value is read just before invoking the method. Thus in a *While* statements every *Call* execution can have different values as its attributes.

The nature of the tests led to a Workload Generator which collects any correct answer or exception sent back by the Application Server or the tested services. Each exception is then reported to the logging tool as well as the configuration, the current context and the step which raised the exception.

Fig. 3 presents an extract of the workload used for testing the Thread Messages Service. The tag *Tns:Call* defines a call to the method *getUserById/1* of service *Portal_UserService*. A unique name for each method is set by suffixing the original method name with the index of the corresponding requested method in the WSDL (Web Services Description Language, it offers a description of the interface of a web service). The tag *Attribute* sets the method attribute named *userId* with the content of *currentUserId* (variable attribute). Instead *StoreResult* declares that the response field *companyId* must

```
<tns:Call portlet="Portal_UserService" method="getUserById/1">
  <tns:Attributes>
    <tns:Attribute name="userId" variable="userId"/>
  </tns:Attributes>
  <tns:Results>
    <tns:StoreResult variable="companyId" name="companyId" />
  </tns:Results>
 </tns:Call>
…
<tns:If variable="userId" condition=”!=” value="NaN">
  <tns:IsTrue>
   …
  </tns:IsTrue>
 </tns:If>
…
<tns:While variable=”number_of_categories”>
  …
</tns:While>
```

Figure 3. Example of XML file describing the workload.

be set to the value indicated by *currentCompanyId* (variable attributes).

The command *Tns:If* defines an *if* statement in which the *companyId* is checked against *NaN*. This command defines a condition that can be set to check if the value is lower than, less than, equal, not equal to a value and, in case of a string, if it contains, starts with or ends with a specific substring. The value, or the substring, can be a constant or can be another variable.

Finally, *tns:While* defines a *while* statement in which the code is executed as many times as the value indicated by *number_of_categories*.

B. Usage of the Workload Generator

Workload Generator is started by command line. It can specify a workload file: in this case, when the workload ends, the Workload Generator stops its execution. Otherwise the Workload Generator is started with the *daemon* option; in this case the Workload Generator opens a communication channel by which an external tool can control and activate the execution of a specific workload. More than one workload can execute concurrently. The Workload Generator terminates when the execution of the workload ends or, in *daemon* mode, when it receives the *quit* message. This command forces the interruption of the execution independently of potential hangs of services.

The information generated by the Workload Generator and logged are, beside of workload and invoked methods, the execution time and the received exceptions. This information together with values read from the AS and OS probes give a description of the system during the execution.

C. Workload Description

A set of workloads were designed specifically for Liferay. To build them, it was necessary to understand how services integrate with each other and which are the sequences of services invocations. This activity required a good knowledge of the tested system, thus it has been carried out following Liferay documentation and its services source code.

As a result, several different workloads have been defined taking into account *high level activities* such as user creation, site creation, posting into a blog or forum, deleting a message and so forth. For each high level activity, the methods invoked by web services have been identified; finally, each attribute of these methods was classified in fixed attribute or variable attributes by taking into account whether their value is defined at the beginning or during the execution.

If the high level activities require invocations to other methods to acquire the values of attributes, the workload includes all the steps that are implicitly or explicitly needed to gather such values.

For example, considering the high level activity "remove a calendar item", the workload is composed of the actions:

- request all the calendar available for the user;
- pick the one which the user is interested in;
- list all the calendar events;
- remove an item.

Using the various high level activities and related actions, it is possible to rapidly compose a single workload, starting from the small sequence of invocations. Overall, we identified 15 high level activities. The identified high level activities constitute around 80% of the Liferay high level activities.

D. Workload Execution

The workload is executed starting the Workload Generator with the XML file which defines the workload.

Before the execution of the workload, data which describe the monitored system are collected. An example of such kind of data is the amount of available RAM or the number of CPUs; furthermore other kind of data can be collected, such as number of current active threads, garbage collector policy, etc.

The workload execution, at the very beginning, sets up the structures where invocation attributes (arguments of the *Call* described below) will be written to and read from.

VI. THE MONITOR

The monitor reads data from OS and AS probes that have been installed on the target system. Data which are collected and sent to the data logging and analysis tools are relying on two different components running on the target system:

- The AS Monitor, in charge of AS Tomcat monitoring;
- OS Probes Server, in charge of OS CentOS monitoring;
- A System Monitor, which is external to the monitored system and coordinates the two above.

A. AS Monitor

The AS Monitor has been specifically designed for interacting with the Management Bean Server. The AS Monitor provides access to the Management Bean Server integrated in Tomcat.

The Management Bean Server collects a wide range of attributes and characteristics of the application server, from "cache usage" to "thread pool usage". At the beginning of this experience, we tried to collect all data, but the drawback of this approach is a massive load on the AS and thus on the monitored system. This invalidates OS probes data and possibly even the Management Beans data itself. In fact for the considered AS (Tomcat), the total number of potential attributes is 8000. Thus a selection of them has been performed, reducing to 50 attributes, on the basis of preliminary runs, during which all the data were collected, and on the basis of experience in selecting the most relevant attributes that describe the behavior of the Application Server. In particular, the choice takes into consideration the main areas where we expect anomalies can manifest:

- Attributes related to *Threads* and *Memory* status, usage and evolution;
- Attributes describing the *Request*, including information on the processing duration and response time;
- Attributes describing the activities of the *Garbage collector*.

For example, *Thread* probes provide information about the number of active threads as well as the number of requests that are waiting to be served. A rapidly and unexpected decreasing number of active threads, or an unexpected and increasing number of waiting requests are examples of possible anomalies.

Anomaly detection is based on the observation of how the system changes over the time. The attributes that can be read from the chosen areas provide a snapshot of the system in a specific moment. Furthermore the Management Bean does not notify AS Monitor about specific events that are affecting the monitored system. These consideration have led to design an AS Monitor that periodically requests status updates of the system. The evolution of these data over time allows to monitor for anomalies.

B. OS Probes Server

The OS Probes server has been designed to pour the data that comes from OS probes toward the outside of the system following the client-server paradigm. A pipe file is used to let OS probes communicate with the OS Probes server.

For the purpose of this work the OS monitoring activity collects information strictly related to the process where AS is running as well as the general state of the system.

Data collected which describe the OS are gained listening for syscalls that are targeted to the OS process. Every time a monitored syscall is called, OS probes notify the event to the OS Probes Server.

The system data collected describe the overall system evolution. Examples of such data are free memory, disk throughput, network throughput. Probes listen for syscall, and data from the syscalls are cached. Memory allocations, disk writing, etc., happen so often that flushing directly all these events would seriously impact to the monitored OS. For these reasons, these values are cached and flushed periodically toward the OS Probes server.

To keep a low impact on monitored system, OS probes write the data that are read from syscalls, without rearranging

or any kind of manipulation for sorting them; thus OS Monitor has to parse, organize and arrange these data.

A detailed description of the OS attributes that we are monitoring is in [6], [17], together with motivations for the selection of the attributes amongst all those available.

C. System Monitor

The System Monitor acquires data from the OS Monitor and AS Monitor at fixed time intervals.

D. Intrusiveness of Probes

Reading AS probes as well as OS probes, their activity can affect the performance of the monitored system. The effort that the monitored system needs for providing the probe values depends on the effort for reading the single value and on the frequency of the readings. The first issue has been addressed by identifying a subset of AS and OS probes, and by an efficient implementation of OS probes. Intrusiveness of the OS probes has been measured, resulting an impact as low as 3,5% [6] when data are collected with a periodicity of 1 time per second.

Preliminary runs have been conducted to evaluate the intrusiveness of AS probes with request frequency of 1 time a second. The runs were aimed to evaluate if the system performance are affected by the monitoring system (we remember that overall we are collecting data from 20 OS probes and 50 AS probes). Results showed that neither CPU or memory usage are significantly altered.

VII. Fault Injection Tool

In this paragraph we describe the fault injection tool that we build in order to exercise the system in the presence of software faults in the application level and thus studying the impact of software errors on the monitored variables. The fault injection tool allows to run experiments and collect data, associating the faults effect to the system behaviour. Tests are made by software implemented fault injection (SWIFI, [19]) strategy. Following the model in [19], we define a fault library, a fault injector, and a controller. Monitor, workload generator and data logger are described instead in the appropriate sections of this paper.

A. Fault Library

Regarding the fault model library, faults we considered for the injections are from [7], where the most common software faults are enlisted. These mainly refer to function calls, conditional blocks, conditional expressions, variable assignments and initializations. Each of this categories contains specific faults, e.g. wrong/missing variable assignment, and each of them helps us to test a specific behaviour of the corrupted system under test. At the time this paper is written, the injection tool is still partially under development and the current subset of implemented faults is composed of: i) missing function call, ii) missing OR/AND condition using an expression, iii) missing if construct before statements, iv) missing if construct plus statements, v) missing if and else construct before statements [7].

The architecture of the tool is scalable to the addition of new kind of software faults, even outside the classification proposed before. The only requirement for the addition of new faults is being able to create parsing rules that identify the injection points in the source code and define the mutated code. To change programming language, even if the tool structure remains the same, it would be necessary to rewrite the code parsing rules to adapt to the new language, and update the fault mutation functions (see Section VII.B) of the code.

At the end of the instrumentation of the faults in the code, for each injection, the following information is reported in a configuration file:

- source file, line, method, and class where each injection is performed, and kind of faults injected;
- number of faults injected of a specific kind.

This configuration file is used to setup the experiments and guide the activation of faults performed by the injector described below.

B. Injector and Controller

We now describe how faults are injected in the code and activated. The injector creates the modified source code that is then compiled. The injector offers the choice of executing the correct or the faulty piece of code, for each inserted fault, according to the specifics of the experiment.

If an experiment requires the activation of a specific fault, the execution flow is the correct flow except for the part of code correlated with the activated faulty code. To ensure this property, we use an activation mechanism based on the system's environment variables. If we want to activate a fault with id=*x*, we define an environment variable with the same name and instruct the controller to cope with information on the environment variable. The execution flow changes with the execution of the code that injects the fault *x*. We can see a pseudocode of the code mutation below for a generic fault with id=*x*:

```
// fault id = x
if(system.getEnv(x) != null)
        signal corrupted execution
        run faulty code
else
        run normal code
```

where system.getEnv(*x*) in this case is the value of the system variable *x*. It is initializated only if the experiment requires the activation of the fault *x*.

This activation process is simple: it requires only small changes in the code with minimal performance impact: the system invocation to the environment variable is very fast (does not require disk access); also each experiment must define only the environment variables related to the faults that must be activated without burdening on their management by the OS.

C. Tool Usage

As described, we want to make a SWIFI fault injection tool that performs Java code mutation that can work on any general purpose Java application. We perform compile time injection, where the code is instrumented with a large number of faults. Those to activate are selected dynamically using system variables. This allows us to make a huge number of experiments without wasting time to recompile our code. To do this we must choose a sufficiently large set of faults to inject, according to a specific fault type (e.g., missing function call) or to a defined set of code pieces (e.g., we can decide to instrument only certain classes).

To instrument the code all at once with all faults isn't a good approach: we have verified that the original Liferay's code dimension is 89,4 MB (8670 Java classes, at average 10,3 KB per class). The dimension of the instrumented code instead is 201,1 MB (at average 23,2 KB per class). When the implementation of the faults library will be complete, we expect even greater growth, that can increase the load on the AS and alter its performances. To avoid this, assuming to have a defined workload, we use a code tracing tool to list all the methods (and the classes) called by the specific workload and we inject only faults in the invoked methods. In other words, first the workload is executed in a golden run with the Liferay system instrumented with a code tracer, to understand the methods that are invoked. Then only these methods (and classes) will be involved in the instrumentation process, restricting the excessive growth of the code. Obviously the code tracer is used only during such golden run and it is disabled afterwards to reduce intrusiveness on the target system [14], [13].

D. Experiments Execution

After describing the injected faults and their activation methods, we provide a brief overview on the methodology applied to execute the experiments. First of all, as said before, we have to choose where faults should be injected in our code (in our case, we define a workload and trace the methods it invokes) and compile the source code; then we run all the tests we need, simply defining each time one or more environment variables (each of them corresponds to the faults we want to activate during the execution of the workload). Note that tests can be executed without the intervention of the user: the controller cyclically sets the specific group of environment variables, starts the application, launches the Workload Generator, and manages the data collection from the AS/OS monitors until a timer signals the end of the work or until the workload terminates.

VIII. Data Logger and Analysis

OLAP (Online Analytical Processing) methodologies enable users to analyze multidimensional data, interactively and from multiple perspectives. In our framework, we use the OLAP techniques for data logging, structuring of results and analysis of results. A data logger aggregates data from the System Monitor and Workload Generator. System Monitor gives the information on the system evolution in an interval of time. The Workload Generator can provide data from user perspective, and it also gives additional details for the data aggregation. In fact the Workload Generator can provide the time span in which the workload has been executed, and environment data can be aggregated.

For each run of the experiments, the data logger retrieves information about the environment of the monitored system and prepares to host collected data. System Monitor starts collecting data from the monitored system and each sample is passed to the data logger. The data logger records all these information until the workload ends. Workload can end with a success or a failure. When the workload terminates, the data logger finally flushes its data into a MySql database.

The data logger organizes data in the database conforming to a defined star schema [12], that is an intuitive model composed of facts and dimensions tables: *facts* tables contain the measurement results that we are collecting and the quantities that we are measuring, and *dimensions* tables contain the key features of the analysis. This model allows to structure and highlight the objectives, the results and the key elements of our evaluation, thus it helps to define in an (as much as possible) unambiguous way the purposes and contexts of the analysis [18].

Given the number of monitored attributes which should be placed in the fact table, for simplicity we organize the star schema with several facts tables. Facts tables must refer to the dimensions tables that describe the context in which the facts have been collected.

When more “fact” tables are involved, each “fact” table aggregates strongly related facts, improving usability, although a some effort for the optimization of complex queries may be necessary.

For this work several fact tables has been defined, for examples: *fact_Cache*, which collects data on the system caches, *fact_Memory*, which collects data on the system RAM, *fact_Thread* which collects data on the monitored threads, *fact_GarbageCollector* which collects data on the Java garbage collector.

Each fact table collects all the data which refers to a specific topic, and is limited to a subset of dimension tables which are strictly related to the collected metrics.

For example, the fact called *fact_AS_Memory* contains information about the amount of heap memory currently allocated, the amount of memory waiting for garbage collecting and a reference to the dimension *dim_AS_Memory* which contains the total amount of heap memory.

The data logger stores the data collected from System Monitor and Workload Generator into the corresponding fact tables of the star schema defined for this work. A parser was build for this scope. The environment variable as well as other information provided by Workload Generator are stored in dimension tables.

To identify the anomalies in the data, it is first required to collect golden runs, where faults are not injected, and then collect runs where faults are injected. We plan to compare the values of the attributes contained in the facts tables, to identify significant differences from the golden runs and the runs in

which faults are activated. To assure that the differences in the executions are due to anomalies, an investigation on the activated faults and its effects on the source code may be needed.

IX. Conclusions and Future Works

This paper presents our ongoing work towards the development and assessment of a monitoring solution for Service Oriented Architecture. Our approach for monitoring aims to use anomaly-based detection techniques at the Application Server and Operating System layers, in order to perform error detection without requiring detailed knowledge of the services executing on the SOA.

In this paper we present the case study and the related testbed that will be used. As future work, tests are planned to exercise the target system using the tools and techniques described in this paper. Results will provide the inputs required to understand the behaviour of the different attributes and tailor a monitoring solutions for the different variables.

Acknowledgments

This work has been partially supported by the European Projects FP7-2012-324334-CECRIS (CErtification of CRItical Systems) and ARTEMIS-JU n.333053 CONCERTO (Guaranteed Component Assembly with Round Trip Analysis for Energy Efficient High-integrity Multi-core Systems), the Regional Project POR-CREO 2007-2013 Secure! funded by the Tuscany Region, the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research, and the PON Ricerca e Competitività 2007 - 2013 VINCENTE (A Virtual collective INtelligenCe ENvironment to develop sustainable Technology Entrepreneurship ecosystems) project.

References

[1] A. Ceccarelli, M. Vieira, and A. Bondavalli, "A Service Discovery Approach for Testing Dynamic SOAs," In Proc. of the Int. Symp on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '11). IEEE Computer Society, Washington, DC, USA, pp. 133-142, 2011.

[2] D.C. Montgomery. Controllo statistico della qualità. McGraw-Hill italia, First Edition, 2000.

[3] L. Cherkasova, K. Ozonat, N. Mi., J. Symons, and E. Smirni, "Anomaly? Application change? Or Workload change? Towards automated detection of application performance anomaly and change," IEEE Int. Conf. on Dependable Systems and Networks (DSN), pp. 452-461, 2008.

[4] A. Ceccarelli, M. Vieira, and A. Bondavalli, "A testing service for lifelong validation of dynamic SOA," IEEE Int. Symp. on. IEEE High-Assurance Systems Engineering (HASE), pp. 1-8, 2011.

[5] M.P. Papazoglou, and W. Heuvel, "Service Oriented Architectures: approaches, technologies and research issues," The VLDB Journal 16(3), pp. 389-415, 2007,

[6] A. Bovenzi, S. Russo, F. Brancati, A. Bondavalli, "Towards identifying OS-level anomalies to detect application software failures," IEEE Int. Workshop on Measurements and Networking Proceedings (M&N), pp. 71-76, 2011.

[7] J. Durães, and H. Madeira, "Emulation of software faults: a field data study and a practical approach," IEEE Trans. on Software Engineering, 32(11), pp. 849-867, 2006.

[8] G. Khanna, P. Varadharajan, and S. Bagchi, "Automated online monitoring of distributed applications through external monitors," IEEE Trans. on Dependable and Secure Computing, , vol.3, no.2 pp. 115-129, 2006.

[9] G. Jiang, H. Chen, and K. Yoshihira, "Modelling and tracking of transaction flow dynamics for fault detection in complex systems," IEEE Trans. on Dependable and Secure Computing, vol.3, pp. 312-326, 2006.

[10] Liferay, http://www.liferay.com [Accessed 30 January].

[11] JMX Java Management eXtension, http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html [Accessed 30 January].

[12] R. Kimball, M. Ross, and W. Thornthwaite. The Data Warehouse Lifecycle Toolkit. J. Wiley & Sons, Inc, 2008.

[13] A. Bondavalli, A. Ceccarelli, L. Falai, M. Vadursi, "A New Approach and a Related Tool for Dependability Measurements on Distributed Systems," IEEE Transactions on Instrumentation and Measurement, vol.59, no.4, pp.820,831, April 2010.

[14] A. Bondavalli, A. Ceccarelli, L. Falai, M. Vadursi, "Foundations of Measurement Theory Applied to the Evaluation of Dependability Attributes," 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07), pp.522,533, 25-28 June 2007.

[15] L. Coppolino, S. D'Antonio, L. Romano, G. Spagnuolo, "An Intrusion Detection System for Critical Information Infrastructures using Wireless Sensor Network technologies," 5th International Conference on Critical Infrastructure (CRIS), pp.1,8, 20-22 Sept. 2010.

[16] M. Cinque, D. Cotroneo, C. Di Martino, A. Testa, S. Russo, "AVR-INJECT: a Tool for Injecting Faults in Wireless Sensor Networks," In Proc. of the 23rd IEEE International Parallel & Distrubuted Processing Symposium (IPDPS '09), pp. 1-6, 2009.

[17] A. Bondavalli, F. Brancati, A. Ceccarelli, D. Santoro, M. Vadursi, "Experimental analysis of the first order time difference of indicators used in the monitoring of complex systems," in Proc. of IEEE Intern. Workshop on Measurements & Networking, M&N2013, Naples, Italy, pp. 138-142, 7-8 October 2013.

[18] H. Madeira, J. P. Costa, and M. Vieira, "The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments," in Proc. of the 2003 International Conference on Dependable Systems and Networks, pp. 86–91, 2003.

[19] Mei-Chen Hsueh, T.K. Tsai, R.K. Iyer, "Fault injection techniques and tools," Computer , vol.30, no.4, pp.75-82, Apr 1997.

[20] Q. Guan, S. Fu, "Adaptive Anomaly Identification by Exploring Metric Subspace in Cloud Computing Infrastructures," IEEE Symposium on Reliable Distributed Systems, pp. 205-214, 2013.

[21] N. Delgado, A. Q. Gates and S. Roach, "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools," IEEE Transactions on Software Engineering, Vol. 30, No. 12, December 2004.

[22] Robinson W.N., "Monitoring Web Service Requirements", Proc. of the 12th Int. Conf. on Requirements Engineering, 2003

[23] SoapUI, http://www.soapui.org/ [Accessed 30 January].

[24] Z. Li, Y. Jin and J. Han, "A runtime Monitoring and Validation Framework for web service interactions", Australian Software Engineering Conference, 2006.