# OREO: A Tool-Supported Approach for Offline Run-time Monitoring and Fault-Error-Failure Chain Localization

Leonardo Scommegna[a], Benedetta Picano[a], Roberto Verdecchia[a] and Enrico Vicario[a]

[a] *Department of Information Engineering, University of Florence, Florence, Italy*

ARTICLE INFO

*Keywords*:
Dynamic Analysis
Fault Localization
Monitoring
Reliability
Runtime Verification

ABSTRACT

The ever-increasing complexity of modern software architectures has exacerbated the need for advanced software tools able to track software execution traces to improve software reliability.

In this paper, we present OREO, a tool for offline and run-time monitoring and fault localization. The tool implements a novel method enabling to trace software executions to discover the run-time status, dependencies, and interactions among software components.

OREO is based on a timeline extractor, i.e., an abstraction of component lifecycles and their interactions. The timeline extractor enables the tool to perform a runtime health state examination of the software under analysis. The profiler is then used to analyze the error propagation originated during the running states among software components. In so doing, the possible fault-error-failure chains are identified.

To showcase the capabilities of OREO and its flexibility, we report the execution of the tool on three software projects of different nature, sizes, and architectures. The analysis results in the localization of fault-error-failure chains and safe components of the three software projects.

A discussion of the versatility, scalability, and applicability of the proposed tool to a rich variety of application contexts is provided.

## 1. Introduction

Complex distributed software architectures are nowadays pervasive. The diffusion of such complex systems requires advanced reliability techniques to ensure functional requirements and quality attributes. Typically, software reliability can be investigated and guaranteed through software testing, formal verification, reliability prediction and estimation, and standard compliance [1, 2, 3, 4]. However, traditional software verification and validation techniques are not sufficient to ensure software reliability due to the high level of complexity and dependencies existing in today's systems that emerge exclusively during execution [1].

Many software systems rely on stateful sessions, processing ordered external event sequences where responses depend on current and prior events (e.g., a marketplace app tracking inserted cart items before checkout). Unpredictable event sequences amplify system complexity, making it challenging, and often even unfeasible, to comprehensively evaluate the correctness of the system at testing time.

A faulty component might enter an erroneous state after an event, but the error could remain latent, manifesting an external failure much later under specific event sequences. In complex cases, errors may propagate silently to otherwise correct components. For instance, a correct component might rely on a value from a faulty component. If it updates its state based on this erroneous value, it could become erroneous itself. This can cause sporadic, inconsistent failures in functional components due to error propagation chains. Such elusive faults, driven by unpredictable event sequences, are termed heisenbugs [5].

Software-intensive systems pose a number of open challenges. During development, anticipating runtime error propagation patterns is difficult [6]. Once in production instead, reproducing failures induced by heisenbugs, tracing propagation paths, and isolating the root faulty component is hard, especially in modern systems that handle multiple parallel sessions, each triggering independent error propagations.

To address these issues, various approaches have been proposed, including testing methodologies [6] and proactive maintenance solutions, often referred to as software rejuvenation [7]. Additionally, runtime verification [1, 8] strategies, particularly logging and runtime monitoring [9, 10, 11], are well-suited for these challenges, as they enable the extraction and analysis of system execution traces. However, these approaches face several key challenges. Manual code instrumentation is costly and error-prone [12]. In contrast, non-invasive monitoring and tracing frameworks eliminate the need for manual source code modifications and their related drawbacks, however they typically generate unsustainable computational overhead [13, 12]. Some strategies solve the overhead problem by performing subsampling of events to be observed, such as selective event logging [14] or reduced sampling rates [15]. Although these filtering techniques can mitigate the overheads, they risk incomplete reconstruction of error propagation. Moreover, logging tools inherently lack native support for fault-error-failure propagation analysis, and manual analysis becomes unfeasible when error propagations span extended execution times or involve chains of multiple interconnected events [16]. This limitation is further exacerbated by the presence of multiple sessions that are extracted by logging tools as a single

✉ leonardo.scommegna@unifi.it (L. Scommegna); benedetta.picano@unifi.it (B. Picano); roberto.verdecchia@unifi.it (R. Verdecchia); enrico.vicario@unifi.it (E. Vicario)

ORCID(s): 0000-0002-7293-0210 (L. Scommegna); 0000-0003-4970-1361 (B. Picano); 0000-0001-9206-6637 (R. Verdecchia); 0000-0002-4983-4386 (E. Vicario)

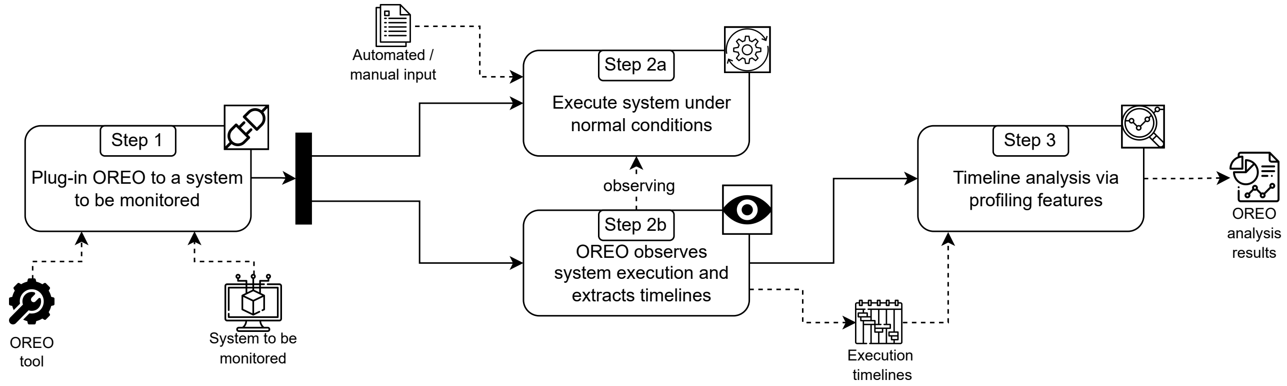
Automated /
manual input
Step 2a
Execute system under normal conditions
Step 1
Plug-in OREO to a system to be monitored
observing
Step 2b
OREO observes system execution and extracts timelines
Step 3
Timeline analysis via profiling features
OREO analysis results
OREO tool
System to be monitored
Execution timelines

**Figure 1:** High-level overview of the OREO execution

trace of interleaved events, further complicating propagation analysis.

While runtime verification tools target model consistency, performance, and/or security [12, 17], they remain underutilized for systematic error propagation analysis in complex systems. This leaves critical gaps in understanding the so-called fault-error-failure (FEF) chains [18] in software-intensive environments.

In this work, we propose a novel conceptual framework for runtime verification and monitoring of software-intensive systems, aimed at comprehending and analyzing the propagation of faults, errors, and failures. Our approach addresses the aforementioned limitations of traditional logging tools and current runtime verification techniques through automated, non-intrusive instrumentation and efficient monitoring. This enables systematic error propagation analysis, even for complex, interleaved execution traces. Additionally, we introduce an abstraction called the *timeline* to represent the concurrent processes occurring at runtime, with particular emphasis on the lifecycle and dynamic dependencies that components establish in response to specific sequences of external events. As a concrete implementation of our framework, we present the Offline RuntimE mOnitoring (OREO) open-source software tool. OREO is capable of observing the business logic of Java/-Jakarta Enterprise Edition web applications and extracting the corresponding timelines. To support fault localization and the identification of fault-error-failure (FEF) chains, OREO is equipped with a profiler module that performs offline analysis of the extracted timelines.

Our experimental results demonstrate that our framework requires minimal instrumentation effort regardless of the size of the target system, imposes only minimal overhead in terms of memory usage and response delay, and effectively supports developers during both the design and debugging phases. In the design phase, the framework highlights crucial components or methods that may facilitate error propagation; during debugging, it aids in identifying components affected by heisenbugs.

The main contributions of the paper are the following:

- The proposal of an abstraction, referred to as *timeline*, able to represent the behavior of the concurrent processes occurring at runtime, bridging the gap between the offline design of the logic and its online evolution;
- The design and development of an open-source runtime monitoring tool, able to observe the evolution of the business logic and extract the corresponding timeline.
- The application of the OREO tool to a concrete application scenario. For this purpose, we implemented a profiler that analyzes the extracted timeline relying on the concept of *error propagation* and FEF chains. Due to the general connotation of the OREO tool, a further discussion is provided to expose the rich variety of software monitoring problems to which the timeline extractor can provide support in decision-making policies;
- In-depth performance evaluation of the OREO tool by considering three heterogeneous software projects.
- The source code of the OREO tool. The open-source repository is made available online at the following link: `https://github.com/STLab-UniFI/oreo-tool`.

For the best of our knowledge, this is the first study developing a tool to analyse cause-effects fault-failure relations, i.e., fault-error-failure chains, abstracting from the type of fault the occurred or the failure manifested by the system.

## 2. Overview

In this section, we provide an overview of the OREO tool presented in this research, both in terms of intuitive description of the problem tackled by the tool (Section 2.1) and general functioning of the tool (Section 2.2, see also Figure 1.)

### 2.1. The Problem Addressed

During the execution of a software-intensive system, instances of programming classes are dynamically created at runtime to satisfy the functional requirements of the system. Such instances, commonly storing their state in volatile memory, communicate among each other, exchanging and manipulating information during the execution of a use case scenario. If, during such execution, a bug is encountered in one of the runtime instances, the error can propagate from one instance to the other as they communicate. These types of heisenbugs [5] may not lead to an instant failure of the system, leading to complex error propagation chains involving numerous instances and interactions before the failure becomes noticeable to the end user (if ever). Identifying runtime heisenbugs, given the impediments related to trace them back to a specific component, is therefore a cumbersome and time-consuming process [6, 16].

```
 1  ────────────────────────────── A.java ──────────────────────────────
 2  @SessionScoped
 3  public class A {
 4      public void startProcedure(double p, double q) {
 5          B bObject = ... // initialization of B object
 6          // some instructions
 7          bObject.initialize(p, q); // A-B interaction
 8          // other instructions
 9      }
10  }
11  ────────────────────────────── B.java ──────────────────────────────
12  @RequestScoped
13  public class B {
14      private static final double CONST = 10.0;
15      private double nonZeroState;
16
17      public void initialize(double startingValue, double factor) {
18          this.nonZeroState=startingValue-CONST;// BUG! state should be checked
19          C cObject = ... // initialization of C object
20          double newCState = this.nonZeroState * factor;
21          cObject.setCState(newCState); // B-C interaction
22      }
23  }
24  ────────────────────────────── C.java ──────────────────────────────
25  @SessionScoped
26  public class C {
27      private double cState;
28
29      public void setCState(double state) {
30          this.cState = state;
31      }
32
33      public double getUpdatedCState(double num) {
34          return num/this.cState;
35      }
36  }
```

**Listing 1:** Source code of the exemplary software system.

As a simple example, consider three Java classes: A, B, and C. A portion of their code is shown in Listing 1. As illustrated, class A, within its method `startProcedure`, invokes class B at line 7, passing the two double values it receives as input. Class B, within its `initialize` method, interacts with class C through the invocation of the `setCState` method at line 21. This method invocation has the side effect of updating the internal state of C, namely `cState`. Additionally, each class is annotated with a Context and Dependency Injection (CDI) annotation: A and C are marked as `@SessionScoped`, while B is annotated as `@RequestScoped`. In Java Enterprise Edition, these annotations define the lifespan of objects: `@SessionScoped` binds the lifespan of an object to the duration of a session, whereas `@RequestScoped` restricts it to a single request.

Let us consider a scenario of usage of a software system that includes classes A, B, and C. The exemplary scenario is graphically represented in Figure 2 and involves four subsequent requests in time to the software-intensive system ($R_1$-$R_4$, depicted on the x-axis in Figure 2). A request is defined as an interaction of the end user with the presentation layer of the software-intensive system, e.g., an input provided by the end user through a graphical user interface or command line. To satisfy the requests, the system instantiates three different objects: one of type A, one of type B, and one of type C (reported on the y-axis and depicted as rectangles colored violet, green, and yellow, respectively, in Figure 2).

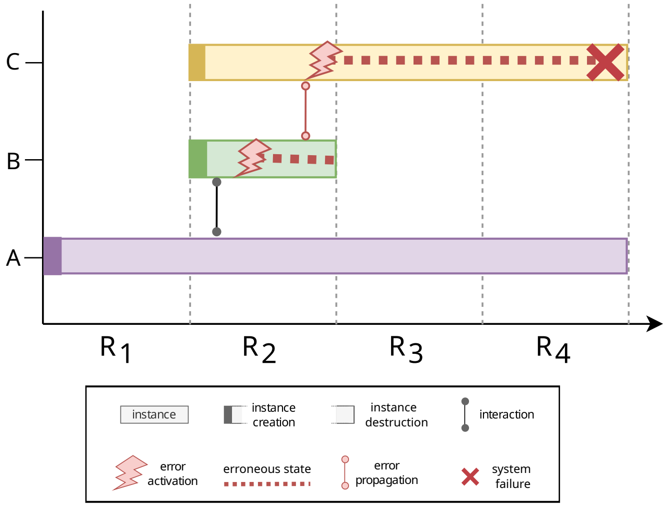

**Figure 2:** Runtime error propagation example of Listing 1.

The first request ($R_1$) leads to the creation of an instance of A. Such item (and its transient state) is kept in volatile memory also during the time span needed to satisfy the subsequent requests ($R_2$, $R_3$, and $R_4$). During $R_2$, instance A communicates information to the new instance B, which lives only along request $R_2$. While $R_1$ was characterized by a correct behaviour, after instance A passed information to instance B, a fault residing in B originates an error while this latter instance is processing the data (depicted with a lightning bolt icon in Figure 2). Before terminating its life, B communicates with instance C, also created in request $R_2$, propagating the error once more. Concretely, during request $R_2$, the user invokes the `startProcedure` method of object A, passing the value 10 as the actual parameter for `p` and 5 for `q`. As confirmed by the A.java code in Listing 1, A interacts with B by invoking its method `initialize` with `startingValue` = 10 and `factor` = 5. As shown in the code of B.java in Listing 1, when `initialize` receives `startingValue` = 10, an error is triggered in B, leading to the assignment of 0 to the variable `nonZeroState`. Subsequently, based on its erroneous state, B invokes `setCState` method of C with an incorrect parameter (0), propagating its error. `SetCState` then updates the internal state of C with the erroneous value passed by B and concludes the processing of request $R_2$ without any failure being externally visible to the user. At the end of $R_2$, since B is `@RequestScoped`, it is destroyed.

As time progresses, while satisfying the third request $R_3$, the error originating in instance A remains silent. As the fourth and last request $R_4$ constituting the use case scenario terminates, the error finally leads to the failure of the software-intensive system. Specifically, assume that during $R_3$, the method `setCState` is not invoked. As a result, at the beginning of $R_4$, the error in object C persists. Finally, suppose that during $R_4$, the user invokes the `getUpdatedCState` method. As shown in C.java code of Listing 1, this method triggers a division by the internal state of C which in this case is zero. Java raises an ArithmeticException, causing the system to fail. Despite the simplicity of the code in the example, the system failure (in $R_4$) occurs significantly later than the activation of the initial error that triggered the propagation (in $R_2$).

Furthermore, the component manifesting the failure contains no coding faults, and at the time of failure, the responsible component is no longer among the active objects. These conditions dramatically complicate reproducing the failure and detecting or removing the underlying fault. In the presence of software-intensive systems with multiple components and interactions, an error has more opportunities to propagate, leading to even more complex propagation scenarios and making the identification of the responsible component significantly more challenging. OREO is designed to analyze, through a formalisation and monitoring strategy, the runtime state of instances and the communication among them, allowing to study FEF chains such as the one reported in the example above. Throughout the paper, we will follow the taxonomy of fault, error, and failure outlined by *Avizienis et al.* [18].

### 2.2. OREO at a Glance

As introduced in Section 2.1, understanding the runtime behavior of the system, visualizing error propagation scenarios, and identifying the potential faulty components that caused a failure is complex. To this end, we propose OREO, a tool designed to support the comprehension and analysis of Fault Error Failure chains. A high-level overview on utilizing OREO is depicted in Figure 1. As shown in the figure, executing OREO relies on three main steps. In the first step (Step 1), OREO is plugged-in into the system to be monitored. As further detailed in Section 3.3, this process intuitively consists of specifying OREO as a Context and Dependency Injection extension in the system to be monitored. In the considered JEE context, practically this step consists of adding a line to the configuration file of the build automation tool used (e.g., Maven[1]).

The second step of OREO's execution consists of two steps executed in parallel (Step 2a and Step 2b). On one hand, the system to be monitored is executed under normal conditions, e.g., by manually or automatically executing use case scenario or a predefined test suite (Step 2a). At the same time, OREO observes the system execution and extracts and stores locally the timelines observed within the business logic of the system under analysis (Step 2b). As final output of this latter step, once the execution of the system to be monitored is terminated, OREO provides the set of observed execution timelines.

As last step of OREO's execution, the timelines collected through Step 2b can be analyzed according to one of the profiling features provided out of the box with OREO (Step 3). As further detailed in Section 4.3, OREO currently implements three different profiling features, namely identification of safe and unsafe instances, identification of FEF root instances, and identification of FEF scenarios. The final output of the execution of OREO is the analysis results of the monitored system according to one or more of the selected profiling features.

## 3. The OREO Tool

In this section, we document the design and implementation of the OREO tool. Additionally, this section covers the background concepts leveraged by the tool, including a description of the timeline abstraction used by OREO and how this abstraction is extracted during the execution of a system under analysis. This essentially covers steps 1, 2a, and 2b as represented in Figure 1.

### 3.1. Representing the Components Life: the Timeline Abstraction

In order to provide a convenient and intuitive way to represent the dynamic evolution of the business logic, we formalize an abstraction named *timeline* able to represent (i) how the business logic hosts concurrently living components and (ii) how the components react if they are subject to a specific input sequence arriving over time. In the context of our study, a *timeline* represents the evolution of the business logic over a specific sequence of events along a single user session. A user session is defined as a sequence of consecutive, time-ordered interactions performed by the same user on the given software-intensive system In the business logic layer, each active user session is managed separately from the others. Particularly, a component belonging to a specific user session cannot interact directly with a component belonging to another session. The isolation of session-specific elements ensures that the actions of one user do not interfere with the session of another user. In OREO, the management of timelines reflects the session management mechanism naturally applied by the business logic, i.e., each session is collected and analyzed separately in an ad-hoc timeline. Thus, a timeline provides a snapshot of an execution scenario performed by a user, capturing the interaction between the presentation layer, which is responsible for the user interface, and the components operating in the business logic layer. The timeline abstracts the concrete behavior to identify the components that persist in memory over time, as well as the interactions and dependencies that occur among these components during runtime. An example of a timeline abstraction is depicted in Figure 3.

Formally, a timeline $TL$ is a tuple $\langle T, C, J, I, O \rangle$ where:

- $T := \{t_0, t_1, ...t_n\} \subset \mathbb{R}$ is a sequence of time points, with $t_0 = 0$ and $t_n > t_{n-1}$, representing the time point at which the presentation layer accepts and forwards the $n$-th interface interaction to the business logic. Time points are represented along the x-axis of Figure 3. By convention, $t_n$ closes the sequence, denoting the end of the observation interval captured

[1] `https://maven.apache.org`. Accessed 3rd October 2024.

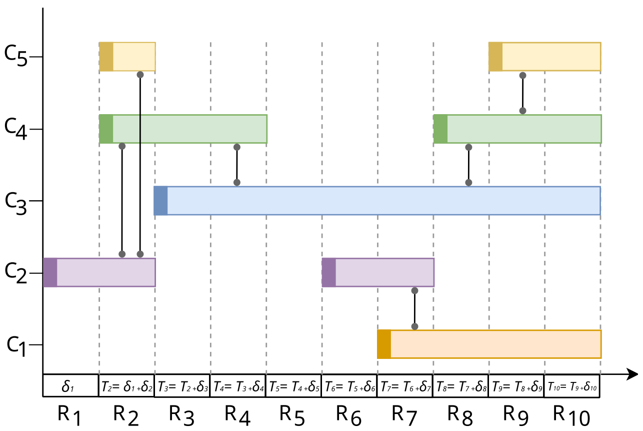

**Figure 3:** Timeline abstraction of a business logic made of 5 component types along a user interaction with 10 steps.

by the snapshot. The (left-closed right-open) interval $[t_i, t_i + 1)$ is referred to as an *epoch* and denoted by $R_i$, and its duration $t_{i+1} - t_i$ is denoted $\delta_i$. In a practical perspective, $R_i$ encompasses all the back-end actions performed after the interface interaction received at time $t_i$, and it also includes the idling time during which the back-end waits for the next interaction. Formally $\delta_i = proc^{R_i} + idle^{R_i}$ where $proc^{R_i}$ represents the processing time and $idle^{R_i}$ the idle time of $R_i$.

The intervals between subsequent interactions have a duration that guarantees that a new interaction does not arrive before the current one has been processed. Formally, $t_i + proc^{R_i} > t_{i+1}, \forall i \in [0, n)$. If sequences of close requests or even bursts of interactions should occur, the presentation layer will take care of keeping the system synchronous and will forward the interactions to the business logic in the order in which they were executed.

- $C$ is a set of components ($C_1$ through $C_5$ in the example of Figure 3). A multiplicity of components exist throughout the timeline. This is represented in the y-axis of Figure 3.

- $J$ is the set of all the component instances. In fact, during the system execution, a component can be instantiated, (i.e., created and placed in memory) multiple times. Let $j_c^i$ be the $i$-th instance of $c$ during the Timeline $TL$, then $J_c$ is the set of all instances of $c$. Formally:

  $$J_c = \{j_c^i \in J | 0 \leq i < m\}$$

  Let $m$ be the number of instances of $c$ during $TL$. Following the definition of $J_c$, the set $J$ can also be defined as follows: $J = \bigcup\limits_{c \in C} J_c$.

  An instance $j_c^i$ is characterized by a specific life cycle. The life cycle of $j_c^i$, denoted as $l_{j_c^i} = [R_b^{j_c^i}, R_e^{j_c^i}]$, is the set of ordered epochs during which $j_c^i$ exists. Thus, $R_b^{j_c^i}$ is the epoch at which $j_c^i$ is instantiated, and $R_e^{j_c^i}$ is the epoch at which $j_c^i$ is destroyed.

  The life of each instance $j_c^i$ is represented graphically by a rectangle spanning along the line of $c$ for all the epochs in $l_{j_c^i}$.

  $PL \in J$ is a special fictitious instance that accounts for the presentation layer.

- $I$ is the set of all interactions that occur between components along the Timeline $TL$. Interactions are divided into two main subsets: $D$ and $U$, hence it follows that $I = D \cup U$.

  $D \subseteq J \times J \times R$ is a collection of undirected interactions occurring between instances within epochs of the timeline: $\langle j_{c1}^i, j_{c2}^j, R \rangle \in D$ represents an invocation of $j_{c2}^j$ performed by $j_{c1}^i$, or vice versa, that occurs during the epoch $R$. Note that interactions are undirected, i.e. $\langle j_{c1}^i, j_{c2}^j, R \rangle$ does not specify whether $j_{c1}^i$ invokes $j_{c2}^j$ or vice versa. The main reason for leaving the direction unspecified is that the timeline abstraction aims at capturing dependencies, and the direction of calls cannot distinguish these unless we are also able to distinguish whether $j_{c1}^i$, $j_{c2}^j$, or both, undergo a side effect during the interaction, which is not supported by the automated logging process implemented by OREO. Note that $D$ is defined as a collection, not as a set, as there can be multiple equal occurrences of the same triple $\langle j_{c1}^i, j_{c2}^j, R \rangle$, each with its own identity, which will occur whenever the same epoch includes multiple calls occurring between $j_{c1}^i$ and $j_{c2}^j$. Note also that two instances $j_{c1}^i$ and $j_{c2}^j$ can establish a dependency $\langle j_{c1}^i, j_{c2}^j, R \rangle$ if and only if $R \in l_{j_{c1}^i}$ and $R \in l_{j_{c2}^j}$. In other words, two instances can establish a dependency if and only if there exists an overlap between their life cycles: $l_{j_{c1}^i} \cap l_{j_{c2}^j} \neq \emptyset$

  $U \subseteq J \times R$ represent a collection of interactions between the presentation layer $PL$ and an instance of $J$: $\langle PL, j, r \rangle \in U$. Within each epoch in $r \in R$, there exist exactly two interactions with $PL$, namely $u_b^r, u_e^r \in U$. $u_b^r$ occurs as the first interaction of the epoch, preceding any other event $D$ in $r$. Conversely, $u_e^r$ occurs at the end as the last interaction of the epoch.

- $O \subseteq I \times I$ is a partial order on $I$ that specifies the ordering in time for any 2 interactions.

The provided formalization identifies a continuous-time system ($T \subset \mathbb{R}$) in which intra-session requests are synchronous but inter-session requests are asynchronous. Subsequent requests are synchronous within the same session while they are asynchronous if they belong to different sessions. Separate session management by the business logic layer allows for individual consideration of each timeline, viewing the system as a collection of synchronously working

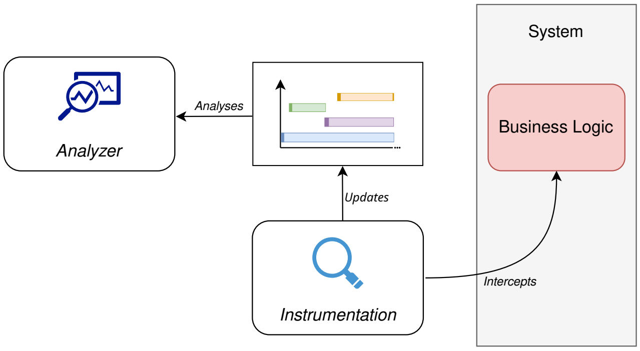

**Figure 4:** The Timeline extraction setup.

subsystems. Such separation allows us to consider single-session scenarios throughout the rest of the paper without loss of generality.

To provide a concrete example, let us consider the timeline represented in Figure 3. Request $r_6$ arrives at time $t_6$ identified as the timeshift of $\delta_6$ with respect to the previous $t_5$. During the response process, a component instance of type $c_2$ is created and subsequently, it is destroyed at request $r_7$. During its life as represented by the link at request $r_7$, the instance interacts with a component of type $c_1$ which lives until $r_{10}$.

### 3.2. Timeline Extraction

The business logic design defines the specific behavior that the application should maintain in response to individual requests. However, the dynamic evolution of the application, in terms of active instances and interactions, also depends on the sequence of interactions performed by the user.

Thus, the system identifies a number of dynamic data flow coupling scenarios almost impossible to consider at implementation time. In particular, the application could show at runtime unexpected and sometimes counter-intuitive patterns (e.g., a lower-scoped component that often lives for an extended period of time), or conversely, some rare input sequences could bring the system to a failure with high probability.

For these reasons, with OREO we proposed a solution aimed to exploit the timeline abstraction beyond the simple graphical support use. To do this, OREO leverages a timeline extraction setup that, as can be seen in Figure 4, is heavily inspired by configurations often proposed in the field of runtime verification [12]. The configuration of OREO consists of (i) the target *system* (i.e., the application we want to observe), (ii) an *instrumentation* module that, running concurrently with the system, extracts information at runtime, and (iii) an *analysis* module that processes the observation. Each request made by the user represents a system event and triggers the instrumentation which observes the business logic behavior during the response process. The information about component interactions, instantiation, and destruction is collected and used to update the execution trace of the application. In this form, the execution trace represents the timeline of the application that is progressively built at runtime by the instrumentation module request after request.

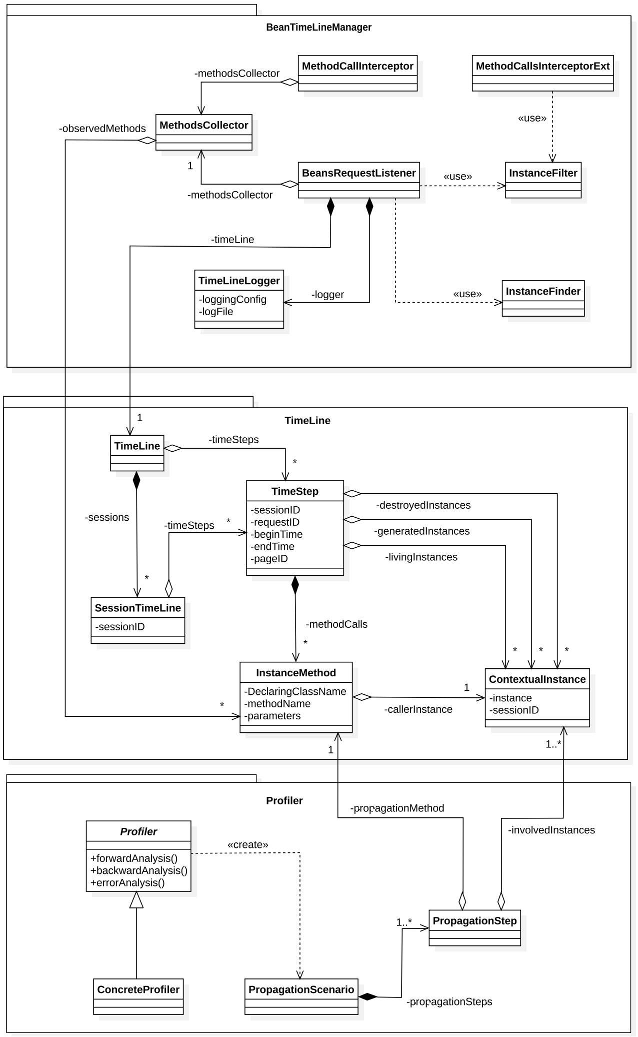

**Figure 5:** OREO Tool Class Diagram.

The identified setup enables both online and offline analysis of the extracted timeline classifying the analysis module as a monitor or a profiler respectively. In this work, we demonstrate how OREO can support the reliability assessment and improvement of software architectures. In particular, we show how the timelines extracted dynamically can be used to perform fault localization tasks and what-if analysis.

### 3.3. OREO: a JEE Implementation for Timeline Extraction

The concrete setup enabling the extraction of instances of the novel timeline abstraction was implemented concretely in the form of a JEE tool named OREO. More in detail, the tool can extract the necessary information from active CDI and Enterprise Java Beans (EJB) components. For this reason, its usage is particularly suited for the so-called stateful software architectures where the business logic is mainly hosted on the backend server.

The architectural view of the tool is represented in Figure 5 in form of a class diagram. Although with different nomenclature, the package structure is based on the setup organization of Figure 4. The `BeanTimeLineManager` package encapsulates the instrumentation module and represents the core of the tool, the `Timeline` package defines the representation of the timeline abstraction, and finally, the `Profiler` provides a concrete instance of the analysis module. In order to depict the main mechanism of OREO, an overview of the `BeanTimeLineManager` package is provided down below.

The class `BeansRequestListener` is the implementation of a CDI listener which is triggered right before and right after each response process to collect all the information of interest to update the timeline with a new time step. Before starting each response process, it exploits the `InstanceFinder`, a class that relies on the Service Provider Interface (SPI) of CDI, to obtain a snapshot of all the living business logic instances at that moment. Once generated the response, `BeansRequestListener` performs another snapshot of the instances. In this way, it can deduce the newly created, destroyed, and living instances through a comparison between the initial and final snapshots.

To also detect the component interactions, the listener consults the `MethodCollector` class, a simple class updated by the CDI interceptor `MethodCallInterceptor`. A CDI interceptor can intercept all the methods invoked by instances whose class is decorated with an ad-hoc annotation. However, annotating manually all the component classes of interest is time-consuming. To overcome this inconvenience, a CDI extension, represented by the class `MethodCallsInterceptorExt` was developed. A CDI extension allows the definition of specific behaviors during standard phases of the CDI framework. The present extension, in particular, will be triggered during one of the startup phases of CDI decorating automatically all the components of interest with the annotation required by `MethodCallInterceptor`.

OREO intercepts each request received by the target system and for each of them, it collects the following information distinguishing among multiple parallel sessions:

- the timestamp of the request arrival;
- the set of instances created during the response process;
- the sequence of interactions performed by components;
- the set of instances destroyed at the end of the response process.

OREO was designed in order to be flexible and allow a straightforward adoption in a “plug and play” fashion. The tool is implemented by enforcing that it does not require to modify the code of the target system. To start an observation process, it is sufficient to specify OREO as the CDI extension of the system. As default behavior OREO gathers information regarding all the components of the business logic. However, it is also possible to fine-tune the mechanism of the tool by defining ad-hoc filters to ignore some components and identify the business logic components of interest.

# 4. Fault Localization with OREO

In this section, we present the procedure for identifying fault-error-failure chains and detail the method for profiling the business logic, effectively covering step 3 shown in Figure 1.

## 4.1. Fault-Error-Failure Chain in the Business Logic

During the usage of an application, a component may enter into an erroneous state. The fault causing the error could be internal or external [18]. Internal faults originate inside the system boundaries, i.e., in the source code of the system itself. External faults instead, originate outside the system boundaries, e.g., faults caused by malfunctions manifested by external services like third-party API, sensors, and databases. An error can be then identified as a shift from the correct state of a component to an incorrect one. A failure is intended as an event that occurs when the service provided by the system deviates from the correct behaviour of the service. Once entered in an erroneous state, the system can continue to maintain its correct functioning for an unpredictable period of time. Failures will manifest when the system has to provide a service relying on its erroneous state.

During the response process, components interact with each other providing their functionalities. Let us consider components as individual systems. A component that manifests a failure during interaction will result in an external fault activation from the perspective of the other component. The external fault, in turn, may drive the correct component into an erroneous state.

Considering now the whole application as the system, a single internal error can propagate among components. Eventually, the system will manifest a failure tangible also for the end user, also known as *system failure*. The propagation phenomenon among components is usually referred to as FEF Chain [18]. An error activated in a component represents a starting point for the error that could spread in the application i.e., *error accumulation*.

The interactions among components and their order depend on the sequence of inputs that the user performs at runtime. Thus, also the FEF chain evolution establishes a strict relation with the unpredictable behavior of the user. In the context of a real application, the number of components in the business logic is considerable. Frequently, real applications also allow multiple alternative causes of actions to obtain the same goal. The high number of components involved jointly with the number of possible combinations of user interactions makes the static prediction of error propagation a very challenging task. Additionally, it is also very challenging, and sometimes almost impossible, to identify the original fault (i.e., the fault that gives rise to the FEF chain) once a failure arises.

The unfeasibility of fault detection is further exacerbated when components live concurrently at runtime and maintain a diversified lifespan. Assigning a limited life cycle to components, also known as *scope*, is a very common practice in software architectures. Scoped components allow easy management of transient information (i.e., session state [19]). The life cycle of the information is bound to the life cycle of the component that encapsulates it. As a result, scope and life cycle management are mechanisms usually provided by third-party frameworks. The life cycle management frameworks usually provide a module, called *container*, that manages the component instances. Frequently, the container in addition to creation and destruction of instances also deals with dependencies management. Specifically, when a component instance requires a reference to a specific type of instance, i.e., a *dependency*, the container at runtime searches for it among the currently living instances. If an eligible instance exists, the container provides the reference to the requiring component. If there are no candidates, the container will provide the reference to a newly created instance initialized for the occasion. This practice is frequently identified as *dependency injection*. It is very popular in software architecture since they promote loose coupling and flexible code [20].

There are plenty of frameworks that provide automatic life cycle management and dependency injection, following the guidelines outlined above. For the sake of concreteness, in this work, we focus on *Contexts and Dependency Injection* (CDI) [2], one of the most popular frameworks used for the development of stateful enterprise architectures.

In CDI, through metadata specifications, each component is associated with a predefined scope that binds its life cycle with a specific amount of HTTP requests [3]. More in detail, in JEE, standard scopes are provided by the CDI framework and are the following:

- *RequestScoped*: the component associated with this scope lasts for the time required to respond to a single HTTP request;
- *SessionScoped*: the component lives for an entire HTTP session;
- *ApplicationScoped*: the component lives for the entire lifetime of the application,
- *ConversationScoped*: the component lives during a single HTTP session for a sequence of HTTP requests explicitly demarcated by the developer.

By considering the introductory example presented in Figure 1, B represents a *RequestScoped* instance since it lives just for the duration of request $R_2$. C is a *ConversationScoped* instance since it lives for the subsequent requests $R_2$, $R_3$ and $R_4$, while A is a *SessionScoped* instance whose lifespan covers the whole session of execution. Finally, while not represented in Figure 1, *ApplicationScoped* corresponds to an instance that spans for the entire time the software-intensive system is operative, i.e., an instance that is created when a software-intensive system is started, and is terminated only when the software-intensive system is terminated.

This business logic setup identifies a scenario where instances are born and eventually die. When a component dies, also any erroneous information carried inside is destroyed. The finite lifespan of components then, enables the possibility to correct the overall state of the application [21]. At the same time, the fault that originates an FEF chain may belong to an instance that no longer exists when the failure manifests, making the fault localization task more challenging.

As a side note, dependency injection containers allow the management of "global" components that are shared among multiple sessions, e.g., ApplicationScoped components in CDI. This makes it possible to develop a transitive dependency between sessions. OREO also handles these corner cases by representing global components in each timeline. To manage the interference of another session on a global component within a timeline, OREO inserts a fictitious instance that interacts with that component.

### 4.2. Fault-Error-Failure Chain Formalization

Based on the timeline abstraction, we now formally define the concept of fault, error, and failure, and how the propagation of the error can give rise to FEF chains.

Let $TL = \langle T, C, J, I, O \rangle$ be a timeline. A fault is activated on an instance $j \in J$, at a specific epoch, due to a specific interaction $i \in I$. From that moment on, the instance is considered an erroneous instance. We identify this event as root error activation $e = (i, j)$. An erroneous instance $j_1 \in J$ has the ability to propagate its error and thus render another instance $j_2 \in J$ erroneous, through an interaction $\langle j_1, j_2, r \rangle$ with $r$ epoch in $R$. A failure $F_j^r$ is an error that propagates to the presentation layer via interaction with an erroneous instance $j \in J$. Such interaction is denoted as $u_e^r = \langle j, PL, r \rangle$.

A FEF Chain in $TL$ is a tuple

$$FEF = \langle T^{FEF}, C^{FEF}, involved^{FEF}, I^{FEF}, O \rangle$$

where:

- $I^{FEF}$ is an ordered list of interactions $\{I_i \in I | I_1, I_2, ...I_n\}$ where $I^{FEF} \subseteq I$ so that for each $I_i = \langle j_1, j_2, R_x \rangle \in I^{FEF}$ exists at least one interaction $I_l =< j_0, j_1, R_y >$ with $l < i$ and $i \neq 1$, i.e., exist a transitive relation between the instance in $I^{FEF}$.
- $T^{FEF}$ is the timespan during which $FEF$ occur. Let $t_1 \in T$ be the time relative to the epoch $r$ where $I_1$ occurs. Let $t_n \in T$ be the time relative to the epoch $r$ where $I_n$ occurs. Then:

$$T^{FEF} = \{t \in T | t_1 \leq t \leq t_n\}$$

- $involved^{FEF} \subseteq j$ is the set of instances involved in the propagation chain:

[2] https://jakarta.ee/specifications/cdi/ Accessed 5th October 2024

[3] https://docs.jboss.org/weld/reference/latest/en-US/html/scopescontexts.html Accessed 5th October 2024

$$involved^{FEF} = \bigcup_{\langle j_1, j_2, \cdot\rangle \in I^{FEF}} \{j_1, j_2\}$$

A root instance of a *FEF*, $root_{FEF} \in involved^{FEF}$ is the instance that initiated the fault propagation chain. Let $e^{FEF} = (i, j)$ the root error activation event of *FEF*, then $root_{FEF} = i$.

- $C^{FEF} \subseteq C$ is the set of all the components contained in the FEF chain. $C^{FEF}$ represents the component types of instances involved in the fault propagation chain. Formally:

$$C^{FEF} = \left\{ c \in C \mid \exists\, j \in involved^{FEF},\ type(j) = c \right\}$$

Where *type(j)* $=$ *c* meaning that the type of the instance j is c.

- $O \subseteq I \times I$ is a partial order on $I$ that specifies the temporal ordering of any two interactions. Given that $I^{FEF} \subseteq I$ and the order of interactions in $FEF$ follows the same ordering criterion as *TL*, *O* is shared between *TL* and *FEF*.

Finally, $U^{TL}_{FEF}$ is the set of all possible $FEF = \langle T^{FEF}, C^{FEF}, involved^{FEF}, I^{FEF}, O\rangle$ where $T^{FEF} \subseteq T$,$C^{FEF} \subseteq C$, $involved^{FEF} \subseteq J$, $I^{FEF} \subseteq I$. Formally:

$$U^{TL}_{FEF} = \left\{ \overline{FEF} \left| \begin{array}{c} T^{\overline{FEF}} \subseteq T, \\ C^{\overline{FEF}} \subseteq C, \\ involved^{\overline{FEF}} \subseteq J, \\ I^{\overline{FEF}} \subseteq I \end{array} \right. \right\}$$

Consider, for example, the timeline shown in Figure 2. Let us denote the instance of type A as a, the instance of type B as b, and the instance of type C as c. Within this timeline, there is a process of fault activation, error propagation, and failure manifestation. This can be represented by an FEF chain, which we will refer to as *exFEF*. The interaction that activates the error in b is the interaction between a and b in $R_2$, $< a, b, R_2 >$. Subsequently, the error propagates from b to c via $< b, c, R_2 >$. Finally, in $R_3$, the failure occurs due to $< c, PL, R_4 >$. The FEF chain *exFEF* is composed as follows:

- $T^{exFEF} = R_2, R_3, R_4$
- $C^{exFEF} = B, C$
- $involved^{exFEF} = b, c, PL$
- $I^{exFEF} = \{< b, c, R_2 >, < c, PL, R_4 >\}$
- *b* as root instance.

## 4.3. Profiling the Business Logic with OREO

The tight relationship between software components and runtime events makes it difficult to statically analyze and predict the actual behavior of the application. In case of system failure, the component that provides the malfunction is not necessarily the component that hides the fault. The system failure could be the result of an FEF chain and the fault could be hidden in a component already dead at the failure instant. Additionally, the FEF chain evolution strictly relies on user behaviour which is performed at runtime and is out of the control of the developer. For this reason, in this work, we integrated into OREO a profiler as a concrete implementation of the *analysis* module represented in Figure 4. The profiler aims to offer insights into the runtime behavior of the system. Analyzing the timelines generated by the *instrumentation* module, the profiler can deduce the possible error propagation scenarios thus supporting both fault analysis and dependability assessment.

Concretely, the profiler offers support for the timeline analysis with three different features.

*OREO Profiling Feature 1: Identification of Safe and Unsafe Instances* To correctly understand this feature, we introduce the concept of safe and unsafe instances. Let us consider a usage scenario executed on a standard software architecture. The scenario is characterized by a sequence of user inputs and by the corresponding response process in the business logic. Let us suppose that, at a certain point, an instance enters an erroneous state. Thus, if there is no chance for an instance to be involved in the FEF chain of the supposed error, we consider the instance safe. Conversely, if there is at least one feasible FEF chain that involves the instance, the instance will be considered unsafe.

Starting from a timeline and supposed an instance as erroneous, the OREO profiler is able to identify which instances are safe and which are unsafe. In the case of real-world applications, the particularly large and intricate business logic makes the identification of safe instances hard, especially in presence of long input sequences. Therefore, the ability to exclude automatically instances from the propagation scenarios constitutes a remarkable boost for the analysis.

Formally, given a timeline $TL = \langle T, C, J, I, O\rangle$ and a root error activation event $e = (i^{err}, j^{err})$ with instance $j^{err} \in J$ and interaction $i^{err} \in I$, an instance $\bar{j} \in J$ is said to be unsafe if there exists at least one FEF with error activation event *e* that involves $\bar{j}$. The *unsafe* set is defined as follows:

$$unsafe(e) = \bigcup_{\substack{fef \in U^{TL}_{FEF} \\ e^{fef} = e}} involved^{fef}$$

With $fef \in U^{TL}_{FEF}$ meaning that *fef* is a feasible FEF in *TL*. From the *unsafe*(*e*) set can also be defined the *safe*(*e*) set as: $safe(e) = C \setminus unsafe(e)$.

By knowing all the interactions performed between instances at runtime, OREO is able to perform a forward analysis of the possible FEF chains on the timeline. This allows OREO to identify the instances that may or may not take part in the propagation of the error. In practice, the number of safe and unsafe instances can help highlight a possibly too strong coupling among instances and suggest a need to redesign the response procedures.

For the sake of concreteness, let us assume the timeline of Figure 3 and suppose that the instance of type $C_4$ living during epochs $[R_2, R_4]$, called for simplicity $c_4$, enters an erroneous state in step $R_2$ due to a root error activation event $e = (i, c_4)$ experienced after the interaction $i$ with the instance of type $C_2$. The OREO profiler will detect the instance of type $C_2$ that lives in steps $[R_6, R_7]$ and the instance of type $c_1$ as safe instances because there is no feasible FEF chain that propagates the error $e$ from $c_4$ to them. Also, since $e$ is activated after the interaction $i$ in $R_2$, the instance of type $C_5$ living in $R_2$ and the instance of type $C_2$ living in $R_1$ and $R_2$ will be classified as safe instances. The remaining instances will be classified as unsafe.

*OREO Profiling Feature 2: Identification of FEF Root Instances* The OREO profiler is also able to extract the instances that may be the root of the underlying FEF chain. We consider an instance as the root of an FEF chain if it is the starting point of the whole propagation process (see Section 4.2). Identifying a restricted subset of possible root instances, considerably facilitate the fault localization process, which could be again particularly complex and time-consuming in real-world scenarios.

Formally, given a Failure $F_j^r$, the OREO profiling feature 2 consists of identifying all possible instances that could lead to the activation of such failure. The output of the feature is the set $roots(F_j^r)$:

$$roots(F_j^r) = \bigcup_{\substack{fef \in U_{FEF}^{TL} \\ F(fef) = F_j^r}} \{root(fef)\}$$

With $F(fef)$ function that returns the failure of *fef*.

Starting from the timeline, OREO conducts a backward analysis of potential FEF chains that may have led to the failure, identifying all possible root causes. In practice, identifying the possible roots starting from a failure means obtaining all possible components within which the original fault that caused the failure is hidden.

Considering the example of Figure 3, and assuming that a failure is manifested in step 10 by component $c_3$. In a standard setting, the developer should start the fault detection process from component $c_3$. Nevertheless, in case of a failure induced by error propagation, $c_3$ will not contain the fault that caused the observed failure. Consequently, once inspected $c_3$, without knowledge about communication among runtime instances, the fault might be difficult to spot through black-box techniques, or white-box techniques which do not consider runtime inter-instance communication. In presence of the timeline describing a scenario where at least one failure occurred, the developer can exploit the OREO profiler to easily identify the subset of components candidate to be the actual faulty component. In the illustrative case depicted in Figure 3, the failure of $c_3$ at step 10 may be caused by a fault hidden into components $c_2$ (the one living during $[R_1, R_2]$), $c_3$, or $c_4$ excluding components $c_1$ and $c_5$.

*OREO Profiling Feature 3: Identification of FEF Scenarios* Finally, the OREO profiler allows studying how a fault hidden in a specific instance may spread throughout the application if the error is activated at a specific time step. In particular, starting from an instance as the root of the hypothetical FEF chain, the profiler lists all the feasible propagation paths.

Formally, given an instance $j \in J$ and an interaction $i \in I$ causing the activation of an error, the OREO profiling feature 3 identifies all possible FEF scenarios that could occur:

$$FEFscenarios(j) = \left\{ fef \in U_{FEF}^{TL} \mid \mathrm{root}_{fef} = j \right\}$$

For instance, let us suppose that in the timeline represented in Figure 3, component $c_4$ activates a fault at step $R_3$. By executing OREO, the tool will automatically determine all possible FEF scenarios with which the fault may have propagated (see Figure 6). This is made possible through OREO as, by knowing the states of all instances and the communications among them, it is able to combinatorially determine in an exhaustive manner which components may have been influenced by the fault during the subsequent requests (from $R_3$ to $R_{10}$ in the example of Figure 6). As a corner case, in the scenario represented in Figure 6a, the error is never propagated, and the correct state is restored after step $R_4$, when $c_4$ ends its life. Conversely, if the error successfully propagates in $R_4$, components retain the effects of an instance that no longer exists. In practice, the OREO profiling feature 3 allows users to perform a what-if analysis and visualize how an error might propagate within a given timeline. This feature can then indicate how the business logic could be susceptible to error propagation.

## 4.4. OREO Expressiveness and Properties

The OREO tool, along with its proposed formalization, enables offline monitoring of the behavior of runtime instances residing in the business logic of a software-intensive system. Specifically, the timeline abstraction extracted through OREO allows to use various specification languages to define properties to monitor both online and offline the system of interest. For example, given a timeline $TL = \langle T, C, J, I, O \rangle$, the partial order identified by $O$ allows to monitoring temporal ordering properties between direct or transitive interactions in $I$. Specifying such properties can be accomplished using temporal logic, regular expressions, or other more expressive variants, e.g., CaRet [22], Eagle [23], or *frequency* Linear-time Temporal Logic [24].

Additionally, the total set $T$ of points in time at which events can occur, and the associated set of discrete epochs $R$ (see Section 3.1), allows for the specification of discrete-time and real-time properties allowing to use, for instance, specification languages like Metric Temporal Logic.

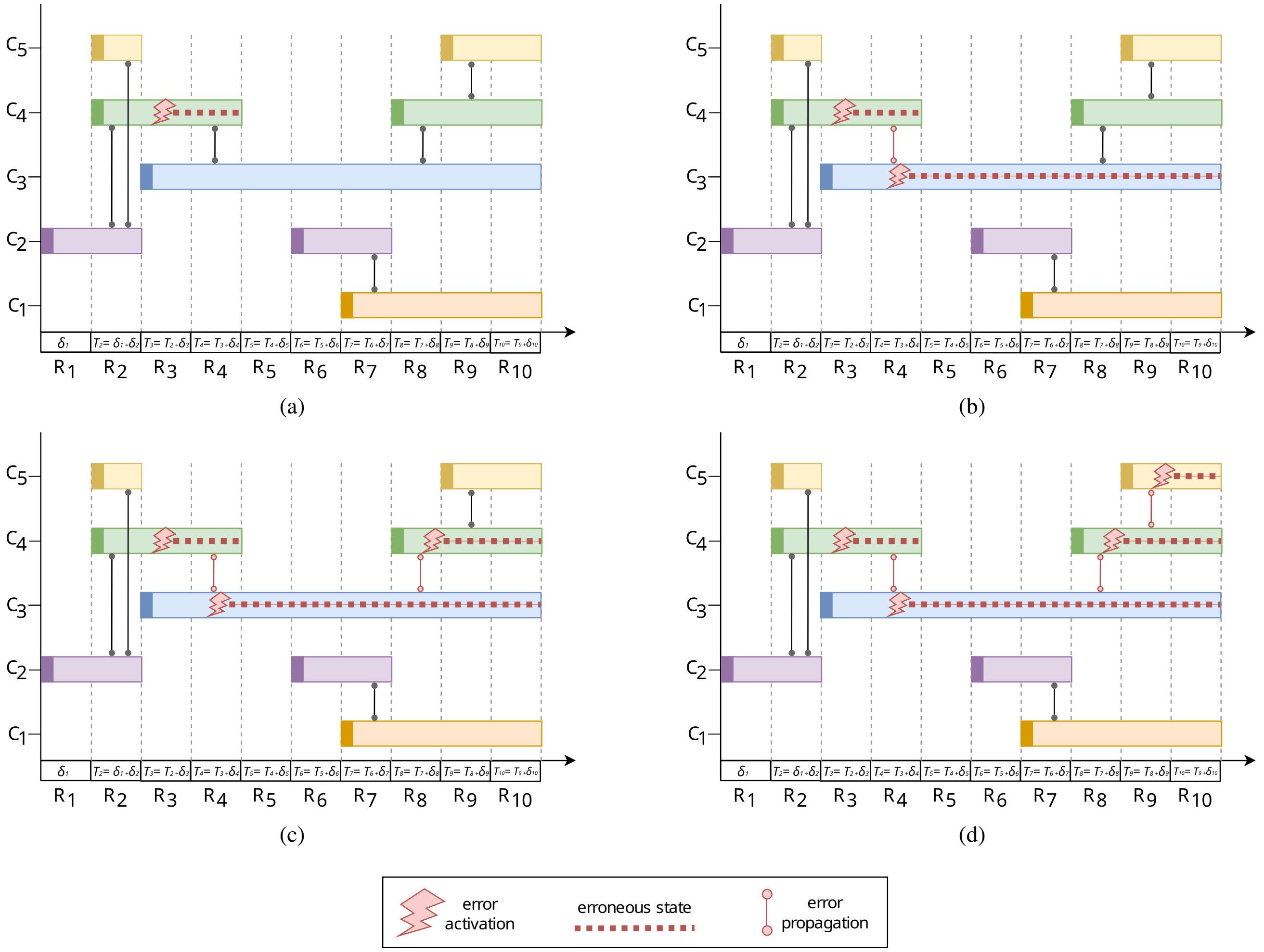

**Figure 6:** Possible FEF chain propagations in case of error activation of instance $c_4$ at step $R_3$ in the usage scenario represented in Fig. 3.

Regarding instead the reliability of OREO, in Section 4.3 we formalized the problem of error propagation on the timeline, by following the definitions of fault, error, and failure identified by Avizenis et al [18] (see also Section 1). On top of the formalization of error propagation, we also described three exemplary profiling features that are provided out of the box in the approach implementation to support the understanding and visualization of different error propagation scenarios. The FEF chain formalization adds expressiveness to the approach making it possible to specify reliabiliy properties on the timeline. For example, it is possible to check if a component ever becomes the root of a failure, or if a component participates in a FEF chain.

In practice, monitors can be implemented by utilizing the timeline representation generated by the OREO tool. In this sense, specifying monitors in OREO is similar to specifying monitors with AspectJ or JavaMOP [25]. The difference is that, once OREO is plugged in, the timeline is automatically extracted without the need to manually define pointcuts, and the verdict can be computed directly from the timeline object.

## 5. OREO Execution Evaluation

To evaluate the OREO tool, we conducted an experimental proof of concept to estimate its applicability. During this experiment, we aimed to assess: (i) the effort required by OREO to instrument the target software-intensive system, (ii) its capability to extract useful insights about potential error propagation phenomena, and (iii) the cost in terms of time and memory overhead that OREO entails.

Our results show that the OREO tool offers a seamless plug-and-play instrumentation process, advanced and extensible profiling capabilities, and minimal overhead in terms of delay and resource utilization.

### 5.1. Research Questions

The evaluation is intended to research the extent to which the theoretical framework of OREO is applicable, and serves as a stepping stone to bridge the purely formal definition of the approach with a hands on practical viability experimentation.

The evaluation is carried out to answer the following research questions (RQs), which guided the design of the evaluation process:

- $RQ_0$: *What is the upfront effort needed to apply the OREO tool?*
- $RQ_1$: *To which extent is OREO able to get insight into faults and error propagations?*
- $RQ_2$: *To what extent is OREO scalable?*

With $RQ_0$, the objective is to assess the extent to which OREO can be applied to software architectures of different sizes, and if the effort required to apply the tool is related to the size of the system under analysis. The term *size* of an application refers to the number of components managed by the container for automatic lifecycle management and dependency injection, as well as the source lines of code (SLOC) and pages. The rationale behind $RQ_0$ is whether the effort for instrumentation increases with the number of potential elements to observe.

With $RQ_1$, we want to show how the OREO profiler can be utilized to execute reliability analyses. Specifically, we investigate how the tool supports fault detection processes, and if the analysis can be used for error propagation identification among business logic components.

With $RQ_2$, we aim to assess to which extent OREO's performance is independent of the size of the application under analysis.

## 5.2. Experimental Objects

To evaluate the effectiveness and applicability of OREO, we selected three distinct software projects as experimental subjects. This selection was guided by three primary criteria: (i) JEE implementation to ensure compatibility with the functional requirements of OREO, (ii) availability of source code to configure OREO, and (iii) diversity in scope and size to test OREO across varied operational scenarios. The three experimental objects considered are:

- `Toy app`: A simple, small-scale application developed in house specifically for this experimentation. The source code is available in the replication package. Functionally, it permits the insertion of numbers among different pages and performs calculations on them, e.g., to verify if the sum of the last two numbers inserted is odd. It consists of a user interface made of 6 main pages, a business logic made of 8 components, 5 of which act as controllers serving page requests, and 3 additional helper components. Toy app is made of 330 source lines of code (SLOC).
- `Books app`: an exemplary, mid-sized, application presented in the book by Muller et al. [26]. `Book app` is an application that manages and lists book reviews. `Book app` comprises 15 pages, 15 components, and a domain model made of 6 entities. `Book app` is made of 2818 SLOC.
- `Empedocle system`: a large-scale real-world software project. `Empedocle` is an Electronic Health Record (EHR) System that allows the management of medical examinations and medical staff. It is a real-world system characterized by a Technology Readiness Level 9 (TRL9) and has been in use since 2011 in a major hospital in the Tuscany region of Italy. The software project was previously utilized in other scientific studies, e.g., in the work of Patara et al. [27] and in the work of Fioravanti et al. [28]. The application comprises 35 pages, 30 DAOs and 35 domain classes supported by a wide internal library of approximately 200 classes. The implementation of the software project is currently closed-source. Therefore, while used for the evaluation as a real-world industrial evaluation subject, we are not able to make the source code public as part of the replication package of this study. `Empedocle` is made of 85718 SLOC.

The size diversity of the experimental objects allowed us to easily study the behavior of OREO in a simple application (`Toy App`), assess its viability in a realistic scenario (`Book App`), and finally, evaluate its capabilities in a large-scale real-world project (`Empedocle`).

## 5.3. Experimental Process

In this section, we outline the experimental procedures employed to investigate the research questions delineated in Section 5.1. All the experiments were conducted entirely on a single laptop equipped with an Intel *i7-8750H* (2.20GHz) CPU and 16 GB 2.666 MHz DDR4 of memory.

### *5.3.1. Upfront Effort Needed to Run the OREO Tool ($RQ_0$)*

To evaluate the effort required to run the OREO tool, we aim to determine if, and to what extent, this effort depends on the static characteristics of the software-intensive system being monitored. Specifically, we are interested in assessing the effort in relation to the SLOC, the number of components to observe, and the number of pages of the target system. To achieve this goal, we rely on the three experimental objects presented in Section 5.2. These objects differ significantly in terms of SLOC, number of components, and number of pages: a small software-intensive system (Toy App), a medium-sized system (BookApp), and a large real-world system (Empedocle).

To address the research question $RQ_0$, we instrumented and subsequently executed OREO on the three experimental objects. More precisely, the timelines for each application were generated by considering the most common use case scenarios covered by the three software projects under analysis. Further details on the use case scenarios, along with a discussion and results of the experimental procedure, are provided in Section 5.4.

### *5.3.2. Profiling the Timelines ($RQ_1$)*

To address research question $RQ_1$, we aim to evaluate the extent to which the OREO tool and its profiling features, as described in Section 4.3, can support the analysis of error propagation within a system. Specifically, we seek to determine how insights that are challenging to deduce from code analysis can be made explicit through OREO and its timeline analysis. The objective of this experiment is not only to demonstrate the practical utility of the profiling features but also to illustrate how OREO can serve as a foundation for a more structured analysis. During the demonstration, we will utilize both the profiling features and the insights gained from their combination.

As shown in the example depicted in Section 2.1, it is challenging to determine, in the event of a failure, which components might have caused the malfunction and which are definitely not involved in the propagation scenario. Therefore, we formalize the concepts of *root candidates*, *unsafe* instances, and *currently unsafe* instances. In particular, assuming the failure $F_r^j$ at epoch $r$ manifested by an instance $j \in J$ of a certain component type $c \in C$, we identify the following information:

- *Root Candidates*: this identifies the instances that could be the root of the FEF chain leading to the manifestation of the considered failure $F_r^j$. Formally, it consists of the cardinality of set $roots(F_r^j)$, obtained using OREO Profiling Feature 2. Conceptually, it provides a measure of how complicated it is to identify the actual component responsible for the failure.

- *Unsafe*: this identifies all instances, both those living at the time of the failure and those no longer available, that could be involved in the FEF chain that led to the manifestation of the considered failure $F_r^j$. Formally, the *Unsafe* column consists of the cardinality of the set:

  $$\bigcup_{j \in roots(F_r^j)} unsafe(j, i)$$

  The *Unsafe* instances set is derived by integrating OREO Profiling Feature 1 and OREO Profiling Feature 2.

- *Currently Unsafe*: this identifies the instances, which could be involved in the FEF chain leading to the manifestation of the considered failure $F_r^j$, living at the same epoch $r \in R$ as $F_r^j$.

  Formally, the *Currently Unsafe* column consists of the cardinality of set:

  $$\{ j \in \bigcup_{j \in roots(F_r^j)} unsafe(j, i) | r \in l_j \}$$

  Where $l_j$ is the interval of epochs in which j lives.

  The *Currently Unsafe* instances set is derived by integrating OREO Profiling Feature 1 and OREO Profiling Feature 2.

The results obtained and the related discussion are outlined in Section 5.5.

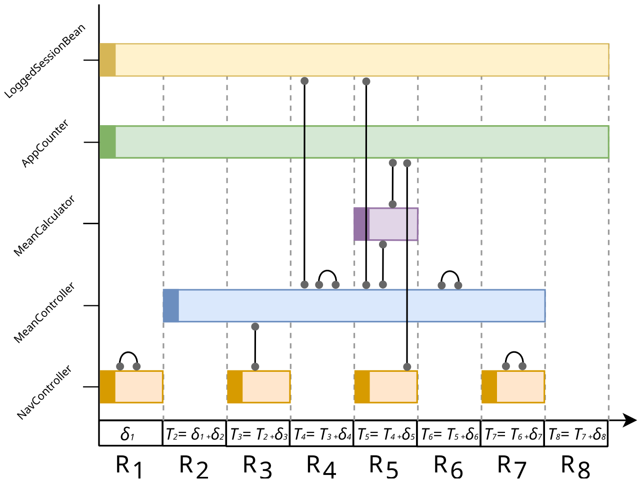

**Figure 7:** Timeline instance extracted from Toy App.

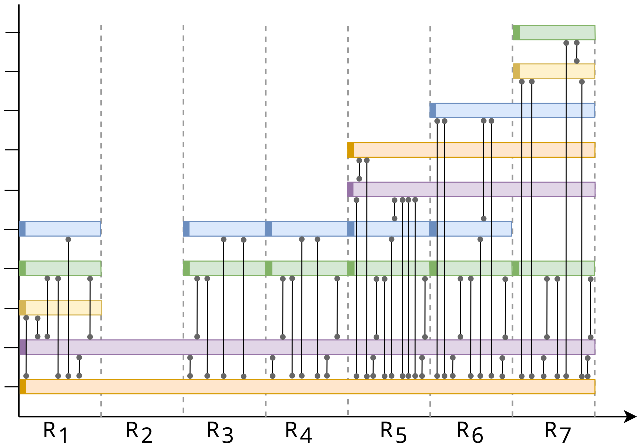

**Figure 8:** Timeline instance extracted from Books App.

### 5.3.3. Scalability of the OREO Tool ($RQ_2$)

To answer $RQ_2$, we investigate if, and to what extent, the user experience could be compromised by the execution of OREO in software architectures characterized by different sizes.

The experiment was conducted on the three experimental objects in order to measure how the tool behaves in different-sized software subjects. In more detail, for each of the three applications we have executed 100 requests and measured the impact that OREO induced at execution time during the response to each request. The requests measured in the experimentation were selected by executing representative use cases of the application under examination (see also internal threats to validity in Section 5.7.2 for this point). This approach allows for obtaining a comprehensive overview of the potential requests that could be made on the application.

We focus on both time overhead and memory overhead. To measure the time overhead, we selectively recorded the operation time of OREO during each request. For memory overhead, we repeated the same 100 requests in the exact same order twice for each experimental subject: once with

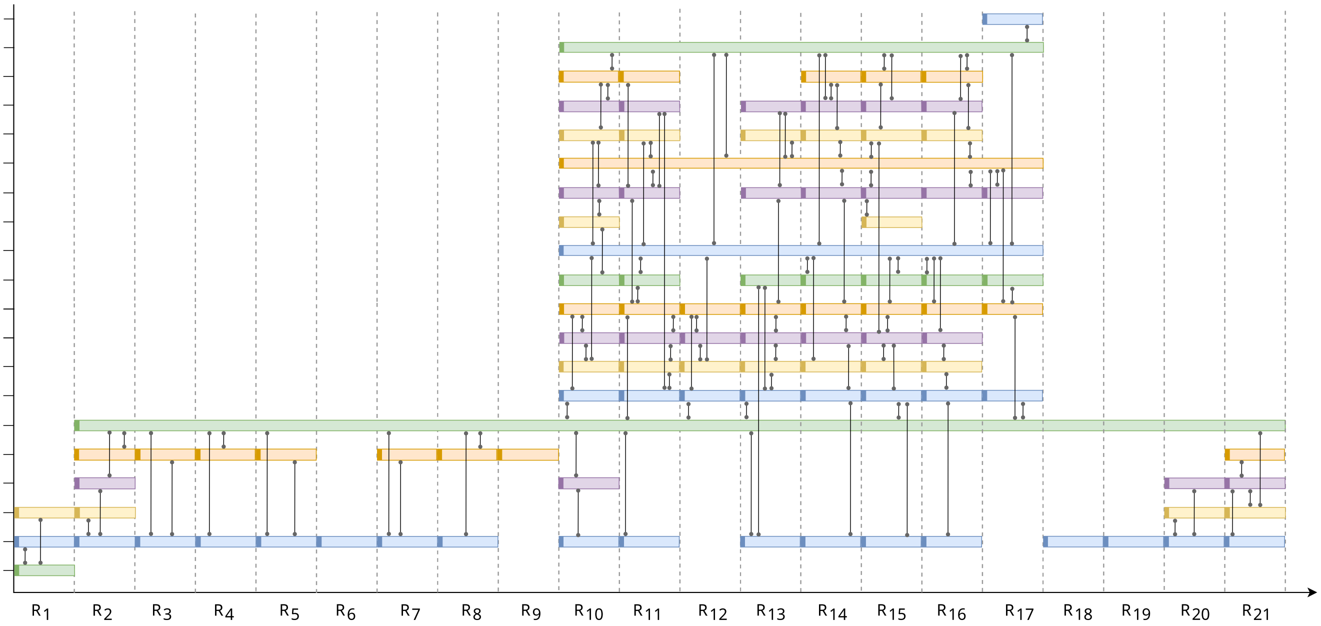

**Figure 9:** Timeline instance extracted from `Empedocle System`.

the OREO tool and once without it. Results and discussion are reported in Section 5.6.

## 5.4. Results of Instrumentation Effort Evaluation ($RQ_0$)

To address research question $RQ_0$, as outlined in Section 5.3.1, we instrumented and executed OREO on top of the three considered applications to measure the applicability and effort required to run the tool. Figures 7, 8 and 9 represent examples of timelines extracted during the utilization of the `Toy app`, the `Book app` and the `Empedocle system`, respectively. The resulting timelines were obtained following the primarily standard uses of the applications. To showcase the functioning of OREO, for `Toy app` we chose as concrete example a timeline capturing the execution of the standard use case scenario where the application checks the disparity of the sum of two variables provided in input by the user. This exemplary scenario execution visualized through OREO is depicted in Figure 7. For the `Books app` instead, we consider as concrete example a use case scenario where the user searches for a specific book and afterward reads the reviews of the chosen book. This illustrative scenario execution visualized through OREO is depicted in Figure 8. Finally, for the `Empedocle` system, we selected as execution trace the once corresponding to the login of an authorized user, followed by the examination of a medical record. The execution of this latter scenario visualized through OREO, is documented in Figure 9.

The represented timelines demonstrate the ability of OREO to observe and extract the evolution of the business logic in all three cases, regardless of the size of the application and the number of components living simultaneously. An exemplary case is represented by the timeline of the Empedocle System in Figure 9. The timeline represented was originally too cumbersome to be represented graphically due to the high number of both components and methods. To ease the interpretability of the figure, the depicted scenario does not include DAOs, converters, and context components. A link between two components represents in this case the existence of one or more interactions.

The experimental results demonstrate, in response to $RQ_0$, that the effort needed to operate OREO remains constant and minimal, regardless of the number of SLOC, components, or pages in the target application. Therefore, running the tool requires an initial configuration phase that is independent of the size of the target application. More precisely, the basic configuration of OREO requires only specifying OREO as the CDI extension of the target application. The procedure can be deemed as rather straightforward, as it consists only in copying a single plain file inside the metadata directory of the target application. Nevertheless, it is worth mentioning that with the basic configuration, OREO observes all the actions and procedures in the business logic, even those concerning components belonging to other libraries and processes over which the developer has no control. The responsibility to specify a narrower group of components to observe is left to the OREO user. The selective observation of the business logic can be configured with ease from the OREO settings using a list of strings or regular expressions.

From the data collected to answer $RQ_0$, we evince that using the OREO tool with a JEE application is trivial. The effort required to run the tool, even considering possible custom configurations, is minimal and does not depend on the size of the target software architecture.

| Step | Failed ComponentType | Scope | Instances | | | |
|---|---|---|---|---|---|---|
| | | | Root Candidates | Unsafe (%) | Currently Unsafe (%) | Currently Safe (%) |
| 1 | *loggedSessionBean* | *@SessionScoped* | 1 | 33.33 | 33.33 | 66.66 |
| | *appCounter* | *@ApplicationScoped* | 1 | 33.33 | 33.33 | 66.66 |
| | *navController* | *@RequestScoped* | 1 | 33.33 | 33.33 | 66.66 |
| 2 | *loggedSessionBean* | *@SessionScoped* | 1 | 25 | 33.33 | 66.66 |
| | *appCounter* | *@ApplicationScoped* | 1 | 25 | 33.33 | 66.66 |
| | *meanController* | *@ConversationScoped* | 1 | 25 | 33.33 | 66.66 |
| 3 | *loggedSessionBean* | *@SessionScoped* | 1 | 20 | 25 | 75 |
| | *appCounter* | *@ApplicationScoped* | 1 | 20 | 25 | 75 |
| | *meanController* | *@ConversationScoped* | 2 | 40 | 50 | 50 |
| | *navController* | *@RequestScoped* | 2 | 40 | 50 | 50 |
| 4 | *loggedSessionBean* | *@SessionScoped* | 3 | 60 | 66.66 | 33.33 |
| | *appCounter* | *@ApplicationScoped* | 1 | 20 | 33.33 | 66.66 |
| | *meanController* | *@ConversationScoped* | 3 | 60 | 66.66 | 33.33 |
| 5 | *loggedSessionBean* | *@SessionScoped* | 3 | 85.71 | 100 | 0 |
| | *appCounter* | *@ApplicationScoped* | 6 | 85.71 | 100 | 0 |
| | *meanCalculator* | *@RequestScoped* | 5 | 85.71 | 100 | 0 |
| | *meanController* | *@ConversationScoped* | 4 | 85.71 | 100 | 0 |
| | *navController* | *@RequestScoped* | 6 | 85.71 | 100 | 0 |
| 6 | *loggedSessionBean* | *@SessionScoped* | 3 | 85.71 | 100 | 0 |
| | *appCounter* | *@ApplicationScoped* | 6 | 85.71 | 100 | 0 |
| | *meanController* | *@ConversationScoped* | 4 | 85.71 | 100 | 0 |
| 7 | *loggedSessionBean* | *@SessionScoped* | 3 | 75 | 75 | 25 |
| | *appCounter* | *@ApplicationScoped* | 6 | 75 | 75 | 25 |
| | *meanController* | *@ConversationScoped* | 4 | 75 | 75 | 25 |
| | *navController* | *@RequestScoped* | 1 | 12.5 | 25 | 75 |
| 8 | *loggedSessionBean* | *@SessionScoped* | 3 | 75 | 100 | 0 |
| | *appCounter* | *@ApplicationScoped* | 6 | 75 | 100 | 0 |

**Table 1**
Step-wise failure manifestation analysis.

> **$RQ_0$ Takeaways (Instrumentation Effort)**
> 💡 **Takeaway 0.1: The effort required to apply the OREO tool is independent of the target software system.**
> 💡 **Takeaway 0.2: The instrumentation procedure is minimal and straightforward.**
> 💡 **Takeaway 0.3: Custom configurations that modify the default behavior of OREO remain minimal and are easy to configure.**

## 5.5. Results of OREO Profiling Capabilities ($RQ_1$)

According to Section 5.3.2, to answer research question $RQ_1$, we applied OREO to a concrete example in order to provide insights regarding the behavior of the tool and its practical application. The results and an accompanying discussion are provided by considering the timeline of Figure 7 extracted during the execution of `Toy app` and already presented in Section 5.4.

Table 1 shows the results of the OREO tool analysis, varying both the steps and the components in which the failure is hypothetically manifested during the execution of the scenario represented in Figure 7. In more detail, starting from the extracted timeline of Figure 7, we considered all the possible failure scenarios; i.e., all the possible failures $F_j^r$ with $r \in R$ and $j \in J$. For each failure scenario, we exploited the OREO profiling features 1 and 2 outlined in Section 4.3 to gain insight about the state of both the instances living at the time of the failure manifestation and the past instances no longer present in memory.

In particular, assuming a failure $F_r^j$ at step $r$ manifested by an instance $j \in j$ of a certain component type $c \in c$, namely *failed component type* in the table, we identify the number of *root candidates*, the percentage of *unsafe*, *currently unsafe* and *currently safe* instances for each failed component type at each step of the timeline.

Table 1 is a demonstrative table of the capabilities of the OREO profiler and summarizes only some of the information that can be obtained with the proposed tool. OREO is able not only to calculate the number or percentage of involved instances but also to identify the specific instances. For example, it is able to identify not only the number but also the specific instances that are candidates to be the root instance for a FEF chain. The decision to report only aggregated data such as percentages is due to space constraints.

Table 2 shows the number of possible FEF chain paths that a fault activated at a specific time step into a specific component may generate. For each set of paths discovered,

| ComponentType | Scope | TimeSteps | #Paths | Total Errors Worst Case |
|---|---|---|---|---|
| *loggedSessionBean* | *Session* | [1, 4] | 13 | 5 |
| | | [5] | 5 | 5 |
| | | [6, 8] | 1 | 1 |
| *AppCounter* | *Session* | [1, 5] | 4 | 3 |
| | | [6, 8] | 1 | 1 |
| *MeanCalculator* | *Request* | [5] | 6 | 4 |
| *MeanController* | *Conversation* | [2, 3] | 32 | 6 |
| | | [4] | 16 | 5 |
| | | [5] | 8 | 5 |
| | | [6, 7] | 1 | 1 |
| *NavController* | *Request* | [1] | 1 | 1 |
| | | [3] | 17 | 6 |
| | | [5] | 2 | 2 |
| | | [7] | 1 | 1 |

**Table 2**
Step-wise error activation analysis.

the total number of erroneous instances is reported, assuming the worst-case propagation condition, i.e., when errors completely follow the cascade propagation.

Results demonstrate that the potential fragility of the business logic, i.e., the number of root candidates, and the percentage of unsafe / currently unsafe instances, is related to the sequence of interactions between components. In addition, results demonstrate that the scope of the components plays an important role in the propagation paths, since it defines the lifespan of the instance and then indirectly the time that an error can last in the business logic. For the same reason, also the interactions have a significant impact since the error survives beyond the life of its component. OREO is able to identify these aspects and highlight potential vulnerabilities that manifest in specific timelines.

From the experiment conducted in this section, addressing $RQ_1$, we observed that the OREO tool is capable of conducting a comprehensive reliability analysis of the business logic of an application. OREO allows to identify the subset of possible root instances automatically, lightening up the combinatorial space and consequently the fault detection processes. In addition, the tool provides valuable insights into how the business logic may be affected by an error propagating among components. Specifically, OREO identifies successfully safe and unsafe instances at a specific step and lists all the possible FEF scenarios.

> **$\text{RQ}_1$ Takeaways (Profiling Capabilities)**
> **💡 Takeaway 1.1: OREO provides valuable insights into the state of the system starting from failure manifestations.**
> **💡 Takeaway 1.2: OREO also enables what-if analysis to study hypothetical error propagation scenarios over a timeline.**
> **💡 Takeaway 1.3: OREO allows for the combination and extension of the proposed profiling features to obtain structured information and insights.**

### 5.6. Results of OREO Scalability Evaluation ($RQ_2$)

As described in Section 5.3.3, to address research question $RQ_2$, we measured the impact of the OREO tool on the three experimental objects. Table 3 shows the results of the experiments. Specifically, for each application, the number of source lines of code (*SLOC*) and their overhead are reported. The total time overhead expressed as a percentage is available in the "*Time Overhead*" column. In addition, in the "*Instances Overhead*" and "*Methods Overhead*" columns, we report the mean overhead and related standard deviation of the two steps by which OREO operates on each request. The role and significance of these two steps will be explained shortly. By summing the mean of the two columns, it is possible to obtain the overall time overhead in terms of milliseconds.

Looking the results, at first glance, it might seem that the time overhead in milliseconds strictly depends on the size of the application under observation, i.e., the number of SLOC. However, a deeper understanding of the OREO observation process shows that the overhead actually depends on the complexity of the individual response processes. With complexity we denote the combination of the internal states of the software under analysis when a request arrives, i.e., the number of living instances (column "*Instances per Request*"), and the specific process that the response implies, i.e., the number of methods called (column "*Methods per Request*").

More in detail, OREO observes the currently living instances at the beginning and the end of the response process in order to derive newly instantiated and destroyed instances. The overhead of this step is indicated by the "*Instances Overhead*" column. This procedure in principle depends on the number of living components observed during the response process. However, experiments highlight that this phase represents an overhead that varies from 0.1 to a maximum of 0.9 ms independently of the number of living instances.

In addition to this baseline time overhead, OREO is also triggered every time a method is called to register both the caller and the callee component. This introduces a delay that is strongly related to the number of methods invoked within the specific response process. The time overhead of this step is indicated by the "*Methods Overhead*" column. Concretely, this results in a minimum delay of 0.1 ms observed with only one method involved, and a maximum of 49.2 ms observed for 383 methods called in a single request.

The dependency of the time overhead on the complexity of each specific response process represents affordable cost in terms of scalability and performance. As can be also observed in the three applications considered, the complexity tends to remain treatable as the SLOC metric grows. The scalability and the lightweight nature of OREO are further confirmed by the percentage time overhead, which remains below 7% for all three applications.

Regarding the overhead in terms of memory usage introduced by OREO, in Table 3, we have included the mean and standard deviation of memory overhead expressed in megabytes ("*Memory Overhead (MB)*"), as well as the overhead expressed as a percentage ("*Memory Overhead (%)*"). As can be seen, the memory overhead for `Book app` is reported

| Application | SLOC | Instances per Request $\mu \pm \sigma$ | Instances Overhead $\mu \pm \sigma$ (ms) | Methods per Request $\mu \pm \sigma$ | Methods Overhead $\mu \pm \sigma$ (ms) | Time Overhead (%) | Memory Overhead (MB) | Memory Overhead (%) |
|---|---|---|---|---|---|---|---|---|
| *Toy App* | 330 | $3.49 \pm 0.69$ | $0.16 \pm 0.11$ | $3.43 \pm 3.04$ | $0.43 \pm 0.38$ | 6.51 | $0.85 \pm 3.13$ | 7.97 |
| *Book App* | 2818 | $2.76 \pm 0.58$ | $0.16 \pm 0.10$ | $34.34 \pm 37.27$ | $4.34 \pm 4.70$ | 6.96 | $-18.41 \pm 39.35$ | $-34.46$ |
| *Empedocle System* | 85718 | $8.12 \pm 1.89$ | $0.812 \pm 0.33$ | $127.33 \pm 97.73$ | $16.14 \pm 12.29$ | 6.08 | $31.64 \pm 36.07$ | 16.62 |

**Table 3**
OREO overhead metrics in the three applications considered, with results obtained from 100 requests per application.

as negative. This counterintuitive result does not demonstrate that OREO leads to better utilization of RAM. Instead, it is the combined outcome of OREO's internal functioning and Java's Garbage Collector. As previously mentioned in this section, OREO observes the currently living instances twice for each request: once at the beginning and once at the end. During each observation, OREO creates as many objects as there are currently living instances. These objects are then used to build the current step of the timeline and are not used beyond that point. The observations made by OREO thus, increase the rate at which unreferenced objects are allocated in heap memory, leading to more frequent garbage collector activations. The increased frequency of garbage collector activation accounts for the lower average memory overhead in the `Books app`. However, the variation between the highest peak observed during executions with and without OREO never surpasses 15 percent. This reinforces the notion that OREO maintains its efficiency and lightweight nature, even from a memory perspective.

In conclusion, the experiments conducted in this section demonstrate that OREO introduces negligible overhead to the user experience. In response to $RQ_2$, our results demonstrate that the OREO tool is scalable, introducing minimal overhead across all analyzed applications, regardless of their size. The measured overhead would be further reduced when we take into account that all the documented experiments were conducted on a personal computer. If these experiments were to be performed on high-capacity servers instead of a personal laptop, we anticipate that the results would demonstrate an even greater enhancement.

> **RQ$_2$ Takeaways (Scalability)**
> **💡 Takeaway 2.1: OREO introduces a negligible response overhead for the user experience.**
> **💡 Takeaway 2.2: The overhead does not depend on the size (SLOC) of the application under monitoring but only on the complexity of the individual response processes.**
> **💡 Takeaway 2.3: The memory overhead caused by OREO remains negligible.**

## 5.7. Threats to Validity

In this section, we discuss the most relevant threats to validity that characterised the evaluation of OREO. The threats to validity follow the classification of Wohlin et al. [29], complemented by reliability considerations [30]. As suggested in recent literature, albeit presented towards the end of the evaluation section, threats were considered from the earliest stages of the experimental research design [31].

### *5.7.1. Conclusion Validity*

In order to mitigate potential threats to conclusion validity, only elemental metrics (e.g., number of failures) and data analyses (e.g., percentages) were utilized to assess the viability of the approach. This limited potential experimental threats related to confounding factors originating from measure reliability or statistical result analyses (e.g., violated statistical test assumptions). Random result irrelevancies introduced due to the use of a specific experimental subject were mitigated by considering three different applications, characterized by heterogeneous size, context, and provenance. To further mitigate unknown variables that may have influenced the results, the collected data and their trends was *post hoc* scrutinized, to ensure that evident conclusion pitfalls in the collected data, such as anomalies and outlier values, were carefully understood and motivated.

### *5.7.2. Internal Validity*

Threats to internal validity may lie in the suboptimal representativeness of the use case scenarios used to conduct the failure manifestation analysis (see also Section 5.3.2 and Section 5.5). To mitigate potential threats of such nature, we applied OREO on three application of different nature, by conducting a tradeoff between both size versus project familiarity, and internal versus external validity. Therefore, the experimentation ensured that, at the cost of loosing generalizability (see also Section 5.7.2), researchers possessed sufficient knowledge to select representative use case scenarios for the application under analysis. Another potential threat to internal validity is constituted by potentially unknown historical threats, i.e., that an experimental execution could have influenced a subsequent one by leaving instantiated objects alive in volatile memory between two executions, therefore influencing future results. To mitigate this threat, all processes related to an experimental run was terminated before a subsequent one took place.

Additionally, the construction of the OREO tool itself could pose a threat to internal validity. Specifically, there might be bugs within the implementation that compromise the extraction and analysis of timelines. To mitigate this threat, we relied on robust and official technologies for information extraction and session management. Specifically, we used Weld, the main reference implementation of CDI [4], and implemented OREO as an extension of CDI through the official Service Provider Interface (SPI) [5]. This approach allowed us to leverage many built-in, tested, and reliable functionalities and information extraction mechanisms.

### 5.7.3. Construct Validity

To mitigate potential threats related to the representatives of the theoretical construct investigated, the experimentation was designed prior the experimental object selection, and experimental objects were not modified in any way. One of the experimental objects included a real-life large-scale industrial project that, albeit the experimentation focused on the viability assessment of the approach, could be deemed representative of a concrete instance on which OREO could be applicable in practice. Construct validity threats related to the definition of experimental artefacts (e.g., step-wise failures and errors) were mitigated by adopting widely adopted and *de facto* standard definitions in software testing [32]. As threat related specifically to $RQ_0$, regarding the effort required to apply OREO, we note that the tool was applied to the experimental objects by a researcher who was already familiar with the tool. Therefore, while the application of the tool started from a clean slate, other experimental subjects, e.g., users inexperienced with the tool may encounter additional difficulties to apply the tool in practice. However, given that executing the tool consists of five atomic and well-documented steps in the companion package, we do not deem this threat as considerably influencing the results collected for $RQ_0$.

### 5.7.4. External Validity

The experimental results reported in this study must be interpreted in light of some external validity threats. While applications of different context, size, and provenance were considered, we do not claim complete generalizability of the results. As a first prominent threat to external validity, the entirety of the applications considered rely on JEE). As such, with the results provided, we do not claim the extensibility of the experimental findings to applications implemented with other technologies. Future research should be conducted to assess if, based on the positive results presented, the theoretical framework on which OREO relies can be applied to programs implemented with other technologies. Similarly, while the presented results may apply to applications of similar size, nature, and context, and one of the experimental objects consisted of a large-scale real-life industrial project, additional experimentation should be conducted to further strengthen the generalizability of the results.

While the provided OREO implementation enables seamless integration with Java/Jakarta Enterprise Edition systems while maintaining flexibility and scalability, it can also be adapted to alternative technologies. The timeline abstraction is inherently language-agnostic and independent from the specific component management framework used (e.g., Java/Jakarta EE or Spring Dependency Injection). Rather than incorporating Java or JEE specific constructs, it models runtime side effects produced by such constructs. For example, instead of including component scope annotations (e.g., `@SessionScoped` or `@RequestScoped`), it represents the runtime lifespan behaviors these annotations induce. Similarly, it excludes CDI dependency injection annotations and instead captures concrete runtime dependencies between objects via method invocations. The generic nature of the timeline allows abstraction not only from component management frameworks but also from the programming language used to implement the system. As a result, the timeline represents parallel execution traces within a software system, regardless of its implementation language (e.g., Java, Python, or C#). Since the timeline is technology-agnostic, its representation and associated profiler (the `timeline` and `profiler` packages in Figure 5), though implemented in Java, can be used as-is in other systems without sacrificing functionality.

The only implementation-specific aspect of OREO is the strategy used to extract the information for the timeline. This dependency on the technology used by the system is inherently unavoidable. However, the design of OREO minimizes the effort required to integrate the framework with other technologies. For Java-based systems that use frameworks other than JEE (e.g., Spring), the integration process would require just an additional step after instrumentation: incorporating JEE dependencies into the system. While this approach ensures flexibility, the additional overhead has not been evaluated, and OREO may perform less optimally than indicated by the results presented in this work. The potential performance degradation stems from the continued reliance on the JEE framework for information extraction. To achieve optimized information extraction, it would be advisable to adapt these mechanisms to the specific framework in use. Specifically, this would involve overriding the two core extraction methods: *i)* `manageMethodCall()` in the `MethodCallInterceptor` class, for extracting the invoked methods; *ii)* `retrieveContextualInstances()` in the `InstanceFinder` class, for extracting the currently active software objects.

If the system is implemented in a different programming language, most of the OREO code can still be utilized even for the timeline construction. Specifically, it will be necessary to implement a process to extract from the system the active methods and objects during each request. Once the information has been extracted from the system, it will be sufficient to invoke the core extraction methods mentioned above.

### 5.7.5. Reliability

To empower the independent scrutiny and reproduction of the reported results by other researchers, both the OREO tool and the experimental objects utilized (with exclusion of the `Empedocle System` due to non-disclosure agreement) is made online as a companion package of this study (see Section 1).

[4] `https://weld.cdi-spec.org/` Accessed 6th February

[5] `https://docs.jboss.org/weld/reference/latest/en-US/html/ri-spi.html` Accessed 6th February

## 6. OREO Usage Scenario

The ease of use of OREO, combined with its lightweight nature (see Section 5 for further details), enables its application across a wide range of scenarios. A fundamental use case involves employing OREO as a validation tool for a system under development. Many of the mechanisms that OREO is capable of making explicit are not easily observable through source code analysis alone. The availability of a tool that provides a runtime description of the behavior of the system along with reliability metrics facilitates analysis and assists in identifying unexpected behaviors and design deficiencies prior to the deployment in a production environment.

Due to its plug-and-play nature, OREO enables the instrumentation of software-intensive systems without requiring prior knowledge of their implementation details. In such cases, OREO can be utilized as a tool to visualize and understand the runtime behavior of even highly complex software-intensive systems without necessitating an understanding of their internal structure or requiring modifications to the original source code.

Furthermore, the lightweight nature of OREO, coupled with its ability to extract meaningful insights for the analysis of Fault-Error-Failure chains, enables its application as a continuous monitoring tool of software-intensive systems in production. This scenario enhances the long-term understanding of system behavior. Additionally, in the event of a failure, OREO allows for the reconstruction and analysis of all potential error propagation scenarios, thereby supporting failure reproducibility, detection, and the removal of rare and difficult-to-trace faults, such as heisenbugs.

## 7. Envisioned applications and extension for OREO Tool

In this paper, we present a novel timeline abstraction that captures the evolution of the business logic of a software architecture. Additionally, a runtime verification framework that extracts and analyzes timeline instances was proposed. The framework extracts at runtime the timelines from the target application, enabling the analysis of the runtime behaviour of the business logic of the application. Finally, a concrete implementation of the framework for JEE architectures, namely OREO, is provided in the form of an open-source tool. OREO includes a profiler module enabling offline runtime monitoring strategies for fault detection and error propagation analysis of the business logic. The implementation of OREO provided with this research was designed be applied with minimal effort to a Java/JEE software-intensive systems (see also Section 1). However, we conjecture that the theoretical foundation OREO relies upon can be applied also to software-intensive systems implemented with different object-oriented languages, e.g., Kotlin and C#. As future work, we deem it interesting to evaluate the extent to which the approach can be applied, or needs to be adapted, in order to work in different context w.r.t. the one used to evaluate the viability of the approach in this research (see also Section 5).

As hinted to in Section 3, the duration of a component life cycle and the interactions performed with other components may represent a potential threat to reliability. A high number of interactions exposes the component to the error propagation phenomenon. A component that lives extensively, has more chance to enter into an erroneous state. Conversely, a component that lives for a restricted period of time has more chance to remain correct. Additionally, if the component enters an erroneous state, is likely to be destroyed before the error propagates. The timeline abstraction proposed captures explicitly both the life cycle and the number of interactions of every single component during the execution. A relevant application of OREO then, may be represented by an offline runtime monitoring strategy aimed to detect possible weak configurations in this regard and consequently suggest modifications to the business logic in order to minimize the menace [21].

Besides, another challenge that a developer may incur during the design of the business logic is to predict the memory occupation. In this sense, the timeline abstraction can provide valuable support to monitor the behavior of the memory, exhibiting both average and peaks of memory usage, and its recurrent patterns.

Last but not least, the theoretical framework OREO relies upon (see Section 4) enables, from a conceptual standpoint, also to implement online runtime monitoring strategies. Specifically, the timeline may be analyzed while the application under observation executes. As a first step, the functionalities implemented in the profiler module may be exploited in an online fashion. Features 1 and 2 of the profiler for instance (see Section 4), if used at runtime would enable proactive strategies like fault detection, isolation, and recovery (FDIR) of business logic components.

Further online runtime monitoring strategies may be implemented from scratch. For instance, since components live in memory, there may be cases where the number of living components is high and in turn, there is excessive memory consumption. The timeline abstraction promotes awareness of the number of living components and additionally allows the implementation of strategies of passivation (i.e., saving the component to temporary storage outside the memory). In principle, the passivation policies can rely on the structural characteristics of currently living components e.g., passivating the component with the biggest memory footprint. However, the nature of the timeline also enables passivation policies based on historic data e.g., passivate the least recently used component.

## 8. Related Work

To compare the contribution of this work with other proposed methodologies, we identified a set of studies and tools that most closely align with our objectives.

Overall, differently from previous approaches, the proposed open-source OREO tool presents, for the first time,

| Subject | Automatic Configuration | Negligible Extraction Overhead | Error propagation Aware | Behavioral Explanation | Parallel Session Support | Main Differences |
|---|---|---|---|---|---|---|
| Logging Tools | ✗ | ✗ | ✗ | ✓ | ✗ | Direct instrumentation of the code, traces with interleaved events and no FEF analysis |
| Du et al. [33] | ✗ | ✗ | ✓ | ✗ | ✓ | No trace extraction framework provided. |
| Jia et al. [34] | ✗ | ✗ | ✓ | ✓ | ✓ | Top-level services observed rather than individual components and methods. |
| Mertz et al. [35] | ✗ | ✓ | ✗ | ✓ | ✗ | Efficient but selective and incomplete trace extraction. |
| Mertz et al. [15] | ✗ | ✓ | ✗ | ✓ | ✗ | Adaptive sampling rate. It does not guarantee the observation of all events of interest. |
| Kong et al. [1] | ✗ | ✓ | ✗ | ✓ | ✗ | Trace extraction configuration can be complex and time-consuming. It does not handle interleaved events. |
| OREO Tool (this paper) | ✓ | ✓ | ✓ | ✓ | ✓ | Complete monitoring setup and FEF propoagation analysis |

**Table 4**
Comparison between OREO and closest methods and tools.

a novel abstraction of business logic evolution behavior (referred to as *timeline*), able to track the lifespan of components and dependencies established among them, during the running behavior of the software observed, to locate and monitors FEF chains. In fact, due to the high number of components and combinatorial user interaction possibilities involved, this is a very challenging task not explored in these terms in the previous works addressing this topic. To the best of our knowledge, the proposed framework and its concrete implementation are the first approaches that propose a runtime verification technique in the business logic layer.

Table 4 summarizes the key differences between the characteristics of our work and the most related contributions in terms of functionality and goals. In the following sections, we discuss the works identified in the table along with other related work by grouping them into categories.

## 8.1. Logging Tools

One of the most widely used methods for extracting and understanding runtime behavior information in software-intensive systems is through logging tools (e.g., log4j, slf4j, or AspectJ). However, as confirmed by He et al. [13], logging tools, unlike OREO, usually entail costly instrumentation of the code, which typically requires modifying the source code of the target system and results in performance overhead (see Table 4). Additionally, such tools inherently lack native support for fault-error-failure propagation analysis of any kind.

In the case of multiple parallel sessions on the system, OREO extracts and analyzes the session traces independently (Section 3). In contrast, standard logging tools extract a single trace with interleaved events, resulting in complexity in understanding and analyzing the system. Some works, such as those by Du et al. [33] and Jia et al. [34] (see Table 4), address the problem of interleaved events; however, they rely on classical logging tools for trace extraction, incurring overhead and instrumentation costs.

## 8.2. Trace Extraction Methodologies

The works of Mertz et al. [35, 15] (see Table 4) propose methodologies for extracting execution traces that, similar to OREO, are lightweight and low-overhead. However, the configuration cost of these methodologies remains uncertain, and to achieve low overhead, they rely on selective event extraction. This reliance results in incomplete execution traces, rendering such methodologies unsuitable for studying fault-error-failure chains.

Efficient online runtime verification for large-scale cyber-physical systems is presented in the work of Zheng et al. [36]. In order to improve the efficiency of the proposed method, a novel linear optimization model is exploited and integrated with a runtime load balancing policy. The approach aimed at guaranteeing bounded computational and memory resources while performing runtime monitoring of local properties. Differently, event transformation algorithms, able to derive essential events from the observed traces, are designed to monitor global properties, achieving efficient monitoring of global properties and formulating the complex distributed monitoring as a standard decision problem for testing the membership of a trace in a regular language [36]. The work of Zheng et al. proposes a time-triggered approach. The runtime verification algorithm is based on following time steps. This implies a delay in the problem detection. The OREO tool instead follows an event-triggered strategy, which allows the detection of malfunctions as soon as possible even maintaining a low overhead.

### 8.3. Runtime Monitoring Frameworks

The work of Kong et al. [1] (see Table 4), proposes, like the present work, a comprehensive runtime monitoring framework: capable of extracting execution traces and subsequently analyzing them to study behaviors that could lead to failures. However, the configuration cost of this methodology could be time-consuming and complex. Additionally, although it is flexible, it does not seem capable of observing the lifecycle of components and does not provide a profiler for analyzing error propagation.

In the paper of Simmonds et al. [11], a runtime verification framework for web services is presented. Specifically, the approach focuses on the dependencies, referred to as *conversations* in the paper, established at runtime between web services. The authors provide an ad-hoc syntax to specify a subset of UML 2.0 sequence diagrams used to express web service conversation properties. Sequence diagrams are then automatically translated into monitor automata used to verify at runtime if the system complies with the properties. Although the framework provides countless possibilities for defining new properties to monitor due to the sequence diagram formalism, the configuration process may be expensive and error-prone and needs to be defined for each specific application. Additionally, both the overhead introduced by the framework and its scalability are not investigated. In contrast to such work, OREO provides a lightweight configuration phase and implies a low and scalable overhead.

A distributed service-based software architecture to support runtime verification for edge-intensive systems is presented in paper [37]. Edge nodes (ENs) host runtime monitors that receive events from end devices and other ENs, and express system requirements. Property evaluation occurs on the board of ENs, and utilizes Metric First-Order Temporal Logic to formalize traces of events. As a concrete case study, the authors considered a spatially-distributed parking system in a smart city, on which a resource-constrained edge computing environment was the testbed. In contrast to this work, OREO does not require establishing an entire software architecture but simply consists in a tool that can be deployed alongside the application under analysis. The plug-and-play nature of OREO provides improved applicability and flexibility.

The development of an efficient model-checking procedure for Internet of Things (IoT) systems, during runtime verification, is the focus of the paper by Lee et al. [9]. In particular, a cache mechanism to reduce the computational time spent for abstraction and verification is integrated into the procedure developed. The paper focused on model checking based on finite-state machine abstraction and model transition as equations, with the aim of verifying the runtime state of IoT applications. In contrast, the framework we propose does not rely on model-checking procedures. OREO, therefore, can overcome with ease the problems identified as critical by the authors themselves: like lack of expressivity in properties definition, and fragility to changes of the system under analysis.

IoT systems are also the main target of the paper of Incki et al. [38], where a novel runtime verification approach for IoT systems is proposed. A domain-specific event calculus (EC) for Constrained Application Protocol (CoAP)-based IoT systems, and an EC-to-EPL statement mapping are developed, to favor the exploitation of Esper complex event processing engine. The validity of the framework presented is then illustrated by taking into account use case applications and analysis. However, the performance impact of the proposed runtime verification architecture is not considered in this specific work.

### 8.4. Trace Analysis

In the work of Bonnah et al. [39], efficient algorithms for offline runtime monitoring are presented. In particular, the work aims to exploit the compactness and expressivity of the time window temporal logic (TWTL) [40] in runtime verification tasks. The approach proposed by Bonnah et al. [39] identifies, as a foundation, basic rewriting rules which are used to iteratively replace subordinate terms of the TWTL formula until they are reduced to truth values. To illustrate the validity and the applicability of the algorithms developed, authors formalized the quality of service (QoS) constraints imposed by unmanned aerial vehicles (UAVs) in time-critical surveillance missions as TWTL specifications. Then, QoS constraint satisfaction is monitored by means of the algorithms proposed. Although using languages like TWTL to express time-bounded properties of the business logic may be effective, the work of Bonnah et al. proposes and evaluates only the algorithms to solve the defined properties. Conversely in our work, we provide and evaluate an entire framework giving a concrete implementation oriented to runtime verification.

An approach to monitoring the workflow temporal conformance (i.e., workflow temporal verification), aiming at ensuring QoS satisfaction over workflow completion time, is presented in the work of Luo et al. [10]. Specifically, the authors develop an efficient and effective procedure to monitor the running behavior of parallel business workflows in the cloud, in order to promote on-time workflow completion. The runtime temporal conformance of workflows is measured by evaluating the workflow throughput for describing the behavior of a large aggregation of parallel workflow instances, differently from existing approaches based on the response time of each activity composing the workflow, to reduce verification overhead. Then, verification checkpoints are introduced, i.e., instants where temporal verification needs to be performed to check the temporal conformance state, to reduce energy consumption. The approach presented by Luo et al. aims to achieve better efficiency and effectiveness in parallel business workflow instances. In contrast, the framework we present in this work support reliability tasks. However, the extensibility of OREO allows the implementation of similar temporal conformance verification strategies in the business logic context.

In another related work focusing specifically on C and C++, Havelund presents LogScope [41], a system for monitoring event streams against formal specifications. Differently from such work, OREO is utilized to extract the execution traces, rather than requiring them as input. In this way, in contrast to LogScope, OREO provides a complete monitoring setup, from the extraction of events to their analysis of their properties. Additionally, OREO focuses on Fault-Error-Failure chain analysis which, while it could be implemented also *via* LogScope, is not considered as one of LogScope's current application scenarios.

### 8.5. Runtime Behavior Visualization

Havelund et al. [42] design and implement the DejaVu tool for monitoring first-order past linear-time temporal logic over a sequence of events. In the paper presented by Ma et al. [43] a novel fault localization method that comprises a first phase of fault-related statement localization, and a second step consisting of fault comprehension is designed. The method developed analyzes the dependence probability of each statement to find the fault-related statements and their propagation to discover the true fault.

In the work of Smyth et al. [44], an approach to bridge the gap between domain-specific notation and modeling/programming languages is presented. In opposition to such work, we focus on the dual aspect Smyth et al. focus on, namely the documentation and analysis of execution traces, rather than making documentation executable. In a different work by Dams et al. instead [45], dynamic documentation using runtime verification *via* monitoring and visualization techniques are discussed. OREO builds upon the future work of Dams et al. [45] by showcasing how a formalization based on the dynamic behavior of a software-intensive system can be used to support its development and maintenance. In another related work by Gorostiaga *et al.* [46], a runtime verification solution based on stream runtime verification is presented. Instead of focusing on general Fault-Error-Failure chain analysis as done in OREO, the work of Gorostiaga *et al.*consider a specific application domain and type of failure, namely robotics and silent mission failures. Apart from the context and failure type, OREO also differs from the work of Gorostiaga et al. in terms of end goal, as the approach of Gorostiaga et al. is used for prediction and correction of robot behaviour, while OREO focuses on the development and maintenance of software-intensive systems.

## 9. Conclusion

This paper focused on the software runtime monitoring, contextualized to the fault localization and error propagation problem. The open-source software tool developed proposes a representation of the running status of the software monitored, catching the interactions and dependencies established among components during their life cycle, considering possible UI interactions. Such an execution component abstraction is exploited to perform runtime fault localization. Performance evaluation illustrates the remarkable ability of the software released in catching and extracting the execution behavior of different software architectures, exhibiting scalability, and confirming its suitability in addressing fault localization problems. Finally, an in-depth discussion about OREO practical complexity, and its possible application to a rich variety of different scenarios are provided. In future work, we plan to apply OREO in an industrial case study on a real system. This will involve engaging developers to use OREO as a tool for understanding the runtime behavior of the system, as well as a tool to support the replication of failure and identification of real heisenbugs.

## References

[1] S. Kong, M. Lu, L. Li, L. Gao, Runtime monitoring of software execution trace: Method and tools, IEEE Access 8 (2020) 114020–114036. doi:10.1109/ACCESS.2020.3003087.

[2] H. Lu, A. Forin, The design and implementation of p2v, an architecture for zero-overhead online verification of software programs (09 2007).

[3] O. Baldellon, J.-C. Fabre, M. Roy, Minotor: Monitoring timing and behavioral properties for dependable distributed systems, in: 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing, 2013, pp. 206–215. doi:10.1109/PRDC.2013.41.

[4] N. Mahadevan, A. Dubey, G. Karsai, Application of software health management techniques, in: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 1–10.

[5] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, I. Neamtiu, Finding and reproducing heisenbugs in concurrent programs., in: OSDI, Vol. 8, 2008.

[6] L. Scommegna, R. Verdecchia, E. Vicario, Unveiling faulty user sequences: A model-based approach to test three-tier software architectures, Journal of Systems and Software 212 (2024) 112015.

[7] T. Dohi, K. S. Trivedi, A. Avritzer, Handbook of software aging and rejuvenation: fundamentals, methods, applications, and future directions, World scientific, 2020.

[8] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, H. Zeisel, Reminds : A flexible runtime monitoring framework for systems of systems, Journal of Systems and Software 112 (2016) 123–136. doi:https://doi.org/10.1016/j.jss.2015.07.008.
URL https://www.sciencedirect.com/science/article/pii/S0164121215001478

[9] E. Lee, Y.-D. Seo, Y.-G. Kim, A cache-based model abstraction and runtime verification for the internet-of-things applications, IEEE Internet of Things Journal 7 (9) (2020) 8886–8901. doi:10.1109/JIOT.2020.2996663.

[10] H. Luo, X. Liu, J. Liu, Y. Yang, J. Grundy, Runtime verification of business cloud workflow temporal conformance, IEEE Transactions on Services Computing 15 (2) (2022) 833–846. doi:10.1109/TSC.2019.2962666.

[11] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, J. Waterhouse, Runtime monitoring of web service conversations, IEEE Transactions on Services Computing 2 (3) (2009) 223–244. doi:10.1109/TSC.2009.16.

[12] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: Lectures on Runtime Verification, Springer, 2018, pp. 1–33.

[13] S. He, P. He, Z. Chen, T. Yang, Y. Su, M. R. Lyu, A survey on automated log analysis for reliability engineering, ACM computing surveys (CSUR) 54 (6) (2021) 1–37.

[14] P. Las-Casas, G. Papakerashvili, V. Anand, J. Mace, Sifter: Scalable sampling for distributed traces, without feature engineering, in: Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 312–324.

[15] J. Mertz, I. Nunes, Software runtime monitoring with adaptive sampling rate to collect representative samples of execution traces, Journal of Systems and Software 202 (2023) 111708.

[16] M. Grottke, K. S. Trivedi, Fighting bugs: Remove, retry, replicate, and rejuvenate, Computer 40 (2) (2007) 107–109.

[17] J. Mertz, I. Nunes, Automation of application-level caching in a seamless way, Software: Practice and Experience 48 (6) (2018) 1218–1237.

[18] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE transactions on dependable and secure computing 1 (1) (2004) 11–33.

[19] M. Fowler, Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch, Addison-Wesley, 2012.

[20] R. C. Martin, Design principles and design patterns, Object Mentor 1 (34) (2000) 597.

[21] J. Parri, S. Sampietro, L. Scommegna, E. Vicario, Evaluation of software aging in component-based web applications subject to soft errors over time, in: 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2021, pp. 25–32.

[22] R. Alur, K. Etessami, P. Madhusudan, A temporal logic of nested calls and returns, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2004, pp. 467–481.

[23] A. Goldberg, K. Havelund, Automated runtime verification with eagle., in: MSVVEIS, IEEE, 2005.

[24] B. Bollig, N. Decker, M. Leucker, Frequency linear-time temporal logic, in: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering, IEEE, 2012, pp. 85–92.

[25] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu, An overview of the mop runtime verification framework, International Journal on Software Tools for Technology Transfer 14 (3) (2012) 249–289.

[26] M. Müller, Practical JSF in Java EE 8, Springer, 2018.

[27] F. Patara, E. Vicario, An adaptable patient-centric electronic health record system for personalized home care, in: 2014 8th International Symposium on Medical Information and Communication Technology (ISMICT), IEEE, 2014, pp. 1–5.

[28] S. Fioravanti, S. Mattolini, F. Patara, E. Vicario, Experimental performance evaluation of different data models for a reflection software architecture over nosql persistence layers, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, 2016, pp. 297–308.

[29] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[30] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.

[31] R. Verdecchia, E. Engström, P. Lago, P. Runeson, Q. Song, Threats to validity in software engineering research: A critical reflection, Information and Software Technology 164 (2023) 107329.

[32] G. J. Myers, T. Badgett, T. M. Thomas, C. Sandler, The art of software testing, Vol. 2, Wiley Online Library, 2004.

[33] M. Du, F. Li, G. Zheng, V. Srikumar, Deeplog: Anomaly detection and diagnosis from system logs through deep learning, in: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, 2017, pp. 1285–1298.

[34] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, J. Xu, An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services, in: 2017 IEEE international conference on web services (ICWS), IEEE, 2017, pp. 25–32.

[35] J. Mertz, I. Nunes, Tigris: A dsl and framework for monitoring software systems at runtime, Journal of Systems and Software 177 (2021) 110963.

[36] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, T. Rakotoarivelo, Efficient and scalable runtime monitoring for cyber–physical system, IEEE Systems Journal 12 (2) (2018) 1667–1678. doi:10.1109/JSYST.2016.2614599.

[37] C. Tsigkanos, M. M. Bersani, P. A. Frangoudis, S. Dustdar, Edge-based runtime verification for the internet of things, IEEE Transactions on Services Computing 15 (5) (2022) 2713–2727. doi:10.1109/TSC.2021.3074956.

[38] K. Incki, I. Ari, A novel runtime verification solution for iot systems, IEEE Access 6 (2018) 13501–13512. doi:10.1109/ACCESS.2018.2813887.

[39] E. Bonnah, K. A. Hoque, Runtime monitoring of time window temporal logic, IEEE Robotics and Automation Letters 7 (3) (2022) 5888–5895. doi:10.1109/LRA.2022.3160592.

[40] C.-I. Vasile, D. Aksaray, C. Belta, Time window temporal logic, Theoretical Computer Science 691 (2017) 27–54.

[41] K. Havelund, Specification-based monitoring in c++, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2022, pp. 65–87.

[42] K. Havelund, D. Peled, D. Ulus, Dejavu: A monitoring tool for first-order temporal logic, 2018, pp. 12–13. doi:10.1109/MT-CPS.2018.00013.

[43] P. Ma, Y. Wang, X. Su, T. Wang, A novel fault localization method with fault propagation context analysis, in: 2013 Third International Conference on Instrumentation, Measurement, Computer, Communication and Control, 2013, pp. 1194–1199. doi:10.1109/IMCCC.2013.265.

[44] S. Smyth, J. Petzold, J. Schürmann, F. Karbus, T. Margaria, R. von Hanxleden, B. Steffen, Executable documentation: test-first in action, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2022, pp. 135–156.

[45] D. Dams, K. Havelund, S. Kauffman, Runtime verification as documentation, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2022, pp. 157–173.

[46] F. Gorostiaga, S. Zudaire, C. Sánchez, G. Schneider, S. Uchitel, Assumption monitoring of temporal task planning using stream runtime verification, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2022, pp. 397–414.