IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 February 2024, accepted 7 March 2024, date of publication 11 March 2024, date of current version 15 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3376239

RESEARCH ARTICLE

MOTEF: A Testing Framework for Runtime Monitoring Infrastructures

ANTONELLO CALABRÒ AND EDA MARCHETTI
ISTI-CNR, 56124 Pisa, Italy

Corresponding authors: Antonello Calabrò (antonello.calabro@isti.cnr.it) and Eda Marchetti (eda.marchetti@isti.cnr.it)

This work was supported in part by the Project RESearch and innovation on future Telecommunications systems and networks, to make Italy more smART (RESTART) under Grant PE00000001, and in part by the Project SEcurity and RIghts in the CyberSpace (SERICS) under the National Recovery and Resilience Plan (NRRP) Ministero dell'Università e della Ricerca (MUR) Program funded by the EU–NextGenerationEU under Grant PE00000014.

ABSTRACT Intelligent monitoring systems can effectively predict or detect anomalies and issues in smart working systems and ecosystems and implement the proper countermeasures. However, for their effective and efficient use, attributes like responsiveness, performance, and quality should be properly tested and assessed before integrating the monitoring system into an ecosystem. The work aims to present a framework, called *MOnitoring TEsting Framework* (MOTEF), focused on the testing and evaluation of a generic monitoring system performance. In particular, the framework allows testing the monitoring system in isolation or when used in a smart environment to provide functional or non-functional property predictions. By simulating the runtime execution of a smart environment, MOTEF lets testing and assessment of a generic monitoring system establish its working boundaries. The results collected can be used to design a smart environment architecture to fulfil its global performance constraints better. This work presents the architecture of MOTEF and its preliminary implementation. It also validated and showcased the use of MOTEF in evaluating the performance of an existing monitoring system in isolation and when it is used in a smart environment. The results have been assessed by considering two research questions about the monitoring system's responsiveness and effectiveness in proving required functional or non-functional property predictions.

INDEX TERMS Monitoring, testbed, empirical evaluation, testing.

I. INTRODUCTION

In recent years, smart environments have become daily software engineering and development process solutions for maximizing productivity, improving cooperation, and timely targeting operational and business goals. Often, in such dynamic contexts, to avoid uncontrolled situations, guarantee the established quality attributes, and promptly recognize or predict possible functional or functional properties deviation, a common choice is to integrate into the smart environment an intelligent monitoring system [1], [2], [3], [4].

They can collect events or evidence from heterogeneous sources (such as applications, components, sensors, or devices) and infer complex patterns to assess observed behaviors. Additionally, intelligent monitoring can promptly predict or detect anomalies and issues and implement the proper countermeasures by working in strict synergy with the operational environment's tools and components. However, the efficiency of the smart environment requires that its systems, components, and processes are effective and efficient in terms of responsiveness, performance, and quality. In particular, the monitoring system should not be a bottleneck of the smart environment in which it operates. Therefore, attributes like responsiveness, performance, and quality must be tested and assessed to select the most appropriate monitor solution before integrating it into the smart environment. Focusing particularly on monitoring responsiveness, this paper describes a framework called MOTEF (MOnitoring TEsting Framework), which is focused on testing and evaluating a generic monitoring system performance. In particular, the framework allows testing a monitoring system in isolation or when used in a smart environment to provide functional or non-functional property predictions. Considering that the overall cost of testing is

The associate editor coordinating the review of this manuscript and approving it for publication was Luca Cassano.

around 40% of the total development costs of a typical software project [5], [6], using automatic frameworks to reduce the verification, validation, and assessment impact is a stringent need, especially in the industrial environment. MOTEF replies to this exigence by providing a practical means for automatically assessing the performance and peculiarities of monitoring systems.

In particular, MOTEF was conceived to simulate the runtime behavior of a smart environment by integrating a monitoring system. In its realization, MOTEF takes as a reference for the smart environment description the Smart Monitoring Framework (SMF) architecture presented in [7]. Therefore, MOTEF includes components for the simulation of the SMF main facilities (i.e. a *GUI*, a *Predictive Simulation*, an *Ontology Manager*, and a *Communication Channel*) and specialized one helpful in executing the testing and assessment of a generic monitoring system instance (i.e. a *TestBed Orchestrator* and an *Events generator*.)

The paper's contribution can be summarized as in the following:

- defining the reference architecture of MOTEF;
- providing the MOTEF preliminary implementation;
- validation and showcase the MOTEF usage with an existing runtime monitoring implementation.
- discuss the obtained results and future work.

When used in testing and assessing a generic instantiation of a monitoring system, the preliminary implementation of the MOnitoring TEsting Framework (MOTEF) provided in this paper can be used for answering the following Research Questions (RQs):

RQ1 (Standalone Performance): What are the performances of the runtime monitoring in terms of responsiveness? The aim is to investigate whether the runtime monitoring is suitable for promptly providing *functional and non-functional properties evaluation*.

RQ2 (Effective Prediction): Can the runtime monitoring ensure effective, timely prediction of functional and non-functional properties? The aim is to establish the minimum time interval required for providing smart predictions so that systems like Digital Twin (TD), Artificial Intelligence (AI), or Machine Learning (ML) can use them in their operational execution.

For research questions data collection, MOTEF implementation includes two experiment settings:

1) Using MOTEF to evaluate the performance of runtime monitoring in isolation and
2) Using MOTEF to evaluate runtime monitoring for effective prediction, i.e., calculating the execution boundaries of monitoring in case of prediction forecasting.

Outline: The remainder of this article is organized as follows. The main background and related works are presented in Section II, while the description of the MOnitoring TEsting Framework (MOTEF) Reference Architecture and its reference behavior are described in Section III. The preliminary instantiation of MOTEF is presented in Section IV and showcased by using an existing monitoring system in Section V. The experimental results are evaluated and discussed in Section VI. Finally, in Section VIII, conclusions and possible future work are highlighted.

II. BACKGROUND AND RELATED WORK

This section overviews the main background and related work regarding monitoring, predictive simulation, and testing frameworks.

A. ON THE MONITORING

Runtime monitoring systems have become increasingly important in various applications, from industrial automation to healthcare to smart cities. These systems use sensors and other data sources to monitor the environment and assess or predict behavioral patterns and properties using algorithms and analytics. Usually, runtime monitoring is grounded on an event-based system that analyzes events sent by the target environment's component, sensors, or subsystem when the status changes or specific operations occur. This results in more efficient use of resources and lower performance and cost impacts. Indeed, an event-driven approach is characterized by loosely coupled components communicating through events rather than a traditional request-response model and provides greater flexibility and scalability since each component can operate independently and asynchronously. In an event-driven system, components only react when they receive an event that matches their specific interest rather than continuously polling the system. The monitoring activity can improve the quality of service (QoS), prevent or reduce violations of properties, and provide recovery mechanisms in case of detected problems [8], [9], especially in dynamic environments where unpredictable events may occur.

Usually, the runtime monitor includes a rule-based complex event processor [10], [11], [12], [13]. The rules evolve automatically based on the data (events) collected during the execution according to the concept of observability [14], [15]. In this case, rule evolution is based on meta-rules [16], enriched and enacted through events generated and aggregated at runtime.

Researchers have explored runtime monitoring in various domains, such as industrial automation, healthcare monitoring, and smart cities. Available proposals include: monitoring based on logical or axiomatic approaches [17], [18], [19]: monitoring focused on instrumentation algorithm for asynchronous components [20], [21]. Cited works highlighted the advantages of the event-driven approach due to its flexibility, scalability, and ability to handle large volumes of data in real time. However, there are also some limitations to the event-driven approach, such as the need for careful event management and the potential for latency in some cases.

Due to its peculiarity, before integrating a runtime monitoring system into a smart environment, it is necessary to test and evaluate the boundaries of its performances, especially the responsiveness and time constraints. These values can be crucial for deciding how and under which

conditions the monitor can adequately work, and avoiding it becomes a blind spot of the overall environment execution. The testing results could also be used for deciding to deploy, lock, or purchase additional resources to improve monitoring activities [22].

B. ON THE PREDICTIVE SIMULATION

Predictive Simulation aims at detecting malicious deviations using techniques like Machine Learning (ML), Artificial Intelligence (AI) [23], or Digital Twin (DT), developed specifically for reaching this goal. Considering, in particular, the usage of DT, it enables a concern-directed prediction of the trustworthiness of intelligent software behavior [24], [25]. Different models of the system behavior are created to enable this approach, including functional models that enable runtime evaluation of functional interaction, temporal models that enable timing predictions used in evaluating a software smart agent's synchronization capability, and models that will allow the runtime evaluation of the communication protocol. From the usage of all these models, according to the temporal logic model used to validate the accuracy of DTs in terms of timing abstraction, monitoring of the conformity of what is going to be executed and what is supposed to be executed (DT) [26] can be made.

C. ON THE TESTING FRAMEWORK

Testing is one of the primary approaches for assessing and validating the quality attributes of different systems. Considering, in particular, the criticalities and quality levels required for IoT systems and smart environments, the possibility of automatically executing testing activity and collecting evidence about the peculiarities and performances of their components before integration into a unique environment is becoming a stringent need. In this context, this section overviews the testing proposals for assessing the monitoring system's quality aspects and possible solutions for automatically testing and simulating its behavior.

Among related proposals, that provided by [27] and [28] describes a framework for validation of the robotic applications developed using the Robot Operating System (ROS) while [29] provides testing approaches for checking the consistency and the correctness runtime systems to ensure their trustworthiness.

However, considering the runtime monitoring peculiarities, most currently available solutions are not explicitly conceived. Contrarily the available literature, this paper proposes a specific testing framework for the runtime monitoring infrastructure that can evaluate the performance of smart monitoring systems based on the event-driven approach using a real testbed from a smart building scenario.

The following sections will provide more specific details about MOTEF architecture and instantiation. The evaluation of a specific event-driven Runtime Monitoring will be presented, and the relevant results will be discussed.

III. MOTEF REFERENCE ARCHITECTURE

This section describes MOTEF reference architecture, which emulates the component needed for executing a Runtime Monitoring in a Smart Monitoring Framework (SMF) that has already been described in [7]. Section III-A provides basic details about SMF, while Section III-B details MOTEF architecture instantiation.

A. SMART MONITORING FRAMEWORK

By referring to [7] for more details, the features of SMF are:

1) providing mechanisms for specifying functional and non-functional properties;
2) integrating predictive mechanisms of leveraging the monitoring of preventive activity;
3) collecting and managing the data generated by different and heterogeneous sources (HW/SW) and at different levels (application stack or system layers);
4) assessing functional or non-functional properties;
5) rising alarms or enacting countermeasures in case of property violation;
6) providing user interaction mechanism.

Figure 1, describes, according to the standard UML component diagram representation, the components of the SMF. In Figure 1, the component named *Environment (Sensors and gateways* of Figure 1 represents the external group of sensors and gateway data that the SMF can use during its execution. Details about the SMF components are provided in the following:

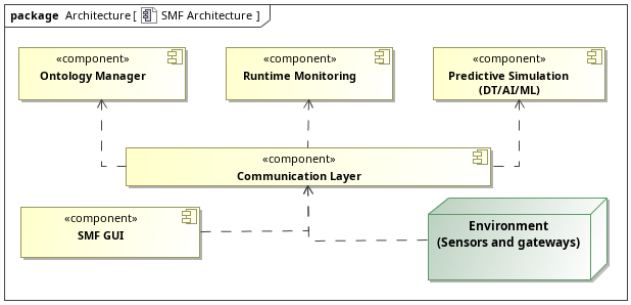

FIGURE 1. The SMF Architecture and the Environment of Sensors.

Communication Layer: That manages all the communications between the components and lets data be exchanged through events.

Ontology Manager: That manages the knowledge by representing monitoring data and functional or non-functional properties by ontologies [30], [31].

Predictive Simulation: That predicts the behavior of target monitored components or environment using one of the solutions described in Section II-B.

Runtime Monitoring: This component collects data (typically events) from the target monitored system or environment and evaluates functional and non-functional properties. It raises alarms or directly enacts countermeasures in case of violation or misbehavior. *Runtime Monitoring* interacts

with the *Ontology Manager*, receiving rules related to the functional and non-functional properties to be analyzed translated according to the *Runtime Monitoring* language [32]. At the same time, it also receives the predictions of expected behavior to check against the real execution of the system from the *Predictive Simulation*. All that information flows across the *Communication Layer*.

SMF GUI: it manages the user interaction by providing features such as *Stop, Resume,* and *Save* activities and data.

B. MOTEF TESTBED

In Figure 2, the testbed architecture of MOTEF is depicted. Some components have been replaced with emulated ones to avoid bias unrelated to the tested component: the Runtime Monitoring. In the following, more details about emulated components instantiated are provided:

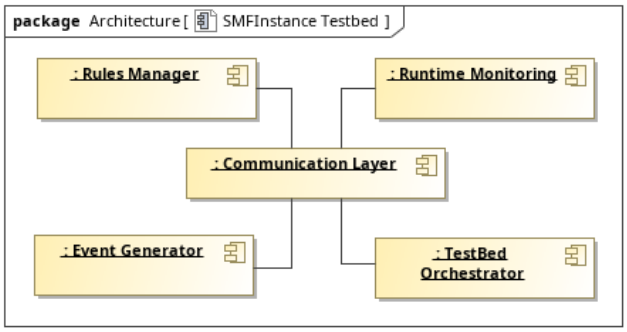

FIGURE 2. Testbed architecture of the MOTEF proposal.

1) TESTBED ORCHESTRATOR

This component manages the runtime execution of the overall testing process and lets the execution of the following testing scenarios: i) testing using only *Simple Rules*, i.e., rules defined using basic logic operators; ii) testing using *Complex Rules*, i.e., rules derived by the composition of *Simple Rules*; iii) testing using sets of actual rules and associated events taken from a real environment execution; iv) testing using smart predictions. The *Testbed Orchestrator* is also responsible for setting up the other MOnitoring TEsting Framework (MOTEF) components according to the selected scenario and providing them the required input.

2) RULES MANAGER

This component emulates the Ontology Manager behavior. In particular, according to the indications and input received by the *Testbed Orchestrator*, the *Rules Manager* selects and sends to the *Runtime Monitoring* the set of rules to be monitored. This set can be composed of *Simple Rules*, *Complex Rules*, or *Predictive Rules*. In this last case, these rules enable the *Runtime Monitoring* to generate the skeleton of rules used for smart predictions during the testing execution.

3) EVENT GENERATOR

This component emulates the Node Environment, see Figure: 2 on which sensors and data gateway are in execution generating events to be analyzed. According to the indications and input received by the *Testbed Orchestrator*, the *Event Generator* fires one by one the events to the *Runtime Monitoring* component for triggering rules according to specific behavioral policies. In particular, *Event Generator* manages two kinds of events: *Normal Events* for enacting a *Simple Rule* or *Complex Rule* and *Predictive Events*. In the latter case, the events are used to simulate the behavior of the *Predictive Simulation* component (see Figure 1) and let the *Runtime Monitoring* enact the prediction mechanism.

C. MOTEF INTERACTION PATTERN

This section describes, through the UML sequence diagram in Figure 3, a simplified interaction pattern of the MOTEF Testbed execution. The interaction among MOTEF components is related to the case in which a set of predictive rules is loaded, and the self-generation procedure is triggered.

The *TestBed Orchestrator* is the core component of the testing process. It is responsible for starting and stopping the activities, providing input to the different MOTEF components according to the test scenario to be executed, and managing the testing session. The *TestBed Orchestrator* interacts with other components through messages sent via a *Communication Layer*. To improve diagram clarity, the *Communication Layer* has been omitted in Figure 3.

In Figure 3 message `1. Start` sent by the *TestBed Orchestrator* to the *Runtime Monitoring* enact the testing session. This message pushes the *Runtime Monitoring* to initialize itself (message `2. Setup` of Figure 3) and to create the channel for receiving rules and events. When setup is complete, the *Runtime Monitoring* notifies the *TestBed Orchestrator* (`3. Running` of Figure 3), letting the testing set-up finalization.

According to the testing scenario to be executed, the *TestBed Orchestrator* indicates the target set of testing rules to the *Rules Manager* (message `4. Load predictive meta-rules` of Figure 3), forcing the preparation of the information to be sent to the *Runtime Monitoring* (messages `5. Meta-rules packaging()` and `6. Sending rules` of Figure 3, respectively).

Once received, the *Runtime Monitoring* loads the received meta-rules set into its knowledge base (message `7. Loading meta-rules`) and sends a notification message (`8. Monitoring ready` message) to the *TestBed Orchestrator*.

The *TestBed Orchestrator* loads now the events related to the predictive meta-rules set already loaded into the *Event Generator* (message `9. Loading events for predictive meta-rules`).

This message will complete the Predictive execution setup, and the simulation will start through the message `10. Start simulation`.

For demonstrating the scenario, the *Event Generator* sends now events containing forecast (message `11. Event (forecast)`) that trigger, on the Runtime Monitoring side, the rule generation procedure using the meta-rules already loaded (message `12. Rules generation procedure and self-injection`).

The successive phase is represented in a **loop** on which a set of activities are executed simultaneously, while there are no more events to be processed: `EventStream not empty` condition.

More in detail, in the **loop** fragment, three activities are executed:

- the sending of events by the *Event Generator* to the *Runtime Monitoring* (message `13. Event`);
- the assessment executed by the *Runtime Monitoring* (message `14. Rules and meta-rules assessment`) to check if the event received at message 13 is triggering a rule. As specified in the tooltip, in this case, the event will trigger the rule generated at message 12;
- the notification message related to triggering a rule (message `15. Rules triggering notification`).

The results of the execution: timing, rule matching values, and rules generated are stored in the *TestBed Orchestrator* for further offline analysis.

IV. MOTEF INSTANTIATION

This section describes the instantiation of the MOTEF architecture. In this version *Rules Manager* and the *Predictive Simulation* are emulated by dedicated components developed to reduce the possible bias of the collected results and focus on the *Runtime Monitoring component*.

Considering the *Rules Manager*, this artifact emulates the Ontology Manager behavior and its execution includes two types of rules:

1) *F&NF Rules:* representing functional and non-functional properties. A *F&NF Rule* can be a *Simple Rule* or a *Complex Rule* and is associated with a predefined set of events to be launched during the test execution by the *Event Generator*.
2) *Predictive Rules:* representing property predictions. As shown in Listing 7, these are realized through meta-rules that trigger the self-generation and self-injection procedure of the *Runtime Monitoring* component. Thus, each *Predictive Rule* lets *Runtime Monitoring* autonomously derive the specific *F&NF Rule* and self-inject it. Each *Predictive Rule* is associated with a set of specific events, fired by the *Event Generator*, each one containing or a forecast event, that will enact the rules' self-generation mechanism or an event related to a prediction already sent.

The behavior of the **Predictive Simulation** is emulated by the *Event Generator* component included in the MOTEF architecture. Through the *Communication Layer*, the *Testbed Orchestrator* provides the sets of events that the *Event Generator* sends to the *Runtime Monitoring* during the test execution. Multiple instances of the *Event Generator* can be executed to simulate parallel sending.

In the following subsections, the procedures used for the realization of the stubs mentioned above are provided. In particular, Section IV-A and Section IV-B describe the generation of the *F&NF* and *Predictive* rules sets, respectively.

A. GENERATION PROCEDURE OF F&NF RULES SET

The generation procedure of the *F&NF* rules and their relative events relies on the following rule structures:

1) `Simple Rule`: represents the basic rule and follows the schema provided in Listing 3. It is grounded on the basic logic operators (like $\prec$, $=$, $\succ$, $\neg$, &, and $\|$) and as shown in the Listing 3 includes three main fields:
   - `header`, containing the parameters definition and initialization (*Attributes* field in Listing: 3);
   - `logic`, that specifies the condition to be checked (*Conditions* field in Listing 3);
   - `tail` that contains the action/countermeasure to be executed (*Actions* field in Listing 3);

   In Listing 4, an example of the *Simple Rule* named *check value threshold* is reported. In the rule, three attributes are considered `(no-loop, salience 20, and dialect)`, and the logic checks if the value stored in the data field of an event with the `EventName` parameter equals to `EVENTNAME_A` is greater than an established threshold. The message `Failure detected` is returned as output in this case.
2) `Complex Rule`: a rule structured as a composition of more *Simple Rules*. The *Complex Rule* results from a join of the `logic` part of a set of *Simple Rule* using the & operator. As shown in Listing 5 the composition is derived as a sequence of logic fields. In this case, the logic part of the rule named `check values` checks if the data field of an event with the `EventName` parameter equals to `EVENTNAME_A` is greater than an established threshold and if the data field of an event with the `EventName` parameter equals to `EVENTNAME_B` is equal to a specific value. If both conditions are satisfied, the message `Failure detected` is returned as output.

The *F&NF Rules* sets have been defined by i) Using two generation procedures able to randomly generate sets *Simple Rules* or a *Complex Rules*; ii) selecting and anonymizing *Simple Rules*, *Complex Rules* (and relative events) from real data collected during a monitoring activity inside a Smart Environment. Two sets of 300 and 900 rules and associated events have been collected in this last case. These sets aim to let the *Runtime Monitoring* evaluation with massive execution of rules and events, simulating the case of a high-density Smart Environment [11].

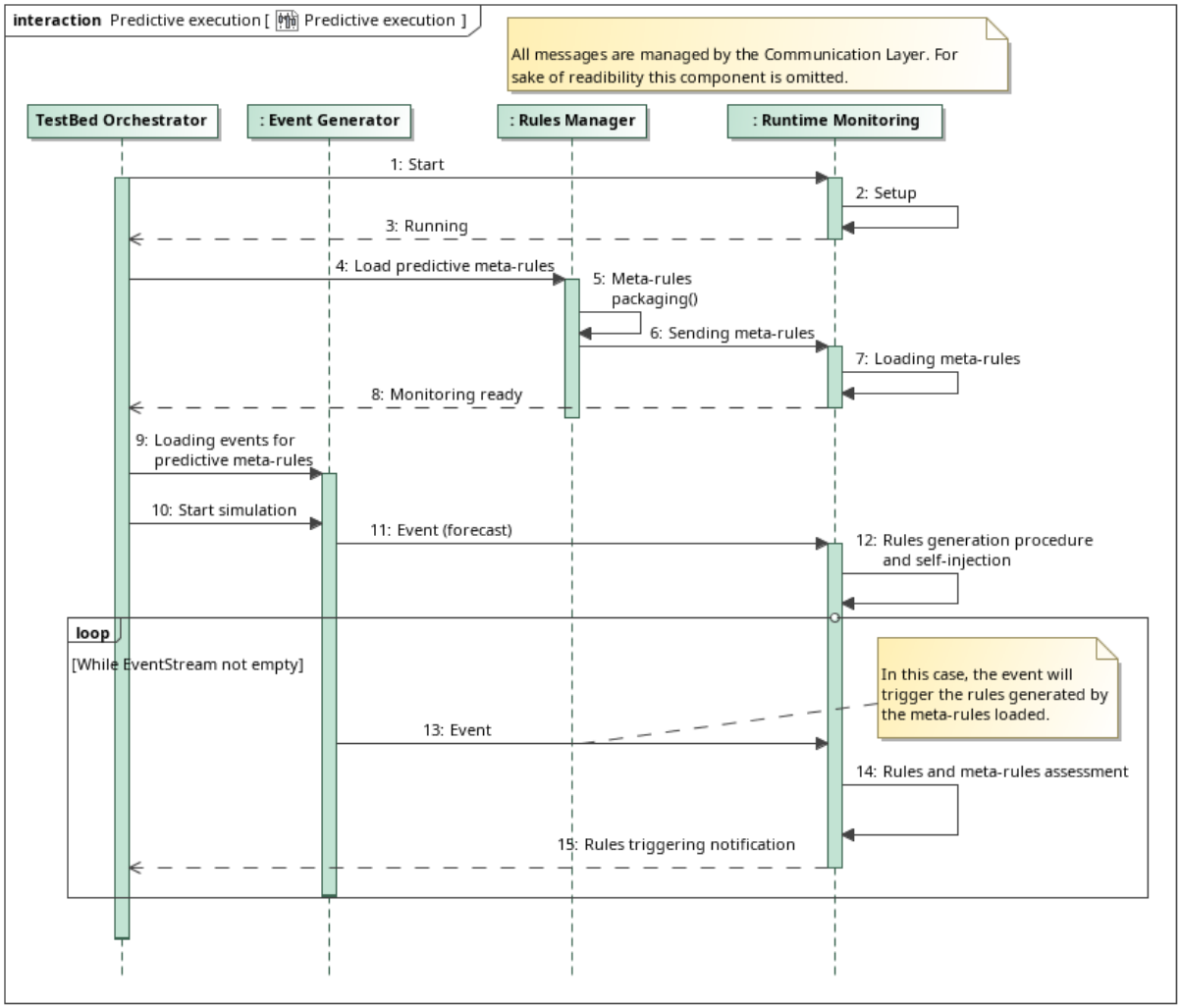

FIGURE 3. Sequence diagram for MOTEF instance execution.

Below are the random procedures for generating the rule sets for the *Rules Manager* component and the relative events for the *Event Generator*.

Generation of sets of Simple Rules: 30 sets of *Simple Rules* have been generated considering the following procedure:

1) Manually generate a set of 50 *Simple Rule*, and the set of events needed to let the rule be fired.
2) Randomly extract *a* number from 1 to 20 (let *a* be the number extracted);
3) Randomly extracts *a* rules from the set of 50 *Simple rule* and their relative events.

The pseudo-code for generating the *Simple Rules* set is described below:

Generation of the sets of Complex Rules: 30 sets of *Complex Rules* have been generated considering the following procedure:

1) Manually generate a set of 50 *Simple Rule*, and the set of events needed to let the rule be fired. For manually generate *Simple Rules* a basic rule template described in Drools Documentation[1] has been taken as reference. Each *Single Rule* contains the check of 3 parameters of a generic event: a check on a boolean value, a string

```
# Define an array of 50 SimpleRules elements
simple_rules = ["Rule 1", "Rule 2", "Rule 3",
    [...], "Rule 49", "Rule 50",]

# Generate 30 sets of rules
num_sets = 30
rule_sets = []

for _ in range(num_sets):
    num_rules_in_set = random.randint(1, 20)
    random_rules = random.sample(simple_rules,
        num_rules_in_set)
    rule_sets.append(random_rules)

# Print the generated rule sets
for i, rule_set in enumerate(rule_sets, 1):
    print(f"Set {i}: {', '.join(rule_set)}")
```

LISTING 1. Simple rule sets generation.

[1]https://docs.drools.org/7.74.1.Final/drools-docs/html$_s$ingle/index.html$_d$roolslanguagereferencechapter

comparison, and a data value comparison using < or > operator.

2) Generate a set of 100 *Complex Rule*, each one containing a composition from 1 to 5 of the *Simple Rule*, according to the following steps:
   - Randomly extract *a* number from 1 to 5 (let *a* be the number extracted);
   - Randomly extracts *a* rules from the set of 50 *Simple rule* and their relative events.
   - Compose a *Complex Rule* using the & logic, package the events into a sequence, and connect the sequence to the generated *Complex Rule*.
3) Generate 30 sets of rules and their relative events, each one having a variable number of `Complex Rules` (from 1 to 20) according to the following steps:
   - randomly extract *a* number from 1 to 20 (let *s* be the number extracted);
   - randomly extract *s* rules from the set of 100 `Complex Rules` and their event and include them in the set.

The pseudo-code for generating *Complex Rules* sets is described below:

```
# Define an array of 50 SimpleRules elements
simple_rules = ["Rule 1",  "Rule 2",  "Rule 3",
    [...],  "Rule 49", "Rule 50",]

# Define a function to generate a ComplexRule
def generate_complex_rule():
    num_simple_rules = random.randint(1, 5)
    return random.sample(simple_rules,
        num_simple_rules)

# Generate 30 sets of ComplexRules
num_sets = 30
complex_rule_sets = []

for _ in range(num_sets):
    num_complex_rules_in_set = random.randint(1,
        20)
    complex_rules_set = [generate_complex_rule()
        for _ in range(num_complex_rules_in_set)]
    complex_rule_sets.append(complex_rules_set)

# Print the generated ComplexRule sets
for i, rule_set in enumerate(complex_rule_sets, 1):
    print(f"Set {i}:")
    for j, complex_rule in enumerate(rule_set, 1):
        print(f"  ComplexRule {j}: {',
            '.join(complex_rule)}")
```

LISTING 2. Complex rule sets generation

These sets of *Simple Rules* and *Complex Rules* and their relative events will be used by the *TestBed Orchestrator* for setting up and enacting the experiment during the execution of MOnitoring TEsting Framework (MOTEF).

```
declare   // Optional
rule "rule name"
    // Attributes
    when
        // Conditions
    then
        // Actions
```

LISTING 3. Simple rule template

```
rule "check value threshold"
    no-loop
    salience 20
    dialect "java"
when
    $aEvent : BaseEvent(
                    this.isConsumed == false,
                    this.getEventName == "EVENTNAME_A",
                    this.getData > #THRESHOLD# );
then
    System.out.println("Failure detected");
end
```

LISTING 4. Simple rule example

```
rule "check values"
    no-loop
    salience 20
    dialect "java"
when
    $aEvent : BaseEvent(
                    this.isConsumed == false,
                    this.getEventName == "EVENTNAME_A",
                    this.getData > #THRESHOLD# );

    $bEvent : BaseEvent(
                    this.isConsumed == false,
                    this.getEventName == "EVENTNAME_B",
                    this.getData = #EQUALVALUE# );
then
    System.out.println("Failure detected");
end
```

LISTING 5. Complex Rule example

B. GENERATION PROCEDURE OF PREDICTIVE RULE SET

Predictive rules are rules generated at execution time by the infrastructure in charge to monitor the system after receiving an event that contains a forecast.

A prediction must contain information related to the event in the system and the possible value that the event may take. Once a prediction is made and sent to the monitoring infrastructure or a dedicated component in charge to receive and generate forecasting, it should be able to interpret the forecast and provide a rule that will be triggered if the forecasted event in specific conditions is notified to it. The forecast may contain a single event, a composition, or a sequence. Information related to the threshold to be respected or other properties must be provided within the forecast. This will allow the component in charge to generate the rule and select the correct meta-rule template. Once the correct template is selected and instantiated with the requested parameters, the output artifact (a new rule) will be self-injected into the monitoring infrastructure.

The generation procedure adopted for the *Predictive Rules* is executed by the *Test Orchestrator*.

It loads 50 randomly generated *Simple Rule* and generates 15 sets, each containing 10 *Predictive Rules*. The *Predictive Rules* are generated by selecting randomly a *Simple Rules* previously loaded. The generated 15 sets are provided to the *Predictive Simulation* to be sent randomly to the *Runtime Monitoring* during the execution.

The pseudo-code for generating the *Predictive Rules* sets is described in Listing: 6.

V. EXPERIMENTAL EVALUATION

An existing Runtime Monitoring has been tested to showcase the use of MOnitoring TEsting Framework (MOTEF) proposal. In particular, from the available Runtime Monitoring engine, the Complex Event Processing Monitor Infrastructure

```python
# Define an array of 50 randomly generated SimpleRules
simple_rules = [f"SimpleRule {i}" for i in range(1, 51)]

# Define a function to generate a PredictiveRule
def generate_predictive_rule():
    simple_rule = random.choice(simple_rules)
    return f"PredictiveRule based on {simple_rule}"

# Generate 15 sets of PredictiveRules
num_sets = 15
predictive_rule_sets = []

for _ in range(num_sets):
    predictive_rules_set = [generate_predictive_rule() for _ in range(10)]
    predictive_rule_sets.append(predictive_rules_set)

# Print the generated PredictiveRule sets
for i, rule_set in enumerate(predictive_rule_sets, 1):
    print(f"Set {i}:")
    for j, predictive_rule in enumerate(rule_set, 1):
        print(f"  PredictiveRule {j}: {predictive_rule}")
```

LISTING 6. Predictive rules sets generation.

CONCERN[2] was selected because it is an open-source, customizable, and generic monitoring proposal. Additionally, its effectiveness and efficiency in monitoring functional and non-functional properties have already been evaluated as appropriate in several specific contexts and application domains (such as [11], [13], [33], [34]). This evidence guarantees that CONCERN could be a good candidate for the MOTEF experimentation. Additionally, to allow loosely coupled communication and to manage vast amounts of data, the *Communication Layer* implementation was based on JMS Messages[3] on top of ActiveMQ Message Broker.[4]

For experimental purposes, the *Testbed Orchestrator* has been set up to collect results helpful in answering the research questions presented in the introduction. In particular, considering the first question, i.e., **RQ1: What are the performances of runtime monitoring in terms of responsiveness?**, the focus has been devoted to evaluating the time required to load different types of rules (simple and complex). In particular, the evaluation includes the following capabilities:

1) the capacity to manage a large number of rules;
2) the impact of loading simple vs. complex monitoring rules;
3) the capability firing the rule;
4) the capability of notification of a rule violation.

Thus, specific metrics for **RQ1** have been defined as the following:

1) **Rules Loading Time**: The metric is calculated according to the schema reported in Figure 4. In particular, the interval time ($\Delta t_{loading}$) between receiving rules time (t_0) and rules loaded time (t_1) is calculated as

$$\Delta t_{loading} = t_1 - t_0 \tag{1}$$

2) **Rules Firing Time**: The metric is calculated as the difference between the time of receiving an event from the monitored systems (in our case, from the *Rule Manager)* and the time of notification of a possible violation, as illustrated below.

$$\Delta t_{firing} = t_3 - t_2 \tag{2}$$

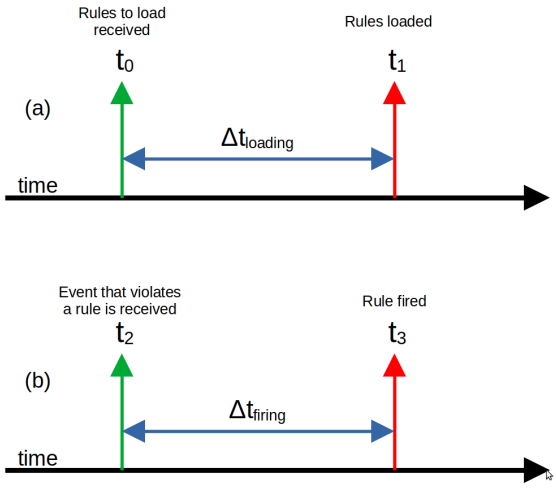

FIGURE 4. Metrics for RQ1: (a) Rules Loading Metric and (b) Rules Firing Metric.

For **RQ2**, the metric considered is:

1) **Effective Prediction:** The average time requested by the *Runtime Monitoring*, after receiving a forecast from the *Predictive Simulation* component to generate a rule capable of capturing the forecasted event and self-inject it into its own complex event processor knowledge base.

The calculation of this capability is done by fractioning the time interval in three phases:

- (Δt_A) time requested for launching and executing the rule generation process when the event that triggers the generation is received;
- (Δt_B) time requested for self-inject the generated rule in the knowledge base;
- (Δt_C) time requested for notifying a violation of the generated and self-injected rule.

The sum of $\Delta t_A + \Delta t_B + \Delta t_C$ is equal to the time boundary that is required to elapse between the sending of a forecasted event and the real occurrence of it.

$$\Delta t_{predictionBoundary} = \Delta t_{generate} + \Delta t_{inject} + \Delta t_{firing} \tag{3}$$

Considering that the generated predictive rule is behaving as a generic simple or complex rule, we avoid recalculating for the **RQ2** the value of (Δt_{firing}) because it is equal to the value calculated on **RQ1**.

[2]https://github.com/ISTI-LABSEDC/Concern
[3]https://javaee.github.io/jms-spec/
[4]https://activemq.apache.org/

For this reason, in Figure 5, the graphical timing representation of the **RQ2** is reported.

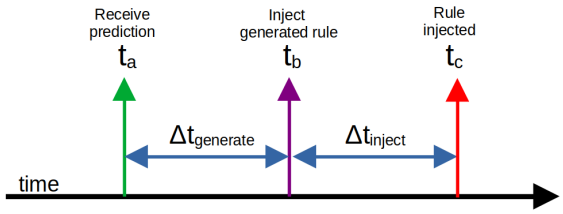

FIGURE 5. Metrics for RQ2: Rule generation and injection from a prediction.

Where $\Delta t_{generate}$ is calculated as follows:

$$\Delta t_{generate} = t_b - t_a \tag{4}$$

whereas Δt_{inject} is computed as in the following equation:

$$\Delta t_{inject} = t_c - t_b \tag{5}$$

In this way, if the forecasted events will occur after a time interval major than the time requested to receive a prediction, generate a new rule, and self-inject it, the monitoring infrastructure used will be useful for prediction.

VI. EXPERIMENTAL RESULTS

This section describes the experimental results collected using the MOnitoring TEsting Framework (MOTEF) for testing the CONCERN responsiveness and effectiveness.

All the testing activities were executed on a machine running Kali Linux OS 2023.03, equipped with an Intel i7-10610U CPU @ 1.80GHz with SSD and 16GB RAM. The MOTEF components have been developed in Java using Eclipse, and the *Runtime Monitoring* test has been executed on top of the Java 19 version.

In particular, the following subsections reply to the research questions **RQ1** and **RQ2** presented in Section I.

A. RQ1: STANDALONE PERFORMANCE

In answering **RQ1** (i.e., *What are the performances of runtime monitoring regarding responsiveness?*), MOTEF has been used for evaluating the CONCERN's capability of managing sets of *Simple Rule* and *Complex Rule*, firing a rule, and triggering a violation considering the following Experiments:

1) **Experiment 1 - Loading Simple Rules:** In this set up, through the *Orchestrator*, the *Rule Manager* receives the sets of *Simple Rules* derived according to the procedure described in Section IV-A and the *Event Generator* receives the relative events.
2) **Experiment 2 - Loading Complex Rules:** In this set up, Through the *Orchestrator*, the *Rule Manager* receives the sets of *Complex Rules* derived according to the procedure described in Section IV-A and the *Event Generator* receives the relative event.
3) **Experiment 3 - Loading real set of 300 rules:** In this set up, through the *Orchestrator*, the *Rule Manager* receives the set of 300 *Simple Rule* and *Complex Rule* selected by real data as described in in Section IV-A and the *Event Generator* receives the relative events.
4) **Experiment 4 - Loading real set of 900 Rules:** In this setup, through the *Orchestrator*, the *Rule Manager* receives the set of 900 *Simple Rule* and *Complex Rule* according to the procedure described in Section IV-A and the *Event Generator* receives the relative event.

In the four Experiments, the MOnitoring TEsting Framework (MOTEF) execution evaluates the time required for the rule loading ($\Delta t_{loading}$) in two different CONCERN setups. A) *Cold* when CONCERN has no rule uploaded in its knowledge base at start-up, B) *Hot* when CONCERN already includes a set of rules loaded.

Experiments 1 and 2 were repeated 20 times, and average values were computed as reported in Table 1.

According to the data collected, in all the 1-4 Experiments, there are important differences between the *Cold* and *Hot* CONCERN setup. In particular, the time required to set up the knowledge base (the component on which all rules are loaded and stored) could have an important impact in the case of a small set of *Simple Rule*. Indeed, in the first Experiment, the *Cold* case takes 507% more time than *Hot*; in the second, it takes around 187%; in the third, it takes 131% more; and in the fourth, it takes 50% more time. These data established the boundary ranges in which monitoring activity can be considered trustable after the monitor's start-up. These can also be used to evaluate if it would be convenient to start and stop the monitoring activity frequently or keep the monitor active, avoiding loading time delay.

Another evaluation of the responsiveness of the CONCERN can be done considering the type of rules to be evaluated. In the case of the *Complex Rule* set on average, the experiment evidenced that CONCERN took more time to load, probably due to the increased number of resources and libraries required for the management. Indeed, comparing data of Experiments 1 and 2 on average, managing 20 *Simple Rules* or 20 *Complex Rules* takes 1.6 times more in the *Cold* case and 3.5 in the *Hot* case. All these aspects can help decide whether it is convenient to deploy multiple instances of runtime monitoring or improve the management of its elasticity.

Finally, Experiments 3 and 4 confirm the intuition that loading time could increase depending on the number of managed rules, either in *Cold* or *Hot* cases, even if without a linear growth. Indeed, comparing Experiments 3 and 4, managing 300 rules or 900 takes 2.5 times more in *Cold* cases and 3.8 in *Hot* cases. This should be considered in all cases where monitor activity should fulfill a strict real-time reaction threshold or manage a huge amount of data and events, such as the case of industrial networks equipped with 5G devices where the monitoring systems need to react according to their high-critical infrastructure time constraints. In this case, the evaluation of raising another monitoring instance instead of

TABLE 1. Experimental execution results.

Experiment n°	1 - Simple		2 - Complex		3 - RS 300		4 - RS 900		5 - Predictive	
CEP State	*Cold*	*Hot*	*Cold*	*Hot*	*Cold*	*Hot*	Cold	Hot	*Cold*	*Hot*
$\Delta t_{loading}$	418ms	69ms	685ms	239ms	1500ms	650ms	3750ms	2500ms	475ms	102ms
$\Delta t_{generate}$	-	-	-	-	-	-	-	-	10ms	
Δt_{inject}	-	-	-	-	-	-	-	-	44ms	
Δt_{firing}	11ms		12ms		103ms		105ms		7ms	

working on the existing one with a higher load of rules could be necessary.

Regarding the average time required for firing a rule after receiving an event that triggers it ((Δt_{firing})), the results show few differences between groups of Experiments 1 and 2 (1 ms on average) and 3 and 4 (2ms on average). Instead, the difference is between the heavy load of Experiments 3 and 4, which require 103 and 105, respectively, and Experiments 1 and 2, which require 103 and 105, respectively. These evaluations are important for establishing the runtime monitor's responsiveness in a time-critical environment. Indeed, analysts and developers could use the collected data to establish a trade-off between the maximum time allowable for receiving a notification of violation (considering the amount and type of rules loaded) and the time required for deploying another instance of Runtime Monitoring.

In conclusion, MOnitoring TEsting Framework (MOTEF) can provide useful data for answering **RQ1** and establishing the runtime monitoring responsiveness time boundary in different situations.

B. RQ2: EFFECTIVE PREDICTION

To answer the second research question presented in the introduction (i.e., *Can runtime monitoring ensure effective Smart Prediction?*), an additional Experiment has been executed to evaluate the CONCERN's ability to collaborate with components to make predictions.

1) EXPERIMENT 5 - PREDICTIVE DATA

With CONCERN in the *Cold* and *Hot* case situations, through the *Orchestrator*, the *Predictive Simulation* receives a set of **Predictive Rules**, randomly selected from one of the 15 available sets (as described in Section IV-B) and the *Event Generator* receives the relative events.

The Experiment focused on the evaluation of the time required to launch and execute the rule generation process ($\Delta t_{generate}$), self-inject the generated rule in the knowledge base (Δt_{inject}), and notify a violation (Δt_{firing}).

According to the CONCERN operational behavior, it generates predictive rules once a prediction is received. In particular, CONCERN loads within its knowledge Base the meta-rule capable of capturing the forecasted event and enacting the rule generation related to the received prediction.

An example of the meta-rule for the self-generation procedure is shown in the Listing 7. In detail, once

```
1  declare ConcernDTEvent
2    @role( event )
3    @timestamp ( timestamp )
4  end
5
6  rule "ppmetarule-1"
7    no-loop
8    salience 10
9    dialect "java"
10 when
11   $$$aEvent : ConcernDTForecast(
12     this.getSenderID == "DigitalTwin" ,
13     this.getDestinationID == "RuntimeMonitoring");
14 then
15   RulesGenerator.generateRuleFromDTForecast($$$aEvent);
16   retract($$$aEvent);
17 end
```

LISTING 7. Rule for enacting rules self-generation through DT forecast.

the forecast event is received (row 11), the CONCERN monitoring uses its parameters (that could include the `trustedIntervalInSeconds`, `forecastedProperty`, `thresholdValue`) to select the rule template matching the forecast sent by the *Predictive Simulation*. An example of a CONCERN forecast event called the `ConcernDTForecast` is shown in figure Figure: 6.

Once the correct template is selected and instantiated with the parameters requested, CONCERN self-injects the new rule for monitoring the occurrence of the event forecasted.

a: EXPERIMENTS [5] RESULTS ANALYSIS

As Experiment [1-4], Experiment 5 has been repeated 20 times considering both the *Hot* and *Cold* setup and average values computed. These include:

- Time required for loading the the predictive meta-rule: $\Delta t_{loading}$;
- Time required for generation: $\Delta t_{generate}$
- Time required for injection: Δt_{inject};
- Time required for firing notification: Δt_{firing}.

Experiment 5 results are reported in the last two columns of Table 1. Considering the loading time ($\Delta t_{loading}$), the performance of CONCERN in both cases is slightly higher than the results collected in Experiment 1. This is mainly due to the structure of a predictive meta-rule, which contains additional external library checks than a *Simple Rule*.

In conclusion, the loading of prediction rules may be slower than the other kinds of rules and, in any case, affected by the limitations already discussed in Experiments 1 and 2 of the previous section.

Considering the other time interval of Experiment 5, the following average results have been collected:

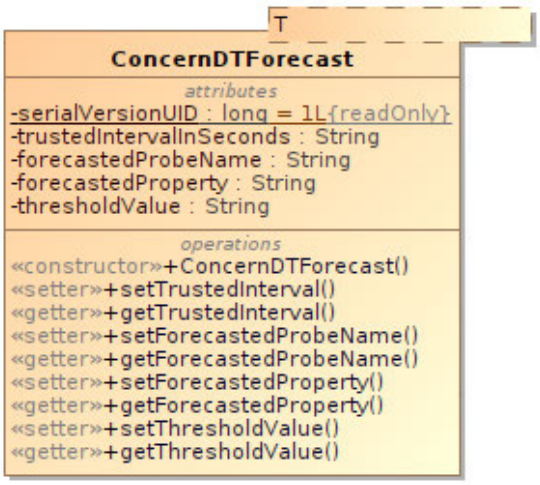

FIGURE 6. ConcernDTForecast class view.

1) **Generation of monitoring rule from Predictive Data:** 10 ms;
2) **Injecting monitoring rule:** 44ms;
3) **Evaluation and generation of monitoring rule violation:** 96 ms.

The use of MOTEF provides clear measures of the effectiveness of CONCERN in dealing with smart predictions and lets us reply to the **RQ2** computing its hard real-time boundaries. Indeed, excluding the communication delay among components, the CONCERN time constraints are:

- 54ms: the minimal time necessary to generate and load a predictive rule for the analyzed *Runtime Monitoring*.
- 61 ms: the minimal time necessary to the *Runtime Monitoring* from generating a rule to the notification of its violation.

Experiment 5 shows that CONCERN can be adopted in every situation in which the reaction time is not less than the above limits.

VII. THREATS TO VALIDITY

This section discusses threats to the internal and external validity threats of the study presented in this paper. Concerning the internal validity, i.e., the amount of confidence in the reported evaluation results, different aspects can be considered:

Fairness of the emulated SMF artifacts

MOTEF emulated components (*Rules Manager* and *Event Generator*) rely on a random selection algorithm based on a predefined set of data that mitigates the risk of biasing the selection of input in favor or against the *Runtime Monitoring* quality evaluation results.

Simple and Complex Rules generation

To avoid bias in the rules generation, considering the *Simple Rules* manual generation, the first 50 rules have been generated considering the same logical constructs as it has been already described in IV-A. From these rules, all the outcomes generated by the script proposed in 1 and 2 have been manually checked by authors that have consolidated experience in Drools rules generation and management. For the same reason and in the same way, the *Complex Rules* 5, generated by automatic composition of *Simple Rules*, has been manually checked from the syntactic and semantic point of view.

Considering the external validity, the bias could derive from: *Usage of specific runtime monitoring*.

To avoid this bias, the selection of the *Runtime Monitoring* (CONCERN) has been done considering that it has already been tested in more than 3 EU projects, is subject to more than 20 research papers, and has been evaluated and validated in several industrial use-cases. This can guarantee that results are independent of the intrinsic quality aspect of the *Runtime Monitoring*.

Another aspect considered from the external validity is related to the *Reliability of measures.*

This threat has been mitigated by executing the experiment 20 times and calculating the average values obtained, noticing slight variance and deviation.

VIII. CONCLUSION FUTURE WORKS

This paper presented the MOnitoring TEsting Framework (MOTEF), a testing framework for evaluating the performance of a runtime monitoring system in isolation and when it needs to provide smart predictions. The purpose is to establish working time boundaries to avoid runtime monitoring becoming a bottleneck for communication and computational processes and to better design and manage the overall system architecture. The result of using MOTEF in evaluating the performance of an existing monitoring system is provided and discussed, considering two research questions related to the responsiveness and effectiveness of predictions. The preliminary validation shows the usefulness of MOTEF in precisely analyzing runtime monitoring working time. Additionally, the positive results are promoting MOTEF as a reference testing framework for evaluating the performance of different runtime monitoring systems inside ongoing research projects SERICS[5] and RESTART.[6] We are working on including an intelligent result analysis system to showcase the collected results better.

As future works, considering the exciting results gathered, we are working on structuring the framework with the possibility of selecting and evaluating further properties not only related to responsiveness, i.e., security, stress test, scalability, and costs. Additional attention will be devoted to simulating possible behavioral models representing specific sensors and component interactions. These further activities require enriching the proposed MOTEF framework with emulated or dedicated components.

REFERENCES

[1] Q. M. Rahman, P. Corke, and F. Dayoub, "Run-time monitoring of machine learning for robotic perception: A survey of emerging trends," *IEEE Access*, vol. 9, pp. 20067–20075, 2021.

[5] https://serics.eu/
[6] www.fondazione-restart.it

[2] A. Correia, D. Stokic, R. Neves-Silva, A. R. Campos, and J. Agirre, "Collaborative environment for intelligent monitoring in manufacturing industry," in *Proc. IEEE Int. Technol. Manage. Conf. (ICE)*, Jun. 2008, pp. 1–6.
[3] D. Popescu, F. Stoican, G. Stamatescu, O. Chenaru, and L. Ichim, "A survey of collaborative UAV–WSN systems for efficient monitoring," *Sensors*, vol. 19, no. 21, p. 4690, Oct. 2019.
[4] K. Huang, Z. Tao, C. Wang, T. Guo, C. Yang, and W. Gui, "Cloud-edge collaborative method for industrial process monitoring based on error-triggered dictionary learning," *IEEE Trans. Ind. Informat.*, vol. 18, no. 12, pp. 8957–8966, Dec. 2022.
[5] V. Garousi, A. Rainer, P. Lauvås, and A. Arcuri, "Software-testing education: A systematic literature mapping," *J. Syst. Softw.*, vol. 165, Jul. 2020, Art. no. 110570.
[6] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. Tripp, "Software engineering body of knowledge," in *Proc. IEEE-CS SWEBOK V4 Public Rev.*, Jul. 2024, p. 1235.
[7] S. Daoudagh A. Calabrò and E. Marchetti, "Towards enhanced monitoring framework with smart predictions," *Logic J. IGPL*, 2024, doi: 10.1093/jigpal/jzae010.
[8] T. Henzinger, M. Karimi, K. Kueffner, and K. Mallik, "Runtime monitoring of dynamic fairness properties," in *Proc. ACM Conf. Fairness, Accountability, Transparency*, Chicago, IL, USA, Jun. 2023, pp. 604–614.
[9] M. Vierhauser and A. Egyed, "Runtime monitoring for systems of system—A closer look on opportunities for manufacturers in the context of Industry 4.0," in *Digital Transformation*. Cham, Switzerland: Springer, 2022, pp. 203–222.
[10] A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta, "GLIMPSE: A generic and flexible monitoring infrastructure," in *Proc. 13th Eur. Workshop Dependable Comput.*, Pisa, Italy, May 2011, pp. 73–78.
[11] P. Barsocchi, A. Calabrò, E. Ferro, C. Gennaro, E. Marchetti, and C. Vairo, "Boosting a low-cost smart home environment with usage and access control rules," *Sensors*, vol. 18, no. 6, p. 1886, Jun. 2018.
[12] A. Calabró, F. Lonetti, and E. Marchetti, "Monitoring of business process execution based on performance indicators," in *Proc. 41st Euromicro Conf. Softw. Eng. Adv. Appl.*, Madeira, Portugal, Aug. 2015, pp. 255–258.
[13] A. Calabró, F. Lonetti, E. Marchetti, and G. O. Spagnolo, "Enhancing Bus. Process performance analysis through coverage-based monitoring," in *Proc. 10th Int. Conf. Quality Inf. Commun. Technol. (QUATIC)*, Lisbon, Portugal, Sep. 2016, pp. 35–43.
[14] D. de Freitas Bezerra, V. W. C. de Medeiros, and G. E. Gonçalves, "Towards a control-as-a-service architecture for smart environments," *Simul. Model. Pract. Theory*, vol. 107, Feb. 2021, Art. no. 102194.
[15] T. Zoppi, A. Ceccarelli, and A. Bondavalli, "MADneSs: A multi-layer anomaly detection framework for complex dynamic systems," *IEEE Trans. Depend. Secure Comput.*, vol. 18, no. 2, pp. 796–809, Mar. 2021.
[16] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker, "Model-driven architectural monitoring and adaptation for autonomic systems," in *Proc. 6th Int. Conf. Autonomic Comput.*, Jun. 2009, pp. 67–68.
[17] L. Aceto, A. Achilleos, D. P. Attard, L. Exibard, A. Francalanza, and A. Ingólfsdóttir, "A monitoring tool for linear-time μ HML," in *Proc. Int. Conf. Coordination Lang. Models*, Lucca, Italy. Cham, Switzerland: Springer, Jun. 2022, pp. 200–219.
[18] L. Aceto, A. Achilleos, E. Anastasiadi, and A. Ingolfsdottir, "Axiomatizing recursion-free, regular monitors," *J. Log. Algebr. Methods Program.*, vol. 127, Jun. 2022, Art. no. 100778.
[19] L. Aceto, I. Cassar, A. Francalanza, and A. Ingolfsdottir, "Bidirectional runtime enforcement of first-order branching-time properties," *Log. Methods Comput. Sci.*, vol. 19, no. 1, pp. 1–44, Feb. 2023.
[20] D. P. Attard, L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen, "Better late than never or: Verifying asynchronous components at runtime," in *Proc. Int. Conf. Formal Techn. Distrib. Objects, Compon., Syst.*, Valletta, Malta. Cham, Switzerland: Springer, Jun. 2021, pp. 207–225.
[21] L. Aceto, D. P. Attard, A. Francalanza, and A. Ingólfsdóttir, "A choreographed outline instrumentation algorithm for asynchronous components," 2021, *arXiv:2104.09433*.
[22] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Comput. Netw.*, vol. 57, no. 9, pp. 2093–2115, Jun. 2013.
[23] S. Caruso, M. Bruccoleri, A. Pietrosi, and A. Scaccianoce, "Artificial intelligence to counteract 'KPI overload' in business process monitoring: The case of anti-corruption in public organizations," *Bus. Process Manage. J.*, vol. 29, no. 4, pp. 1227–1248, Jan. 2023.
[24] E. Cioroaica, F. Di Giandomenico, T. Kuhn, F. Lonetti, E. Marchetti, J. Jahic, and F. Schnicke, "Towards runtime monitoring for malicious behaviors detection in smart ecosystems," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Thomas Kuhn, France, Oct. 2019, pp. 200–203.
[25] G. Park and M. Song, "Optimizing resource allocation based on predictive process monitoring," *IEEE Access*, vol. 11, pp. 38309–38323, 2023.
[26] M. Perno, L. Hvam, and A. Haug, "A machine learning digital twin approach for critical process parameter prediction in a catalyst manufacturing line," *Comput. Ind.*, vol. 151, Oct. 2023, Art. no. 103987.
[27] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "ROSMonitoring: A runtime verification framework for ros," in *Towards Autonomous Robotic Systems*, A. Mohammad, X. Dong, and M. Russo, Eds. Cham, Switzerland: Springer, 2020, pp. 387–399.
[28] S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "A formal toolchain for offline and run-time verification of robotic systems," *Robot. Auto. Syst.*, vol. 159, Jan. 2023, Art. no. 104301.
[29] G. Fainekos, B. Hoxha, and S. Sankaranarayanan, "Robustness of specifications and its applications to falsification, parameter mining, and runtime monitoring with S-Taliro," in *Runtime Verification (RV)*, vol. 11757. Cham, Switzerland: Springer, 2019, pp. 27–47.
[30] A. Calabrò, S. Daoudagh, and E. Marchetti, "MENTORS: Monitoring environment for system of systems," in *Proc. 17th Int. Conf. Web Inf. Syst. Technol.*, 2021, pp. 291–298.
[31] S. Daoudagh, E. Marchetti, A. Calabrò, F. Ferrada, A. I. Oliveira, J. Barata, R. Peres, and F. Marques, "An ontology-based solution for monitoring IoT cybersecurity," in *Internet of Things. IoT Through a Multi-Disciplinary Perspective*. Cham, Switzerland: Springer, 2022, pp. 158–176.
[32] A. Bertolino, A. Calabrò, and G. De Angelis, "Adaptive SLA monitoring of service choreographies enacted on the cloud," in *Proc. IEEE 7th Int. Symp. Maintenance Evol. Service-Oriented Cloud-Based Syst.*, Sep. 2013, pp. 92–101.
[33] A. Calabrò, E. Cioroaica, S. Daoudagh, and E. Marchetti, "BIECO runtime auditing framework," in *Proc. Comput. Intell. Secur. Inf. Syst. Conf.*, vol. 1400, Bilbao, Spain. Cham, Switzerland: Springer, Sep. 2021, pp. 181–191.
[34] P. Barsocchi, A. Calabrò, A. Crivello, S. Daoudagh, F. Furfari, M. Girolami, and E. Marchetti, "COVID-19 & privacy: Enhancing of indoor localization architectures towards effective social distancing," *Array*, vol. 9, Mar. 2021, Art. no. 100051.

ANTONELLO CALABRÒ graduated in computer science engineering.

He is a Researcher with ISTI-CNR. He has participated in different roles for ISTI-CNR to several national and international EU projects and actively participated in more than 15 national and international projects. His research activities are focused on the design and development of smart monitoring infrastructure, responsive and adaptable complex event processing, and cybersecurity assessment. His research activities have been applied to several application domains, such as smart environments, SoSs, the IoT, vehicular networks, emerging technologies, and UAVs.

EDA MARCHETTI received the Graduate (summa cum laude) and Ph.D. degrees in computer science from the University of Pisa.

She is a Senior Researcher with ISTI-CNR. She has been responsible for ISTI-CNR of several national and international EU projects and actively participated in more than 20 national and international projects. Her main research focuses on software testing in general, and in particular, on introducing novel methodologies for testing emerging software technologies, cybersecurity, and value-based and ethical aspects. Her research topics include security and privacy testing, testing of access control systems, model-based testing, SOA and component-based testing, requirement management and assessment, monitoring business process, human-centric testing process management and scheduling, operational and structural testing, interoperability testing, domain-specific testing, product certification, and assessment.

Open Access funding provided by 'Consiglio Nazionale delle Ricerche-CARI-CARE-ITALY' within the CRUI CARE Agreement