MADneSs: A Multi-Layer Anomaly Detection Framework for Complex Dynamic Systems

Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli, *Member, IEEE*

Abstract—Anomaly detection can infer the presence of errors without observing the target services, but detecting variations in the observable parts of the system on which the services reside. This is a promising technique in complex software-intensive systems, because either instrumenting the services' internals is exceedingly time-consuming, or encapsulation makes them not accessible. Unfortunately, in such systems anomaly detection is often ineffective due to their dynamicity, which implies changes in the services or their expected workload. Here we present our approach to enhance the efficacy of anomaly detection in complex, dynamic software-intensive systems. After discussing the related challenges, we present MADneSs, an anomaly detection framework tailored for the above systems that includes an adaptive multi-layer monitoring module. Monitored data are then processed by the anomaly detector, which adapts its parameters depending on the current system behavior. An anomaly alert is provided if the analysis conducted by the anomaly detector identify unexpected trends in the data. MADneSs is evaluated through an experimental campaign on two service-oriented architectures; software faults are injected in the application layer, and detected through monitoring of underlying system layers. Lastly, we quantitatively and qualitatively discuss our results with respect to state-of-the-art solutions, highlighting the key contributions of MADneSs.

Index Terms—Anomaly detection, software-intensive system, dynamicity, MADneSs, SOA, multi-layer, context-awareness

1 Introduction

Complex systems are intrinsically difficult to model due to dependencies, relationships, or interactions between their parts or the environment [49]. As an example, control systems for power grids management have to supply energy, whilst at the same time maintaining operational performance and reducing costs also when the demand is changing frequently and unpredictably.

To deal with the above dynamicity and complexity, there is a need for better interoperability and integration of control functions on different hierarchical levels, such as process control and operations management services. These systems are usually modeled and realized as modular reconfigurable systems based on reusable distributed components integrated within Service Oriented Architectures (SOAs) or Systems of Systems. Implementation details are often not accessible, as they are proprietary or legacy software. Additionally, services are often characterized by a dynamic behavior, where the services themselves and their interactions with others may be updated. Further, services can evolve through time, due to changes in their requirements and in their behavior [22].

As a result, instrumenting each individual service to monitor dependability-related properties in software-intensive systems is generally difficult if not unfeasible [33]. Several works in the last decade [13], [16], [34], [35] report on the difficulties of detecting service errors in complex systems before they escalate into system failures. These difficulties are mainly related to the multitude of relations, interconnections and interdependencies that propagate wrong decisions. For example, an update, a configuration change, or a malfunction in a single module or service can affect the whole system.

To tackle this problem, several works are focusing on *anomaly detection*, which refers to the problem of finding patterns in data that do not conform to the expected, or normal, behavior [1]. In most of the cases, specific and non-random factors are the causes of the pattern changes above. For example, the activation of software faults or malicious activities may generate a system overload.

However, the characterization of the expected behavior, and consequently the identification of the anomalies, is challenging due to dynamicity and evolution characteristics of complex systems. *Currently, there are no clear state-of-the-art answers on applying error or anomaly detection in highly dynamic and complex systems*. Problems are mainly related to the necessity of frequently reconfiguring the detection algorithms to match changes of the system. These issues call for monitoring solutions that: i) require minimal knowledge on the services; ii) avoid direct instrumentation of the services with monitoring probes, iii) automatically reconfigure the monitoring system.

Our Contribution. In this paper we present our approach to anomaly detection in complex dynamic systems. Our main contributions reside in i) leveraging multi-layer monitoring and ii) exploiting context-based detection. In addition, we also provide a framework that—to the authors' knowledge—is the first that allows applying the techniques above in an orchestrated and structured way. More in detail, we list

• *The authors are with the Department of Mathematics and Informatics at University of FlorenceViale Morgagni 67/A, Florence, Italy. E-mail: {tommaso.zoppi, andrea.ceccarelli, andrea.bondavalli}@unifi.it.*

Manuscript received 22 Jan. 2018; revised 7 Feb. 2019; accepted 26 Mar. 2019. Date of publication 30 May 2019; date of current version 12 Mar. 2021. (Corresponding author: Tommaso Zoppi.) Digital Object Identifier no. 10.1109/TDSC.2019.2908366

several research challenges related to dynamic systems that negatively affect the efficacy of traditional anomaly detection techniques. Then, as a methodological contribution, we propose general design choices to mitigate such challenges. Afterwards, we present *MADneSs*, a novel *Multi-layer Anomaly DetectioN framEwork for complex Dynamic SystemS* that tackles the challenges above. The monitoring approach we adopt in MADneSs consists in shifting the observation perspective from the application layer, where services operate, to the underlying layers, namely operating system (OS), middleware, and network. This allows detecting anomalies due to errors or failures that manifest in services that are not directly observed. This *multi-layer* approach is suitable to cope with system dynamicity. In fact, when services change, the expected behavior may change and, consequently, a new configuration of the anomaly detector is needed. The layers underlying the services' layer are not modified, and consequently the monitoring system is unaltered. Further, as already examined in [45], [46], [47] and according to our previous work [5], we show that a more accurate definition of the context i.e., *context-awareness*, improves the detection accuracy. In fact, we substantiate this hypothesis as follows. MADneSs allows monitoring a wide set of indicators, where the most relevant for anomaly detection purposes are identified depending on the current context, which is reconstructed by dedicated mechanisms. Context-awareness is also used to train the parameters of the implemented anomaly detection algorithm, tailoring them on the current context and ultimately maximizing their ability in detecting anomalies. The anomaly detection algorithm we selected for MADneSs is SPS (*Statistical Predictor and Safety Margin*, [19]), which predicts an acceptability interval for the next observed value based on a sliding window of past observations. SPS is more suitable for dynamic systems than other algorithms as *clustering* or *neural networks* [1] since it requires short periods of training and quickly re-computes the values of its parameters.

The experimental assessment is conducted by exercising MADneSs on: i) a part of the prototype of the Secure! [11] *Crisis Management System* (CMS), and ii) *jSeduite*. Both systems are structured as a SOA, where services are managed by different entities and are deployed on different nodes. Such services may incur in frequent updates, or even new services may be introduced, together with modification to their orchestration. Consequently, while instrumenting each service with probes is unfeasible, the opportunity to observe the underlying layers, i.e., *middleware*, *OS* and *network*, is offered in both systems.

Paper Structure. The paper is organized as follows. Section 2 presents basics on anomaly detection and its application on complex systems. Section 3 discusses the design choices and our approach. The resulting MADneSs framework is described in Section 4 along with the devised methodology. An extensive experimental campaign targeting the *Secure!* and *jSeduite* SOAs is presented in Section 5. Discussions and comparisons are expanded in Section 6, letting Section 7 to conclude the paper.

2 Basics, State of the Art and Challenges

2.1 Basics on Anomalies and Anomaly Detection

Anomalies are classified in [1] as i) *point anomaly (outlier)*: a single data instance that is out of scope or not compliant with the usual trend of a variable, ii) *contextual anomaly:* a data instance that is unexpected in a specific context, and iii) *collective anomaly:* a set of related data instances that is anomalous with respect to the dataset.

Anomaly detectors have been proposed to detect errors and intrusions [16] or to predict failures [2], based on the hypothesis that the activation of a fault or the manifestation of an error generates increasingly unstable—and anomalous - performance-related behavior before escalating into a failure. An anomaly detector analyzes such behavior, detecting anomalies used to trigger adequate diagnostic routines or recovery strategies.

Point anomalies can be detected using algorithms that identify outliers [24] in a trend, as pattern recognition [23] or statistical-based methods which are able to reconstruct the statistical inertia of the trend under investigation [19], [21]. *Contextual anomalies* are detected by techniques that are able to tune their behaviour depending on the current state of the system. They define the expected behaviour in the current context of the system; then they use historical [24], user/operator-provided [25], or runtime [47] data to compare the expectations with the data provided by the monitoring modules. Summarizing, contextual anomalies identify data points that are not expected in a given context. *Collective anomalies* are usually harder to detect [1], and require more sophisticated detection techniques, either looking for specific patterns [25] or using wider training sets to better define them. Despite different techniques may be effective for detecting point and also collective anomalies without having any information on the context, to identify contextual anomalies the chosen technique should include strategies that allow tracing the current state of the system.

2.2 Anomaly Detection in Complex Systems

MADneSs was primarily designed to target complex *dynamic* systems. In [22], dynamicity is described as *the capability of a system to react promptly to changes in the environment*. A system that is not intended to change during its life is called static or semi-static. Some anomaly detection algorithms that dynamically adapt their behavior to suit the current state of the system were already proposed in the literature [7], [21], but in general they require heavy manual intervention when services change. Noteworthy, no clear answers to characterize the expected behavior and to define monitoring and data analysis strategies in complex systems were provided in the state of the art.

Instead, several studies describe frameworks [2], [3], [4], [6], [7], [9], [19] tackling anomaly detection in complex systems that rarely change i.e., *semi-static* systems. In general, these works address either error detection or failure prediction, gathering data about indicators related to multiple system layers. In particular, in *CASPER* [2] the authors use different detection strategies based on symptoms aggregated through *Complex Event Processing* (CEP) techniques using data gathered by observation of network traffic parameters. *Tiresias* [3] predicts crash failures through the observation of network, OS and application layers by applying an anomaly detection strategy instantiated on each parameter. Differently, SEAD [7] aims at detecting configuration or performance anomalies in cluster and cloud environments by observing data coming from the middleware or the cloud hypervisor. In [6], the authors describe a process for invariant

TABLE 1
General Design Choices, Challenges Involved and Our Approach to Them

Scope	Design Choice	Challenges	Our Approach	Alternative Approaches
Monitoring	Monitoring Strategy	CH3	Static Multi-Layer Monitoring	Dynamic Monitor [42], Database [15], Non-Intrusive [2]
Monitoring	Context-Awareness	CH1, CH3	Server-Side Context Awareness	User Profiling, Environment [18]
Data Analysis	Scoring Metrics	CH4	FScore(2), FPR	Recall (Coverage), Accuracy [9], Look-ahead Time [3]
Data Analysis	Selection of Indicators	CH2, CH5	Goodness of Fit, Filtering	Profiling [39], Manually Identified by Experts
Data Analysis	Detection Algorithm	CH1, CH2, CH4	SPS, Historical Checks	Invariants [6], Sliding-Window Algorithms [43]
Data Analysis	Voting Strategy	CH1	Majority, 1-out-n, n-out-n	Weighted sum [37], median, mean and plurality [38].

building, including advanced filtering and scoring techniques aimed at selecting the most relevant ones. Moreover, in [19] the authors applied SPS to detect the activation of software faults in an *Air Traffic Management* (ATM) system that has a defined set of services and predictable workloads. Observing only OS indicators, SPS allowed anomaly-based error detection with high scores.

While the works mentioned above target semi-static systems, ALERT [9] aims at triggering anomaly alerts to achieve just-in-time anomaly prevention in dynamic hosting infrastructures. The authors propose a novel context-aware anomaly prediction scheme to improve prediction accuracy. In [4], we tackled the problem of performing anomaly detection in dynamic systems by adapting the approach in [19] to work in a dynamic context, where a multi-layer anomaly detection strategy was instantiated on the prototype of Secure! [11]. The results achieved showed that analysing Secure! without adequate knowledge on its behavior did not lead to a satisfactory solution. We obtained a high number of false positives and negatives because boundaries between expected and anomalous behaviour were not identified properly. Further, the lack of information on the current state of the system *does not allow detecting contextual anomalies*.

We show that information on the context can be used to improve anomaly detection in dynamic systems [20]. Context-awareness usually refers to knowledge of the user environment that is used to improve the performances of web services [18]. Differently from that, in MADneSs we introduce *server-side context-awareness* that does not require information on the user. Instead, we investigate the context defined by the services that are running at application layer. This helps defining the expected behavior of such services, also subject to frequent updates and without requiring information of their internals. These observations have been first introduced in [5]. Here we report on the general methodology, with a more extensive evaluation to put in practice our preliminary results.

2.3 Main Challenges

Anomaly detection in complex and dynamic systems is hard: we list here the intrinsic challenges that need to be tackled to build suitable detection techniques.

CH.1. Adaptive Notion of Expected Behavior. Dynamicity leads to frequent changes in the expected – and consequently anomalous - behavior. This means that the model of the expected behavior needs to be repeatedly updated [1], because its validity is going to become false through time. In [7] and [9], the authors propose self-adaptive anomaly detection strategies to deal with such evolving notion of expected behavior.

CH.2. Avoiding Interferences and Minimizing Overhead. The anomaly detection logic must not interfere with the target system: the anomaly detection framework must not steal computational resources or introduce relevant overheads e.g., during training of the anomaly detection algorithms. Intrusiveness of anomaly detection frameworks are evaluated by authors in [2], [4], [19], mainly analyzing CPU and RAM usage.

CH.3. Applicable Monitoring Strategy. Data is collected from different sources that compose the target system. However, insights of the services or components may be not observable, e.g., in case of third-party components or encapsulated components [34]. Moreover, the set of services may change, e.g., services may be updated, added or removed. This calls for a monitoring strategy that identifies viable monitoring targets and does not require manual reconfiguration when the services evolve.

CH.4. Suitable Anomaly Detection Algorithms. The anomaly detection logic needs to rapidly cope with frequent changes of the expected behavior. Algorithms that require a massive training effort - such as clustering [7] or Markov-based models [2] - are not adequate when the system changes frequently.

CH.5. Selection of the Indicators. To minimize impact on systems and/or networks, monitors should observe all and only the minimum set of features (indicators) required by the anomaly detector. For example, indicators values obtained observing the network layer are generally suitable for intrusion detectors [21], [23], while the operating system is usually monitored when detecting performance issues [4] or malware [17]. Other studies on the selection of the indicators are in [6] on the filtering of invariants, and in [7] where authors describe how they select 14 indicators out of 653 from the *Xen hypervisor*.

3 Design Choices

We report the design choices (see Table 1) that address these challenges and that have been implemented in the MADneSs framework. For each design choice, we i) describe the general approach, ii) describe our instantiation for the case studies, and iii) report on alternative state-of-the-art strategies. Despite the approach is generic, the instantiation of each design choice may be changed according to the specific needs and the availability of sophisticated algorithms or techniques.

3.1 Data Collection

3.1.1 Monitoring Strategy

When dealing with dynamic and evolving systems, instead of instrumenting each service or application it appears more appropriate to instrument the underlying layers such as operating systems, application servers, network protocols [14] or databases [15]. This switches the observation perspective from each service to the whole system. Problems generated by services e.g., due to manifestation of errors or attacks, can be detected observing anomalies at the underlying system layers.

Our Approach. Consequently, we apply a multi-layer monitoring strategy (CH.3). The system layers that are instrumented in MADneSs are the OS, the network, and the *Java*-based middleware. We choose to investigate indicators related to OS and network since these are common layers. We selected *Java* middleware because it is the foundation of several service-based software systems, e.g., *Apache Tomcat* or *JBoss*. Depending on the specific system, layers such as database [15] may be considered.

3.1.1 Context-Awareness and Contextual Information

Context-awareness can facilitate the description of the expected behaviour of the services (CH.1, CH.3). This approach was suggested by [45], where authors demonstrated how a highly precise context-sensitive program representation allows improving static program models. Also, in [46], the context helps the data analyst making decisions: the context is the latest data triage operation, which change every time the analyst performs new operation. In fact, *contextual information* has a key role in defining boundaries between expected and anomalous behaviour. For example, let us consider a user that invokes a "store file" service at time *t*. We can combine contextual information with information on the current behaviour of the service i.e., a *fingerprint*, which here concerns data transfer. Therefore, if the "store file" service is invoked at time *t*, we expect an exchange of data during the majority of the service. If no data is exchanged, we can reveal that something anomalous is happening.

Our Approach. We refer to *server-side* context awareness. In SOAs, web-services share common information through an *Enterprise Service Bus* (*ESB*, [27]) that is in charge of i) integrating and standardizing common functionalities, and ii) collecting data about the services. The ESB provides knowledge on the services running at any time *t*. Moreover, we define a *fingerprint* for each service of the SOA, composed by statistical indexes i.e., *average*, *median*, *standard deviation*, related to the expected usage of each monitored indicator while a specific service is running. The running services and their fingerprint build contextual information used for more accurate analyses. For example, the ESB acquires information as the time instant a web-service is called or replies, or the amount of data exchanged through a service invocation, and logs them to make such information available to other processes. We do not require knowledge on the user, contrary to what is typically done when monitoring systems based on (web) services [18].

3.2 Data Analysis

3.2.1 Scoring Metrics

The basic measures are correct detections - *true positives* (TP), *true negatives* (TN) - and the wrong ones, either missed detections (*false negatives*, FN) or false detections (*false positives*, FP). More complex measures based on the basic ones are *precision*, *recall* (or *coverage*) and $F-Score(\beta)$ [10]. Especially in the $F-Score(\beta)$, varying the parameter β makes it possible to weight *precision* and *recall* (note that $F-Score(1)$ is referred as *F-Measure*).

Our Approach. Since we are mostly targeting critical systems, we prefer to reduce the occurrence of missed detections (FN), even at the cost of a higher rate of FP. For this reason, our reference metric is $F-Score(2)$, which weights the recall double than the precision. However, since anomalies and related errors are supposed to be rare events, using only precision and recall is not a good choice as they do not account for TNs. Thus, in combination with $F-Score(2)$, we use *False Positive Rate* (FPR), or rather the ratio of incorrectly detected anomalies to the number of all the correctly labeled expected instances (TNs).

3.2.2 Detection Algorithm

Monitored data are processed by the selected anomaly detection algorithm(s). To cope with the dynamicity of the system, adaptive anomaly detection techniques need tailoring their parameters on the current context (CH.1). *Self-adaptive* and *online machine learning* algorithms allow detecting observations that do not follow the inertia of their trend with reduced computational or memory demands (CH.1, CH.4) [19]. The subsequent observations of the value of an indicator are managed through a *sliding window* mechanism [43], which keeps track of the past elements related to such indicator. Past elements are used by the algorithm to build a prediction, or to build an acceptability range for the current value of the indicator.

Our Approach. MADneSs performs anomaly detection by adopting two strategies: i) *Historical Checker* (HIST), which checks if the current data instance complies with the expectations defined through contextual information, and ii) the *Statistical Predictor and Safety Margin* (SPS, [19]) algorithm. This algorithm predicts the next value of a sequence of observations depending on a statistical analysis of the past values. The prediction produces an interval of two values in which the next value is expected to fall. If the next value is outside the interval, SPS signals that this data point is anomalous. It is worth noting that SPS identifies point anomalies, but does *not* identify groups of subsequent similar anomalies (collective anomalies).

3.2.3 Selection of Indicators

The monitoring activity produces a sequence of observed values for the available system performance *indicators* (e.g., memory usage, network packets sent), to provide a complete vision of the state of the monitored machines, because they embrace different aspects of the monitored system. However, monitoring all the system indicators is not usually allowed, and may not be a proper choice, since the amount of data that generated is challenging to be analysed without incurring in delays (CH2). An accurate selection is thus required to optimize the whole monitoring and data analysis environment. As example, studies in [39] were directed to find the smaller set of indicators providing good detection scores in different systems (CH5). Moreover, it is difficult to catch possible relations when observing each indicator separately. An invariant-based approach to detect faults by identifying dependencies between indicators was proposed in [40] and then expanded in [6]. Briefly, *invariants* are stable relations among indicators that are expected to hold: a broken invariant reflects an anomalous state of the system.

Our Approach. Our initial set of indicators is composed by i) *simple indicators*, or rather specific system indicators, and

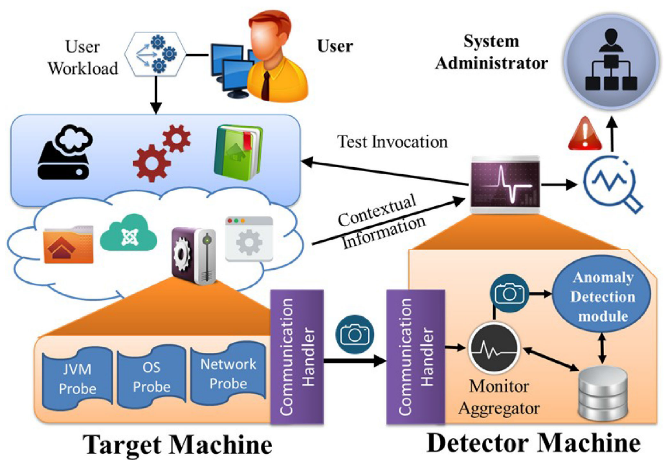

Fig. 1. High-level view of the MADneSs framework.

ii) *composed indicators*, couples of indicators that are linked by a given relation. Composed indicators are identified according to invariants as described in [40]. When a given invariant between two simple indicators shows a *Goodness of Fit* greater than a given threshold, a composed indicator is created. Then, the set of indicators, either simple or composed, to be used at runtime is filtered during training, discarding indicators that are not informative (e.g., semi-constant values) or extremely unstable, generating multiple false alarms (i.e., high FPR scores).

3.2.4 Voting

The outputs of HIST and SPS for different indicators are collected at the end of the data analysis process and are used to evaluate if the state of the system is anomalous or not. The way they are combined heavily impacts on the outcomes of the anomaly detection process. To such extent, in the literature several works on *voting strategies* in n-modular redundant systems [37], [38] were primarily proposed as adjudicators for improving fault tolerance of redundant systems. Noticeably, strategies as *majority*, *median*, *mean*, *k-out-n* and *plurality voting* are still relevant when aggregating different individual results.

Our Approach. We assume of each algorithm that searches for anomalies for a given indicator provides boolean results: therefore, *majority*, *median*, *mean* and *plurality voting* lead to the same scores. As a result, we will consider *majority* and *k-out-n* strategies for the final voting of the single anomaly scores. It is worth remarking that the strategies above consider all the single scores having the same relevance, or reputation. In many cases, it would be safe to weight detected anomalies in different trends of indicators differently depending on the context e.g., the running service(s). As we do not have any trustable way to assign weights, we use a voting strategy that gives the same weights to our indicators.

3.3 Point, Contextual and Collective Anomalies

Our design approach supports the detection of *point*, *contextual*, and *collective* anomalies (CH.4).

SPS identifies values that do not follow the statistical inertia of the trend of observations, thus detecting point anomalies. Information about the context makes us able to check if the observed behaviour is compliant with the expected behaviour defined by contextual information i.e., *HIST strategy*. Consequently, this makes possible to identify contextual anomalies.

Dealing with collective anomalies is generally difficult. In some cases, algorithms catching either point or contextual anomaly can also successfully identify collective anomalies. However, collective anomalies may not differ significantly from the expected trend in a given context, or they can be erroneously evaluated as a new trend resulting from system dynamics. In addition, background noise negatively influences their detection since it generates fluctuations that may be misinterpreted as anomalies. To cope with this specific and often tricky category of anomalies, we take advantage of *composed indicators*. As described earlier, composed indicators identify stable relations among couples of indicators that are overall less sensitive to noise than simple indicators, reducing false alarms.

4 MADneSs Framework

We describe here the MADneSs framework, which implements the design choices described in Section 3.

4.1 Architectural Overview

In Fig. 1 we depict a high level view of MADneSs; from left to right, the framework can be described as follows. The users execute a *workload*, which is a sequence of invocations of services hosted on several physical or virtual *target machines*. One or more target machines can be monitored as shown in the bottom left of the figure. In each target machine, *probes* are instrumented, observing the indicators related to 3 different system layers: i) *OS*, ii) *middleware* (*Java Virtual Machine*, JVM) and iii) *network*. These values are collected by custom probes aimed at minimizing the disturbance of target system (CH.3).

These probes repeatedly collect data on the *Target Machine*, sending them to the *communication handler*, which forwards data to the *communication handler* of the *Detector Machine*. Here, the *monitor aggregator* encapsulates probes data in a snapshot. The snapshot is then coupled with the fingerprint of the running service that is obtained through tests e.g., *average*, *standard deviations*, on the expected trend of indicators, and stored in the database (see the bottom-right of Fig. 1). More in detail, once changes in the services are detected, tests are run (*test invocation*) to gather a novel or updated fingerprint of each updated service. In our implementation, the fingerprint is used together with *contextual information* extracted from the ESB. Executing the monitor aggregator on a separate machine allows i) reducing intrusiveness on the target machine (CH.2), and ii) connecting more machines to the same detector machine.

Finally, the snapshot and the fingerprint are then sent to the *Anomaly Detection module*. If it evaluates the snapshot as anomalous, it creates an object containing all the information related to such anomaly e.g., which indicators are having anomalous behaviors.

Anomaly alerts activate automated diagnosis strategies e.g., testing quality of services [50], while the administrator is alerted or recovery strategies executed only if the malfunction is confirmed. Detailed diagnostic routines, countermeasures

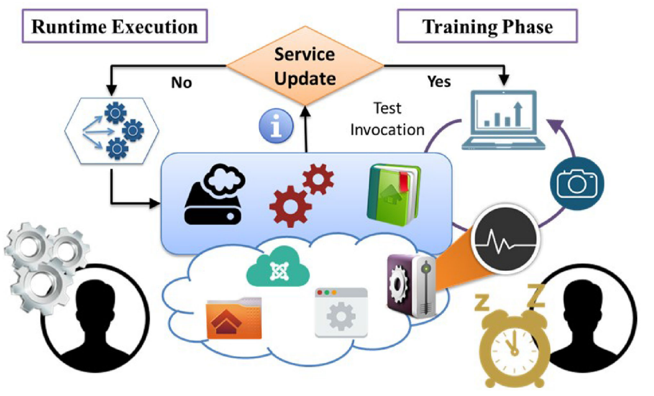

Fig. 2. Methodology: the system is executed until a service update is triggered. In that case, the *training phase* starts targeting one or more target machines while the user is waiting (right side). Then, the *runtime execution* phase starts again with anomaly detection support (left side).

or reaction strategies are outside from the scope of this work and will not be elaborated further.

4.2 Insights on the Monitor Aggregator

Data Categories. Periodically (once per second), the Monitor Aggregator of the Detector Machine in Fig. 1 provides to the anomaly detection module a *snapshot* of the observed system status, composed of the quantities retrieved from the probes installed on the Target Machine(s). For each indicator, two quantities are provided: i) *PLAIN*: the current probes' observation, and ii) *DIFF*: the difference among the current *PLAIN* value and the previous one.

Data Series. A data series is defined as a triple $< indicator, data\_category, series\_layer >$. *Indicator* represents the indicator responsible for the set of data we are analyzing. For example, this can be the usage of the memory, the number of accesses to the hard disk or the number of active threads managed by the OS. *Data_category* specifies if the data series refers either to PLAIN e.g., 234 threads are currently active, or to DIFF elements e.g., 2 threads were created since the previous observation. Lastly, *series_layer* describes the system layer from which the data are collected. Noteworthy, MADneSs also works with composed indicators; in this case, the *series_layer* is labeled as CROSS. For the sake of simplicity, in our study composed indicators are obtained only through linear combinations using a single arithmetic operator $(-, /)$.

4.3 Insights on the Anomaly Detection Module

The Anomaly Detection module includes a set of *anomaly checkers,* selected according to a given *metric*. An anomaly checker is assigned to a given *data series*, and evaluates if the current value of such data series is anomalous following a set of rules. Two or more anomaly checkers can be created for the same data series. Taking a snapshot as input, each anomaly checker produces a score; then, individual outcomes of the set of anomaly checkers are combined to decide if an anomaly is suspected. Consequently, an anomaly is raised only if this combined score reaches or exceeds a given treshold.

Anomaly Checkers. For each data series, we build two anomaly checkers:

- *Historical (HIST)*: implements a contextual check by comparing the values of a given data series with the expectations contained in the fingerprint. If this quantity is outside of the interval defined by *average* ± *standard deviation* contained in the fingerprint, an anomaly is raised.
- *SPS*: for a given data series, this anomaly checker applies the SPS algorithm described in [19].

Selected Anomaly Checkers and Anomaly Threshold. Once the metric is defined, it is used to automatically detect the best configuration of each anomaly checker. MADneSs detects anomalies depending on a set of anomaly checkers that are selected from the pool of available ones according to specific rules. The *selected anomaly checkers* are extracted either by choosing:

- BEST x: the x anomaly checkers that have the best scores according to the metric e.g., BEST 5 (B5) represents the 5 checkers with higher $FScore(2)$, or
- FILTERED y: the y checkers with the best metric scores, filtered to avoid having two anomaly checkers exercised on the same data series. For example, it avoids selecting SPS and HIST anomaly checkers on the same "*HeapMemoryUsage*" PLAIN data series.

Having selected the anomaly checkers, we chose the appropriate *anomaly threshold* as follows. A snapshot is voted as anomalous if at least a threshold of the anomaly checkers raise an anomaly. We propose different approaches to set this threshold, namely

- ALL: all the selected anomaly checkers evaluate their data series as anomalous;
- QUARTER / THIRD / HALF: at least a quarter / third / half of the selected anomaly checkers evaluate their data series as anomalous;
- ONE: the snapshot is evaluated as anomalous if at least one of the anomaly checkers raises an anomaly.

It is important to remark that the choice of the threshold heavily affects the overall detection performance. The usage of a single checker (i.e., adopting ONE) reduces the amount of false negatives; instead, a consensus among anomaly checkers, e.g., ALL, reduces the number of (false) alarms.

4.4 Methodology to Exercise the Framework

The methodology to exercise MADneSs is composed of two steps to be repeated when major reconfigurations occur: *Training Phase* and *Runtime Execution*.

Training Phase. This phase is organized in two steps. In the first step, fingerprints are obtained through the *test invocation* in Fig. 2. Then, *preliminary runs* exercising the expected workload are executed, storing the obtained data in the database. Preliminary runs are conducted by either i) observing the behavior of the system through functional tests, or ii) injecting anomalies in one of the SOA services, and observing the effects they generate on the monitored indicators. Preferably, the service in which anomalies are injected is a custom service devoted exclusively to testing, allowing to modify its source code. This strategy is particularly useful when we cannot inject faults into the services exposed by the target system.

During this step, the services are not open to users, which consequently are waiting until the SOA is available again. When the SOA is first deployed, deploy is completed only after completion of step i) and ii). Once the SOA is available to

TABLE 2
Models of Anomalies Adopted in Different State-of-the-Art Studies

Framework			Anomalies			
Name	Target System	Dynamicity	Reconfiguration	Misconfiguration	Interaction	Resource Usage
[8]	-	-	✓		✓	
CASPER [2]	Air Traffic Management	Very Low				✓
[6]	Distributed Web App	Low	✓	✓		✓
SEAD [7]	Cloud Environment	Low				✓
TIRESIAS [3]	Distributed Environment	Low				✓
[19]	Air Traffic Management	Low				✓
ALERT [9]	Cluster Environment	Medium				✓
[4]	SOA	High				✓
MADneSs	**SOA**	**High**		✓	✓	✓

users, it is expected that only few services will be updated each time, requiring specific tests and consequently short periods of unavailability. Moreover, to avoid bothering the user, preliminary runs can be exercised in low load periods such as at night or on mirror systems. Scalability and solutions to reduce training times are explored in Sections 6.3 and 6.4.

In the second step, services information and data extracted from preliminary runs are used by the anomaly detection module to train its parameters (CH.1), automatically choosing the best *selected anomaly checkers* and threshold.

Runtime Execution. The fingerprints used during training are now used by the *Monitor Aggregator* to build a complete snapshot. Along with contextual information, such snapshots are sent to the anomaly detection module: depending on its outcomes, an anomaly alert is raised. If a service update is detected during this phase, a new *training phase* is scheduled. The scheduling policy is strictly dependent on the characteristics of the system, and it is outside of the scope of MADneSs.

4.5 MADneSs Anomalies Model

We identify the model of anomalies that MADneSs aims to detect. To such extent, we review well-known anomaly models from the literature identifying which of them are relevant for MADneSs. The anomalies in such models cause perturbations on the expected behaviour of system indicators and should thus trigger detection. They describe common manifestation of attacks or faults but here we are not interested in identifying their precise root cause, i.e., the specific individual attacks or faults which generated the anomaly. In Table 2, we show the anomaly models adopted by the frameworks analyzed in Section 2.2. Most of these frameworks consider anomalies in the resource usage; while in [6] authors consider reconfigurations and erroneous configurations (misconfiguration) of parameters as sources of anomalies. Concerning reconfiguration, the re-training of MADneSs every time a reconfiguration is detected makes the detection of these anomalies out of scope. Instead, we do include anomalies concerning misconfiguration in our model.

Anomalies due to interaction among modules and components of the complex system are sometimes considered as well. In particular, [8] defines a set of behaviors that can emerge in complex systems, such as: i) *deadlock*/*livelock*, ii) *trashing*, iii) phase change, iv) *synchronization* and *oscillation*, and v) chaotic. Similarly to reconfigurations, phase changes are considered main system variations, thus calling for a new training and not considered in our model. Furthermore, since the Anomaly Detection module treats each Target Machine individually, synchronization, oscillation and chaotic behaviors have minor impact or likelihood. Additionally, our middleware (JVM) automatically controls thrashing: the JVM manages the garbage collection and context switches with the objective of avoiding performances degradation. Instead, deadlock and livelock are considered in our model as they are possible cause of anomalies in complex systems.

Consequently, the resulting model is composed by i) performance anomalies, and in particular we identify in this paper four anomalies that we name MEMORY, CPU, DISK, NETWORK, ii) anomalies due to deadlock/livelock, that we label as DEADLOCK, and iii) anomalies due to misconfigurations. More in detail, we will focus on misconfiguration of the network permissions, labeling them as NETPERM.

4.6 Implementation

The implementation of the framework is divided in Target Machines and Detector Machine, and detailed as follows.

Target Machine. First, we assume that the target machines have a *Linux* OS and use a Java-based application server to run (web)services. OS and Network probes are shell modules that read data from the */proc* virtual filesystem of the *Linux* distribution, while the JVM probe consists in a Java module accessing performance data through the *Java Management Beans* (MBeans). These three probes monitor 55 indicators: 23 from the *OS*, 25 from the *Java*, and 7 related to the *network*. These probes are coordinated by a *Java*-based communication handler that manages the collection of data, encapsulates them in *JSON* format and sends the data through a *TCP* socket. The version of Java required on the target machine(s) to be instrumented with our probing system is 7 or higher. Here we remark that despite several enterprise solutions providing monitoring facilities exist, we chose to develop our own probes building on [44], [19] to limit the intrusiveness of such monitoring tools. Intrusiveness of the adopted probes was already studied in [4].

As indicators, let us consider our set of 55 indicators - selected as in Section 3.2 - and nc composed indicators, which are combined using the $-, /$ matemathical operations obtaining $2 * nc$ novel composed data series. Considering both PLAIN and DIFF data categories for each indicator, we obtain 110 separate single data series and $4 * nc$ composed data series. On each data series we can instantiate either the SPS or the HIST algorithm, totalizing $2 * (110 + 4 * nc)$ possible anomaly checkers. The set of checkers to be used is reduced and then ranked during the training phase, according to the chosen metric.

Detector Machine. The engine of the Detector Machine is based on the Complex Event Processor *Esper* [26], an open-source software based on Event Stream Processing techniques. Esper has been developed specifically to process huge amounts of data in near real-time by means of SQL-like queries called *rules*. The Esper engine facilitates collection of data from different Target Machines, managing the different trends in parallel. Further, it facilitates the combination of snapshots and fingerprints performed by the Monitor Aggregator, and the application of the Anomaly Detection module. In particular, the Anomaly Detection module is realized in a multithreading code that fetches the data from a *MySQL* database and instantiates the anomaly checkers on the data series provided by the Monitor Aggregator. The test invocations are generated through a sequence of SOAP invocations, which are executed through *Apache AXIS* calls.

Tailoring MADneSs to Other Systems. MADneSs can be applied to any other service-oriented system as follows. Suppose that you want to setup a server that uses the *OwnCloud* [52] suite to manage a private repository of sensitive data, events, documents and contracts. As opposed to our examples, the server will run a *Windows Server* distribution and OwnCloud will be supported by a *MySQL* database. To successfully setup and exercise MADneSs, a system administrator should devise the main layers and preliminarily trying to figure out where meaningful indicators may be located. In this case, since there will be lots of data exchanges, the administrator targets *Network*, *OS* and *Database* layers. Then, administrator has to define probes for each layer e.g., using the *Performance Counters API* for *Microsoft OS*, or activating a packet sniffer as *Wireshark* to gather network. MADneSs is able to read CSV files created by probes by adjusting a preferences file. In any other case, the administrator has to extend the source code of MADneSs by creating a new *Java* class extending *CycleProbe* and implementing the method *readParams*, that defines how the probe gather data at each instant of time. Further, the administrator has to setup a strategy to derive the active services at a given time, e.g., a connector to *occ* – the interactive OwnCloud shell, to provide contextual information. Then, MADneSs is ready to be used, requiring minimal additional tuning e.g., specifying the *IP address* and *port* of Detector Machine, where the monitored data is sent for analysis. Note that the code running on Detector Machine does not mandatorily require specific setup, while preferences to optimize the framework may be set e.g., if intermediate data or anomaly scores of snapshots should be stored in a DB.

5 Experimental Evaluation

We describe the experimental evaluation of MADneSs. To the purpose of the evaluation, we run an automatic controller that checks input data and manages the communications among the Target and the Detector Machine. This facilitates the automatic execution of the experimental campaign without user intervention, except for the setup. All data is available at [32], including additional files we do not report here for brevity.

5.1 Description of Our Case Study

The Target and Detector Machine are virtual machines running on a rack server with 3 Intel Xeon $\mathrm{E5}-2620@2.00$ GHz processors. The Target Machine, which hosts either the *Secure!* or *jSeduite* code, is instrumented with the probing system which produces 1 snapshot per second.

Secure!. Our Target Machine is one of the four virtual machines that host the *Secure*! crisis management system [11]. Secure! is built on the Java-based *Liferay 6.1.1* [12] portal on *Apache Tomcat 7.0.40* as application server, and it exposes web services such as authentication mechanisms, file storage, and calendar management. We identified 11 different web services that can be invoked by the Secure! users, and we created the *All Services* workload that invokes them with different orders, with a time interval of 1 second between successive invocations. One execution of the workload lasts approximately 65 seconds.

jSeduite. jSeduite is a SOA dealing with information broadcast inside academic institutions. It is composed of atomic web services representing information sources and orchestrations of business processes [41]. Some of the available services cover basic operations as *ErrorLogger*, *DataCache*, *FileUploade*, *FeedRegistry*, and *TwitterWrapper*, that represents pillars on which safety and/or security critical services are built upon. We distributed *jSeduite* on a *Glassfish Java*-based server, setting its services to use a *MySql* database.

5.2 Injection Approach

The injection of anomalies was performed by code mutation in different injection points.

Injection Points. In Secure!, the injections targeted i) the *com.liferay.portlet.documentlibrary.store.FileSystemStore* in charge of managing the addition of a directory in the *Liferay* filesystem, ii) the encoding strategy involved in the exchange of data between the database and the UI showing the calendar (*com.liferay.portlet.calendar.service. impl.CalEventLocalUtil._encodeKey function*), and iii) the creation of a SOAP model (function *com.liferay.portlet. bookmarks.model.BookmarksFolderSoap.toSoapModel*) describing the response of the request of adding a bookmark in the user data. The injection is triggered by a timer expiring approximately at 70, 80, or 90 percent of the workload: the code is mutated only when the timer expires. This leads to 9 possible ways of injecting anomalies in Secure!: 3 injection points *per* 3 injection instants. An example of XML workload can be found in [36].

With *jSeduite,* the injections were performed in 10 different functions related to 4 of the selected 8 web services (*fr.unice.i3s.modalis.jSeduite.technical* package). More in detail, we instrumented *TvHelper* (*tv.extract*), *TwitterWrapper* (*messaging.twitter.{getIntendedTweets*, *getChannel*, *getFreeTokens}*), *ApalWrapper* (*apal.{getTopWithTreshold*, *getTop10*, *getLoosers*, *getPromos}*), and *FeedRegistry* (*registry.{getURL*, *getCategories*, *getNicknames*) services.

Injection Approach. We inject a single fault in each individual run. Noteworthy, we are interested in detecting the *first* anomaly that is generated after the activation of the fault and that can be explained by the fault itself. In fact, the manifestation of a single fault can lead to several cascading effects on the system, with consequent variations from the expected behavior i.e., multiple anomalies. The complexity of our target system does not allow studying the propagation effects of the injected faults. Therefore, we are not able to distinguish *if* and especially *how* different anomalies are related. It follows that only the first anomaly that is detected after the activation of the injected

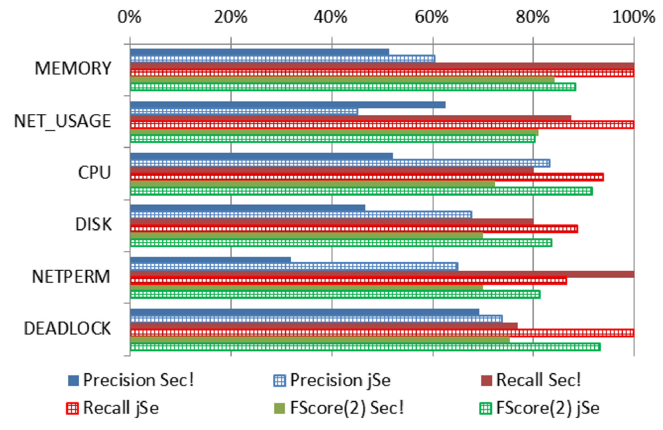

Fig. 3. Values of precision, recall, and FScore(2) for different injections.

fault can be considered a true positive (TP) in our analysis. Further, this anomaly must be detected within a limited temporal distance from the injection time instant; this way, we are confident that the anomaly is a consequence of the injected fault and not a false positive (FP). We experimentally set this temporal distance to 2 seconds. Successive anomalies, which can occur due to i) new manifestation(s) of errors, ii) cascading effects of the injected faults, and iii) false positives, are ignored.

5.3 Experimental Campaign

We first conducted 90 golden runs in which we executed the workload without any injection. Then, we performed runs with injections for the 6 anomalies that are present in our model of anomalies. For each anomaly, we executed 10 runs for injection point identified, as discussed before.

Overall, 630 and 690 experiments were conducted on *Secure!* and *jSeduite* respectively. This number was defined observing the standard deviation for the results, which was acceptable. To guarantee the correctness and repeatability of the experiments, we reset the Target Machine before each experiment. This also provides independency among subsequent experiments. The experiments above served i) as preliminary runs for the training phase of MADneSs, and ii) to validate the efficacy of the framework itself. In particular, the 90 golden runs and 70 percent of the others (e.g., 378 runs with Secure!, i.e., 63 runs for each anomaly) were used for the training phase. The remaining runs (e.g., 162 for Secure!) were used for the validation of the framework.

5.4 Results: Detection Efficiency

The results of the experiments in terms of precision, recall, and FScore(2) are depicted in Fig. 3. Each set of bars reports on the results with the injection of a given software fault: as example, the first set is related to injections of a MEMORY anomaly. Moreover, each set of bars reports the results related to our two case studies, obtained by using the optimal setup discovered during training to detect specific anomalies. Results related to jSeduite are reported using bars with solid fill, while the Secure! ones are bars filled with a vertical-striped pattern.

Overall, we can observe how recall scores (*Recall Sec!* and *Recall jSe)* are significantly higher than their precision counterparts. This is due to the choice of targeting FScore(2) as reference metric for the whole process favors anomaly checkers that give higher recall scores. Consider though that in our

TABLE 3
Relevant Indicators for Experiments of Both Secure! and jSeduite

Secure!		jSeduite	
Indicator	Layer	Indicator	Layer
HeapUsage.committed	CENTOS	*CPU Kernel Processes*	CENTOS
CPU Kernel Processes	CENTOS	Minor Page Faults	CENTOS
ThreadCount	JVM	*ThreadCount*	JVM
Tcp_Syn	NETWORK	*CollectionCount*	JVM
Free Virtual Pages	JVM	Active Memory	CENTOS
CPU Soft Interrupts	CENTOS	CPU Soft Interrupts	CENTOS
Major Page Faults	CENTOS	Net_Sent	NETWORK

scenario when anomalies are injected we can have at most one true positive, therefore in such scenarios false positives are a better indicator than precision. Regarding the MEMORY experiments in Fig. 3, all the injected anomalies were detected i.e., recall scores are optimal at the price of a few false alarms, which make precision scores lower than recall. The NETPERM experiments resulted in 2 false positives this was the highest number of false positives observed. The average was 1.48 false positives per experiment, (with a FPR of 2.07).

Depending on the system, a given rate of false alarms can be either a strong limitation for the application of our technique or a reasonable price to pay considering the challenges identified in Section 2.3. In any case it should be considered that anomaly alerts normally activate automated diagnosis strategies e.g., testing quality of services [50], while the administrator is alerted or recovery strategies executed only if the malfunction is confirmed.

5.5 Results: Indicator Selection

Table 3 reports on the indicators which overall give the more relevant contribution to anomaly detection in all our experiments. The detailed list of the indicators and anomaly checkers for each category of anomalies can be found in [32]. The indicators are ranked in Table 3 according to the $TOP\_10\ Avg\ FScore(2)$ index, which is not shown for brevity. We compute the average FScore(2) of the 10 better anomaly checkers which process data of either a simple or a composed indicator. For each indicator, we report the name and its layer for both the Secure! and jSeduite. Indicators regarding the three layers are mentioned in the table, highlighting how OS-related information resulted more useful than others for detecting anomalies.

The list in Table 3 is *not* intended to be a selection that is valid for any complex system. This is the list of the indicators that scored better in our case studies: the list may vary depending on the target system, although some indicators (reported in italic font in Table 3) seem to be relevant in both case studies. A guide for the selection of the appropriate data series when using MADneSs is further expanded in Section 6.5.

5.6 Results: Sensitivity Analysis

We apply all the possible combinations among the *selected anomaly checkers* to identify the setup that ultimately gives a higher FScore(2). In Fig. 4 we report the graphical result of the sensitivity analysis on the runs we used as validation of our framework to detect MEMORY anomalies. On the horizontal axis, are listed the different strategies for the choice

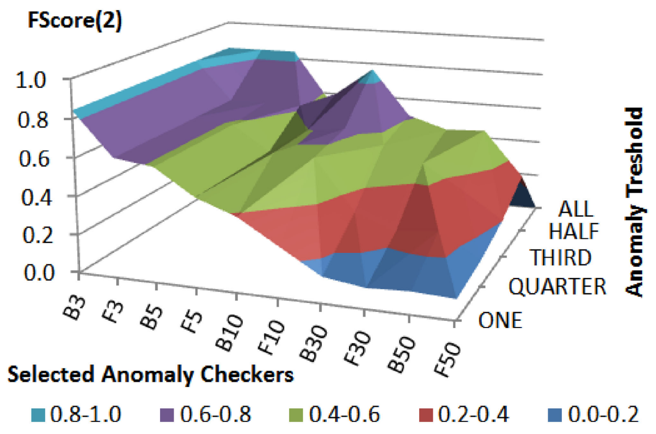

Fig. 4. Sensitivity analysis aimed at finding the best combination of anomaly checkers for the detection of MEMORY anomalies.

of the selected checkers. On the depth axis, instead, the anomaly thresholds.

Regarding the detection of the anomaly MEMORY, an optimal setup is represented by F10 (HALF), as can be observed in Fig. 4. F10 (HALF) labels a snapshot as anomalous if at least five anomaly checkers trigger an anomaly for that snapshot. Other optimal setups can be obtained by considering the B3, that raises an anomaly either if i) at least one out of three checkers detects an anomaly (ONE, QUARTER, THIRD on the z-axis of Fig. 4), or ii) all three anomaly checkers raise anomalies (ALL in the z-axis). Results of the sensitivity analyses for the other injected anomalies are not reported here for brevity, but lead to similar observations.

5.7 Detecting Unknowns

Beyond the results presented in Section 5.4 and depicted in Fig. 3, we evaluated also the capabilities of MADneSs in identifying *unknown anomalies*, which in the security domain can represent the manifestations of *zero-day attacks*. To such extent, we executed a different training of the framework. When aiming at detecting anomalies linked to resource usage, we used as training set: i) the golden runs, and ii) runs with injection of either NETPERM or DEADLOCK anomalies. Instead, runs with the injections of MEMORY, NETWORK, CPU or DISK anomalies—together with the golden runs—were used to train MADneSs when looking for NETPERM or DEADLOCK anomalies. Consequently, when looking for a given anomaly, during training phase we did not include this specific type of anomaly, that therefore can be considered unknown in this particular setup.

Results of this complementary analysis are reported in Table 4. Scores related to anomalies detected as unknowns (right part of the table) are obviously lower wrt. the case of training but still very good considering they are unknown. The 7th column of Table 4 tells us that MADneSs is able to identify on average 82 percent of the previously unseen anomalies we injected. Clearly the price is higher false positives, as background noise could often result in a false alarm. As example, regarding DEADLOCK we can observe in Table 4 that recall is a bit higher when detecting anomalies as unknowns: however, the *false positive rate* is higher compared with the results obtained when training MADneSs with such anomaly.

TABLE 4
Precision (P in the Table), Recall (R), FScore(2) (F2), and False Positive Rate (FPR) Average Scores When Detecting Specific Categories of Anomalies

Anomaly Category	Anomaly Training				Unknown			
	P	R	F2	FPR	P	R	F2	FPR
MEMORY	0.56	1.00	0.86	2.25	0.37	0.85	0.67	3.24
NETWORK	0.54	0.94	0.82	2.21	0.50	0.73	0.67	1.84
CPU	0.68	0.87	0.82	1.62	0.46	0.80	0.70	2.46
DISK	0.57	0.84	0.77	2.33	0.34	0.79	0.63	2.43
NETPERM	0.48	0.93	0.79	2.60	0.38	0.84	0.68	3.37
DEADLOCK	0.72	0.88	0.84	1.38	0.37	0.92	0.71	3.75
Average	**0.59**	**0.91**	**0.82**	**2.07**	**0.40**	**0.82**	**0.68**	**2.85**

Columns on the left regard scores obtained by MADneSs after ad-hoc training, while columns on the right indicate the detection scores when MADneSs is not trained to identify such anomaly, which is unknown.

5.8 Results: Performance

According to our methodology, we need to build the fingerprints of the services. Here we investigate the time required to i) exercise the workload by conducting preliminary runs, and ii) analyse these data to characterize services, to obtain the parameters of each anomaly checker.

Preliminary Runs. We computed the time needed to test i) a single web service, and ii) all the web services on the Secure! system. Overall, testing a single service once requires between 8 and 12 seconds, while a single invocation of all the 11 services in a row requires approximately 72 seconds. Once preliminary runs are conducted, and services information is stored in the database, data is analysed to select the best combination of parameters for the anomaly checkers. The performance of this operation is strictly dependent on the characteristics of the anomaly checker, and on the amount of training data that is used to select the best configuration.

Training Times. We measured the time needed to select the best anomaly checkers when processing Secure! data, using all the 468 training runs. For (*average, median, standard deviation*), each SPS anomaly checkers took $(42.68, 42.21, 1.15)$ *ms*, while each HIST needed $(0.45, 0.33, 0.19)$ *ms*. While these are short, we remark that they are related to a single anomaly checker. In the worst case, when all the possible 612 (49 composed indicators are selected in Secure!) anomaly checkers are considered(e.g., at the start of the system), we need 814.5 and 8.7 seconds to select the best configurations of anomaly checkers respectively for SPS and HIST for a single set of 63 training runs.

Complexity Analysis. The time needed to select the best configuration is linear to the number of runs used for training. Let us consider *te* as the number of training runs, while O(SPS) and O(HIST) represent the computational complexity to train a given checker respectively for SPS and HIST using data from a single run. It follows that the worst-case complexity to select the best configuration of a checker on a single experiment can be approximated as $O(te) * max\{O(SPS), O(HIST)\}$. In other words, the complexity of the training phase of MADneSs is strictly related to the complexity of the heaviest anomaly detection algorithm implemented (SPS in our case).

TABLE 5
Six Anomaly Detectors, Where Technical Advancements are Progressively Introduced Until Reaching MADneSs

#	Layers	Data	Context-Aware	Composed Data Series
[19]	OS	*PLAIN*	NO	NO
[4]	OS, JVM	*PLAIN*	NO	NO
iii	OS, JVM, Network	*PLAIN*	NO	NO
iv	OS, JVM, Network	*PLAIN, DIFF*	NO	NO
v	OS, JVM, Network	*PLAIN, DIFF*	YES	NO
MADneSs	**OS, JVM, Network**	***PLAIN, DIFF***	**YES**	**YES**

Table 6
Comparing Metric Scores of MADneSs with Respect to Frameworks in Table 2

Framework		Metric Scores (%)			
Name	Monitored Layers	Precision	Recall	FScore(2)	FPR
CASPER [2]	Network	88.5	76.5	78.6	11.3
[6]	OS,Network	76.0	99.0	93.3	n.a.
SEAD [7]	Hypervisor	n.a.	92.1	n.a.	n.a.
TIRESIAS [3]	OS,Network	97.5	n.a.	n.a.	2.5
[19] (best setup)	OS	97.0	100.0	99.3	1.9
ALERT [9]	Host	~100.0	>90.0	>90.0	<10.0
[4] (best setup)	OS JVM	35.1	44.3	42.1	4.4
MADneSs (avg)	OS,JVM,Network	59.1	91.2	82.3	2.1

6 Comparisons and Discussions

6.1 Incremental Improvements of Detection Scores

To explain the difficulties of detecting anomalies in dynamic systems, in Table 5 we show the incremental introduction of six technical advancements that led to MADneSs. These can be interpreted as six different frameworks, where the amount of detection features available is progressively increased. From the top of the table, i) consider the framework in [19], ii) adds probes that monitor the JVM middleware [4], iii) introduces probes that monitor the network layer, iv) includes the DIFF data series in addition to the default (PLAIN), for each indicator, v) uses services information in combination with context awareness, and vi) finally, extends the solution for composed data series, completing MADneSs. In Fig. 5 we report only Precision and Recall obtained when applying the 6 alternatives to our SOAs. The introduction of a technical advancement never lowers Recall, which significantly improves when more indicators and data series are added i.e., steps ii, iii, iv.

Instead, precision decreases in Secure! in the technical advancement *iv*. The introduction of the DIFF data series increases the number of data series and, consequently, anomaly checkers, without the ability to distinguish which are the most effective (this is instead done in the successive technical advancement, number *v*). As a result, the anomaly checkers are able to improve detection capability (increase of recall) at the cost of an increased number of false alarms. This is solved in the next technical advancement *v*, where information on the context supports the characterization of the expected behavior and allows contextual anomalies to be detected. Lastly, the introduction of composed data series and a sensitivity analysis lead to the final framework, and consequently to the best scores in Fig. 5.

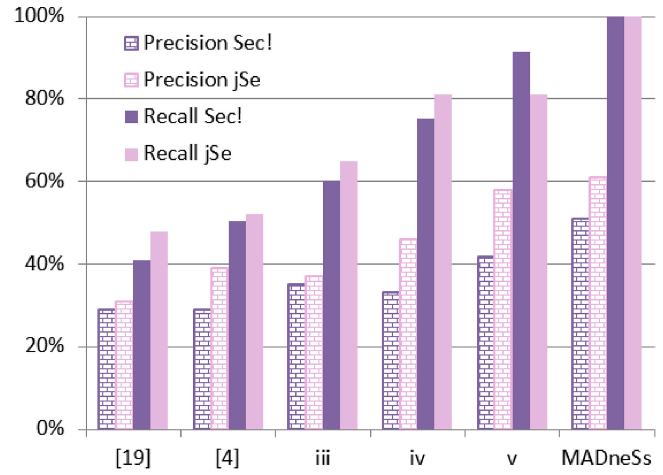

Fig. 5. Detection capabilities for the six versions of anomaly detector when looking for MEMORY anomalies.

6.2 Comparison with Respect to Other Frameworks

Surveyed Studies. In Table 6 we reported the detection performance from the surveyed studies, including MADneSs. We show only results for anomalies related to the resource usage, because they are the only anomalies common to all the surveyed studies (see column *resource usage* in Table 2). Further, authors of [8] did not report detailed information about their detection scores; consequently, we excluded this work from our comparison.

Generally precision, recall, F-Score(2) and FPR - where available - are strongly influenced by the characteristics of the target system. When the dynamicity of the system is low it is easier to define the expected behaviour resulting in a significantly lower number of FPs and FNs (as in [19] [2] and [3]) with respect to [4] and, to a smaller extent, [6] (see Table 6). High precision scores are obtained by ALERT [9], which is exercised in a cluster environment. The system shows good dynamicity, because hosts are added or removed; however, the cluster runs a fixed pool of tasks during its operational life. Instead, SEAD [7] obtains high recall scores by analysing data from a hypervisor, but no precision or FScore are reported.

Finally, MADneSs presents a recall index that is competitive with the other solutions, especially considering that we are exercising the anomaly detector on a highly dynamic system. Precision of MADneSs is low, because despite a relatively low number of false positives is observed, in our case at mot 1 true positive can be observed.

Focusing on Unknowns. In addition to the surveyed works, we refer to the context-aware studies in [47], [48] to evaluate the ability of MADneSs to detect manifestation of unknowns, either due to undiscovered faults or zero day attacks. In [47] authors use a contextual misuse detector combined with a neighbour-based algorithm to detect zero-day cyber-attacks in *static systems*. As summarized in Table 7, in their study authors reach recall and FPR scores that are lower than ours. Another work for comparison is [48], where authors detect anomalies in the frequency of service calls targeting a "synthetic" SOA system, reporting experimental data related to *k-means* and *emerging patterns* algorithms. Recall is higher than ours with *emerging patterns*, but it generates a remarkable amount of FPs. Moreover, in their experimental campaign

TABLE 7
Comparing Metric Scores of MADneSs Dealing with Unknowns

Framework	Metric Scores (%)			
	Precision	Recall	FScore(2)	FPR
[47] (local optimum $\theta = 0.8$)	n.a.	67.0	n.a.	22.00
[48] (kmeans - best)	n.a	80.0	n.a.	13.00
[48] (emerging pattern - best)	n.a	91.0	n.a.	23.00
MADneSs (avg)	40.2	82.1	68.4	2.85

authors consider a reduced (two) amount of anomalies, which impacts on the significance of the results above.

6.3 Scalability

Here we discuss how MADneSs scales when varying the resources of the Detector Machine, the number of connected Target Machines, and the monitored data.

Impact of Training. Once preliminary runs are executed, the collected data is aggregated by the monitor and analyzed to choose the best set of anomaly checkers and the most profitable anomaly threshold. This requires testing combinations of data series and detection algorithms. The cost of this operation is linear with respect to the number of data series and detection algorithms. Some algorithms require short periods of training, since they either have a limited set of parameters or simpler strategies: this is the case of the historical checker. Instead, other algorithms, such as SPS, have more parameters that need to be instantiated.

Impacts on Runtime Execution. In [4] we analyzed the intrusiveness of system probes for OS and JVM, and the network load required to transfer monitored data to a remote Detector Machine. Although the analysis did not consider network probes and the data processing was much simpler than MADneSs, main results are still valid. We empirically demonstrated that we were able to aggregate, process and analyze data coming from 5 different Target Machines at the same time. This is adequate for the Secure! system as it is composed of four machines.

In this paper we replicated the analysis, confirming the results from [4]. Further, we computed the time elapsed from the observation of a snapshot on the Target Machine until its evaluation by MADneSs. This time is 32.10 ± 5.99 ms in our experiments. It is considered fully adequate for the Secure! system.

6.4 More on Training Activity

When dealing with dynamic systems, frequent training phases may be needed to keep the parameters and the anomaly checkers compliant with the current notion of expected and anomalous behavior (see *conformal anomaly detection* [28]). In [29], authors tackle online training for failure prediction purposes i) continuously increasing the training set during the system operation, and ii) dynamically modifying the rules of failure patterns by tracing prediction accuracy at runtime. A similar approach is also used to model up-to-date *Finite State Automata* tailored on sequences of system calls for anomaly-based intrusion detection [30] or *Hidden Semi Markov Models* for online failure prediction [31]. Consequently, training phases defining the “new” expected behavior should start when specific triggers activate. We are currently considering three triggers that can be detected looking at the SOA and system setups: i) *update of the workload*, ii) *addition or update of a web service* in the platform, iii) *hardware update*.

6.5 System-Specificity of Our Findings

Experiments showed that MADneSs is able to perform anomaly detection on dynamic systems with scores that are comparable to those of other frameworks intended for more static ones. Unfortunately, we could not find other datasets collected from dynamic systems, and therefore we did not test MADneSs on systems different from *Secure!* and *jSeduite*. It is thus hard to generalize part of the findings of our work. Despite that, we believe that the design choices suggested in Section 3 are valid regardless system-specific settings, and their validity goes beyond our case studies. Moreover, the selection of indicators for anomaly detection presents some generically applicable results. For example, we observed that some indicators are especially relevant to detect certain anomalies, as the *HeapMemoryUsage.committed* is for MEMORY anomalies; we expect this trend is maintained when observing errors that manifest as memory anomalies in other systems, although further studies would be needed to confirm this.

7 Conclusions

We presented our approach to anomaly detection in complex dynamic systems. We listed challenges related to dynamic systems that negatively affect the efficacy of traditional anomaly detectors; then, we proposed design choices to address such challenges. We discussed *Monitoring Strategy*, ii) *Context-Awareness*, iii) *Scoring Metrics*, iv) *Selection of Indicators*, v) *Detection Algorithm*, and, finally, vi) *Voting Strategy*. The key aspects of our approach were integrated in *MADneSs*, a novel framework for anomaly detection that employs a large anomaly model, includes an automatic tuning of its parameters, and manages relations among system indicators. MADneSs integrates a multi-layer monitor, which allows shifting the observation perspective from the application layer—where services operate—to the underlying layers, which are subject to smaller updates, allowing instrumenting a monitor that does not require substantial maintenance.

The detection algorithm was chosen accordingly, resulting in the adaptive SPS [19] algorithm, which calculates an acceptability interval for a data instance depending on a sliding window of previous observations. Lastly, we conducted our experimental campaign by exercising MADneSs on the Secure! [11] CMS and on the jSeduite [41] SOA, showing that context-awareness improves the detection accuracy. The analysis included performance, complexity, sensitivity and scalability analyses, and comparisons with other frameworks, showing that MADneSs has potential for the considered target systems.

Acknowledgments

This work has been partially supported by the European FP7-IRSES DEVASSES projects. We thank *Dr. Paolo Lollini* for his remarkable and valuable contribution to this work.

REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surveys*, vol. 41, no. 3, 2009, Art. no. 15.

[2] R. Baldoni, L. Montanari, and M. Rizzuto, "On-line failure prediction in safety-critical systems," *Future Generation Comput. Syst.*, vol. 45, pp. 123–132, 2015.

[3] A. W. Williams, S. M. Pertet, and P. Narasimhan, "Tiresias: Black-box failure prediction in distributed systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–8.

[4] A. Ceccarelli, et al., "A Multi-layer anomaly detector for dynamic service-based systems," *Computer Safety, Reliability, and Security*. Berlin, Germany: Springer, 2015, pp. 166–180.

[5] T. Zoppi, A. Ceccarelli, and A. Bondavalli, "Context-awareness to improve anomaly detection in dynamic service oriented architectures," in *Proc. Int. Conf. Comput. Safety Rel. Security*, 2016, pp. 145–158.

[6] L. Aniello, C. Ciccotelli, M. Cinque, F. Frattini, L. Querzoni, and S. Russo, "Automatic invariant selection for online anomaly detection," in *Proc. Int. Conf. Comput. Safety Rel. Security*, 2016, pp. 172–183.

[7] H. S. Pannu, J. Liu, and S. Fu., "A self-evolving anomaly detection framework for developing highly dependable utility clouds," in *Proc. IEEE Global Commun. Conf.*, 2012, pp. 1605–1610.

[8] J. C., Mogul, "Emergent (mis) behavior versus complex software systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 40. no. 4, pp. 293–304, 2006.

[9] Y. Tan, X. Gu, and H. Wang, "Adaptive system anomaly prediction for large-scale hosting infrastructures," in *Proc. 29th ACM SIGACT-SIGOPS Symp. Principles Distrib. Comput.*, 2010, pp. 173–182.

[10] M. Sokolova and S. Japkowicz, "Beyond accuracy, F-score and ROC: A family of discriminant measures for performance evaluation," in *Proc. Australasian Joint Conf. Artif. Intell.*, 2006, pp. 1015–1021.

[11] Secure! project, secure.eng.it (last accessed on 3rd Jan 2018).

[12] Liferay, www.liferay.com (last accessed on 3rd Jan 2018).

[13] J. Zhang, et al., "EnCore: Exploiting system environment and correlation information for misconfiguration detection," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014.

[14] Y. Zhang and W. Lee, "Intrusion detection in wireless ad-hoc networks," in *Proc. ACM 6th Annu. Int. Conf. Mobile Comput. Netw.*, 2000, pp. 275–283.

[15] A. Kamra, E. Terzi, and E. Bertino, "Detecting anomalous access patterns in relational databases," *VLDB J.*, vol. 17, no. 5, pp. 1063–1077, 2008.

[16] C. Modi, et al., "A survey of intrusion detection techniques in cloud," *J. Netw. Comput. Appl.*, vol. 36, no. 1, pp. 42–57, 2013.

[17] M. R. Watson, A. K. Marnerides, A. Mauthe, and D. Hutchison, "Malware detection in cloud computing infrastructures," *IEEE Trans. Depend. Secure Comput.*, vol. 13, no. 2, pp. 192–205, Mar./Apr. 2016.

[18] H.-L. Truong and S. Dustdar, "A survey on context-aware web service systems," *Int. J. Web Inf. Syst.*, vol. 5, no. 1, pp. 5–31, 2009.

[19] A. Bovenzi, et al., "An OS-level framework for anomaly detection in complex software systems," *IEEE Trans. Depend. Secure Comput.*, vol. 12, no. 3, 366–372, May/Jun. 2015.

[20] T Zoppi, A. Ceccarelli, and A. Bondavalli, "Exploring anomaly detection in systems of systems," in *Proc. ACM Symp. Appl. Comput.*, 2017, pp. 1139–1146.

[21] S. Ponomarev and T. Atkison, "Industrial control system network intrusion detection by telemetry analysis," *IEEE Trans. Depend. Secure Comput.*, vol. 13, no. 2, pp. 252–260, Mar./Apr. 2016.

[22] A. Ceccarelli, A. Bondavalli, B. Froemel, O. Hoeftberger, and H. Kopetz, "Basic concepts on systems of systems," in Proc. *Cyber-Physical Systems of Systems: Foundations–A Conceptual Model and Some Derivations: The AMADEOS Legacy* Chapter 1, Berlin, Geramny: Springer, 2016.

[23] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2015, pp. 57–68.

[24] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, "A geometric framework for unsupervised anomaly detection," in *Applications of Data Mining in Computer Security*. US: Springer,2002, pp. 77–101).

[25] Y. Zheng, H. Zhang, and Y. Yu, "Detecting collective anomalies from multiple spatio-temporal datasets across different domains," in *Proc. ACM 23rd SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, Nov. 2015, Art. no. 2.

[26] Esper Team and EsperTech Inc., "Esper reference version 8.1.0", 2019. [Online]. Available: http://esper.espertech.com/release-8.1.0/reference-esper/pdf/esper_reference.pdf, (last accessed 3/6/2019).

[27] T. Erl, *SOA: Principles of Service Design*. vol. 1. Upper Saddle River, NJ, USA: Prentice Hall, 2008.

[28] R. Laxhammar and G. Falkman, "Online learning and sequential anomaly detection in trajectories," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 6, pp. 1158–1173, Jun. 2014.

[29] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B. H. Park, "Dynamic meta-learning for failure prediction in large-scale systems: A case study," in *Proc. 37th Int. Conf. Parallel Process.*, Sep. 2008, pp. 157–164.

[30] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. Security Privacy*, 2001, pp. 144–155.

[31] F. Salfner and M. Malek,"Using hidden semi-Markov models for effective online failure prediction," in *Proc. 26th IEEE Int. Symp. Rel. Distrib. Syst.*, 2007, pp. 161–174).

[32] MADneSs Scores Data (archive file from GitHub), https://github.com/tommyippoz/Miscellaneous-Files/blob/master/MADneSs_TDSC_Archive_2018.rar (last accessed on 3rd Jan. 2018).

[33] L. Cherkasova, et al., "Anomaly application change or workload change? Towards automated detection of application performance anomaly and change," in *Proc. Depend. Syst. Netw.*, 2008, pp. 452–461.

[34] G. Khanna, P. Varadharajan, and S. Bagchi, "Automated online monitoring of distributed applications through external monitors," *IEEE Trans. Depend. Secure Comput.*, vol. 3, no. 2, pp. 115–129, Apr.-Jun. 2006.

[35] S. Dekker, *Drift Into Failure: From Hunting Broken Components to Understanding Complex Systems*. Boca Raton, FL, USA: CRC Press, 2016.

[36] MADneSs Workload and Database (archive file from GitHub), https://github.com/tommyippoz/Miscellaneous-Files/blob/master/TDSC_MADneSs_DB_Archive.rar (last accessed 3rd Jan. 2018).

[37] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proc. IEEE 9th Symp. Rel. Distrib. Syst.*, 1990. pp. 114–123.

[38] D. M. Blough and G. F. Sullivan, "A comparison of voting strategies for fault-tolerant distributed systems," in *Proc. 9th Symp. Rel. Distrib. Syst.*, Oct. 1990, pp. 136–145.

[39] I. Irrera, J. Duraes, M. Vieira, and H. Madeira, "Towards identifying the best variables for failure prediction using injection of realistic software faults," in *Proc. IEEE 16th Pacific Rim Int. Symp. Depend. Comput.*, 2010, pp. 3–10.

[40] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang, "Fault detection and localization in distributed systems using invariant relationships," in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2013, pp. 1–8.

[41] A. Ceccarelli, M. Vieira, and A. Bondavalli, "A testing service for lifelong validation of dynamic SOA," in *Proc. IEEE 13th Int. Symp. High-Assurance Syst. Eng.*, 2011, pp. 1–8.

[42] L. Baresi and S. Guinea, "Towards dynamic monitoring of WS-BPEL processes," in *Proc. Int. Conf. Service-Oriented Comput.*, Dec. 2005, pp. 269–282.

[43] H. F. Li and S. Y. Lee, "Mining frequent itemsets over data streams using efficient window sliding techniques," *Expert Syst. Appl.*, vol. 36, no. 2, pp. 1466–1477, 2009.

[44] A. Bondavalli, A. Ceccarelli, F. Brancati, D. Santoro, and M. Vadursi, "Differential analysis of operating system indicators for anomaly detection in dependable systems: An experimental study," *Meas.*, vol. 80, pp. 229–240, 2016.

[45] J. T. Giffin, S. Jha, and B. P. Miller. "Efficient context-sensitive intrusion detection," NDSS, 2004.

[46] C. Zhong, T. Lin, P. Liu, J. Yen, and K. Chen, "A cyber security data triage operation retrieval system," *Comput. Security*, vol. 76, pp. 12–31, 2018.

[47] A. AlEroud and G. Karabatis, "A contextual anomaly detection approach to discover zero-day attacks," in *Proc. Int. Conf. Cyber Security*, Dec. 2012, pp. 40–45.

[48] I. Bluemke and M. Tarka, "Detection of anomalies in a SOA system by learning algorithms," *Complex Systems and Dependability*. Berlin, Germany: Springer, 2013, pp. 69–85.

[49] Y. Bar-Yam, “General features of complex systems,” *Encyclopedia of Life Support Systems (EOLSS)*, Oxford, UK: UNESCO, EOLSS Publishers, 2002, p. 1.
[50] A. Benlian, M. Koufaris, and T. Hess. “Service quality in software-as-a-service: Developing the SaaS-Qual measure and examining its role in usage continuance,” *J. Manage. Inf. Syst.*, vol. 28, no. 3, pp. 85–126, 2011.
[51] N. Baliyan, and S. Kumar, “Quality assessment of software as a service on cloud using fuzzy logic,” in *Proc. IEEE Int. Conf. Cloud Comput. Emerging Markets*, 2013, pp. 1–6.
[52] ownCloud, https://owncloud.org/ (last accessed on 12th Dec 2018)

Tommaso Zoppi received the PhD degree in computer science from the University of Firenze, Italy, in March 2018, where he is currently a post-doc researcher. His research focuses on error detection and failure prediction through anomaly detection. Often, his research crosses the domains of security, software engineering and machine learning as well.

Andrea Ceccarelli is a research associate with the University of Florence. He has been involved in several European and National funded projects, and he served in the Program Committee of several International Conferences. His scientific activities originated more than 80 papers appeared in International Conferences, Workshops and Journals.

Andrea Bondavalli is a full professor of Computer Science with the University of Firenze. His scientific activities originated more than 220 papers appeared in international Journals and Conferences. He led various national and European projects and has been chairing the program committee in several International Conferences. He is a member of the IEEE, the IFIP W.G. 10.4 Working Group on “Dependable Computing and Fault-Tolerance”.