2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) | 978-1-6654-2603-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSREW53611.2021.00040

Evaluation of software aging in component-based Web Applications subject to soft errors over time

Jacopo Parri*, Samuele Sampietro†, Leonardo Scommegna‡ and Enrico Vicario§
Department of Information Engineering, University of Florence
Florence, Italy
Email: *jacopo.parri@unifi.it, †samuele.sampietro@unifi.it, ‡leonardo.scommegna@unifi.it, §enrico.vicario@unifi.it

***Abstract*—Modern Web Applications rely on architectures usually designed with modular software components whose behaviour is shaped over fundamental principles and characteristics of the HTTP protocol. Dependency Injection frameworks support designers and developers in the automated management of components lifecycle, binding them to predefined scopes, thus delegating to an outer and independent participant the responsibility of creation, destruction and inter-dependencies definition of runtime instances. In this way, different scopes configurations implicitly act as different software micro-rejuvenation policies, emphasising the importance of choices in the assignment of component scopes; while supporting the *stateful* behaviour in data-retention mechanism, wider scopes may majorly expose *in-memory* components to software aging processes.**

We report a practical experience illustrating how the memory maintained in the business logic of a Web Application may give space to aging processes affecting the runtime behaviour of a stateful web application, and we show how this threat is contrasted by micro-rejuvenation at component level implemented by the container under different assignment strategies for components scopes. To this end, we propose an accelerated testing approach relying on a fault injection process that executes an event-driven simulation of arising *soft errors over time*. Experimentation on an exemplary web application implemented on the stack of Java Enterprise Edition show how manifestation, correction, and propagation of errors are conditioned by different scopes assigned to components by the software developer.

***Index Terms*—Accelerated testing, Software Aging, Software Micro-Rejuvenation, Fault Injection, Soft Errors, Software Architecture.**

I. Introduction

In the architecture of Web Applications [1]–[3], the state of user interaction is commonly maintained in a layer of transient components so as to reduce the workload on the DataBase Management System (DBMS) and to avoid persistence of intermediate data that become unrelevant after completion of the user goal [4]. In particular, in *stateful* architectures these software components are maintained server-side, in a *business logic* layer [5] made of page controllers and utility beans (e.g., Data Access Objects in architectures exploiting the Object Relational Mapper pattern [6]) that serve the client in the access to displayed data and control page navigation according to user actions.

Most Web Applications exploit *ad hoc* functionalities for automating management of components lifecycle according to built-in *scopes*, shaped by the client-server paradigm and the HTTP protocol [7], [8] and specialized to fit specific needs of application use cases: components with a *session* scope maintain their state along the entire HTTP session, from the first user contact (e.g., the login use case) until the application is left (e.g., the logout use case); whereas, *request* scoped components live only for the time interval needed to serve a single HTTP request triggered by a user action (e.g., for displaying a page content). Other components live along an intermediate scope, here referred to as *conversation* (from the term used in the Java Enterprise Edition ecosystem), demarcated by specific begin/end programmatic events that capture the limits of a use case in the *function level* [4] (e.g., in the management of the shopping cart of an e-commerce transaction). Finally, components with *application* scope live from start-up to shut-down of the application (e.g., to handle log data or to maintain shared variables accessed by multiple users).

In the good practice of Web Applications development, *lifecycle management* of stateful components in the business logic layer is delegated to a *container*, which implements the architectural pattern of *Dependency Injection* [9] and takes care of creation, sharing, and destruction of components instances. Various technologies support Dependency Injection and automated lifecycle management in major programming languages and ecosystems, notably including *Autofac* [10] in C# and *Contexts and Dependency Injection* (CDI) [11] in Java.

Operation of the *container* is controlled by the software developer through class-level configurations (e.g., Java annotations) that associate all the instances of each plain type with a scope, which elevates *plain object* types to the level of *managed components*. When doing so, the developer has a design space in the choice of scopes to be associated with each type of bean, which implicitly specifies a kind of software rejuvenation policy that will be implemented by the *container* through automated handling of creation, initialisation, and destruction of managed components. Note that this will not result in reboot of the physical servers (hot/cold spares) [12], of the *Virtual Machine* [13], of the *Application Server* [14], or of the client-side mobile device [15], [16]. Instead, this comprises a reboot at the software component-level

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00040

that produces a kind of *micro-rejuvenation* [17], [18]. In this perspective, wider scopes, maintain alive (in-memory) instances for a longer time and thus expose components to aging processes [19], [20], while smaller scopes produce more frequent refresh of the state of each single component instance. Specifically, by associating each managed type with a *request* scope, the developer maximises the frequency of rejuvenation, which will occur for each component on completion of the actions triggered by each HTTP request generated by the User Interface (UI). This minimises the probability that an error, occurred in the state of some managed component, is propagated in the computation and then transferred, either to other components or to functional behaviour delivered by the UI. However, components living only for the time of a single request result in *stateless* behavior and thus require that data needed along the user *session* be stored with some more resource demanding action, usually implemented through a DBMS access or through web cookies [21]). Conversely, if a component is associated with the *session* scope, any error accumulated along its interactions may be maintained and propagate along the Fault-Error-Failure chain [22] established by dependency relationships among components. as a logical consequence, the aging effects may be reduced by *conversation*-scoped components and may be maximised by *application*-scoped components, that maintain and propagate their states until the application shut-down.

In this paper, we report a practical experience illustrating how the memory maintained in the business logic of a Web Application may give space to aging processes affecting the runtime behaviour of a stateful web application, and we show how this threat is contrasted by micro-rejuvenation at component level implemented by the container under different assignment strategies for components scopes.

To this end, we present an approach for accelerated testing [23]–[25] based on a process of software fault injection [26], [27] that emulates the arise of errors arriving over time, as in the case of *soft errors* [28], during the operation of a *stateful* Web Application. Experimental results, based on the implemention of the proposed approach in a fault injector and its application to an exemplary Web Application implementing common patterns of the good practice based on the JEE technology stack, permit to observe how different scopes assigned by the software developer yield different sensitivity to errors arriving over time, both when assigned to individual components or when applied as global policies.

The rest of the paper is organized as follows: in Sect. II, the proposed approach for accelerated testing of Web Applications designed with distinct scoped components is described; in Sect. III, the experimentation of the approach on a stateful Web Application, developed with JEE Specifications, is discussed and significant results are presented; conclusions are finally drawn in Sect. IV.

II. Accelerated Testing Approach

We describe the accelerated testing approach implemented and applied in the experiments, specifying principles of operation for injection and simulation of errors arriving over time (Sect. II-A) and reporting details of a concrete implementation that was developed to support fault injection for Web Applications in the JEE environment (Sect. II-B).

A. General principles of operation

The testing approach aims at observing the behaviour of a stateful Web Application relying on Dependency Injection and automated lifecycle management - here termed Implementation Under Test (IUT) - when external failures bring components of the business logic layer in a state of error that may eventually affect functions delivered by the UI. In particular, this is intended to reproduce the impact of various kinds of external failures arriving *over time*, independently from the specific actions taken by the user and by the application. As a notable case, this can result from *soft errors* affecting transient and persistent memories (e.g., RAM, cache, hard disks) [28]–[30]. This class of errors, due to cosmic rays and alpha particles, was widely addressed in the literature of dependability, in machine learning techniques based on (deep) neural networks [31], [32], in availability and reliability analyses of system-level effects over data storage systems [33], and more recently in specific evaluation of sensitivity different RESTful frameworks [34].

To this end, we consider a suite of test cases, each made of the sequence of HTTP requests received by the server-side during the realization of a variety of use cases by a real user. To make the effect of errors observable, testing is accelerated by replacing the real-time concurrent execution of HTTP sequences and error arrivals with a simulation that reproduces the real sequence of concrete states traversed by the application in the test case, assumes that the state sojourns in each state for a random duration with controlled distribution, and determines the state where the error is injected by evaluating the probability that each concrete state is hit at the error arrival time.

In the current version of the error injection tool, inter-arrival times between subsequent requests are assumed to be exponentially distributed, and errors are assumed to arrive according to a uniform distribution within the total duration of the sequence. Both the assumptions can be easily relaxed with no significant impact on the injector implementation. Note that the latter assumption of uniform distribution comprises a fairly precise approximation for the case where: errors arrive according to a Poisson process with a constant rate much lower than that of the sojourn time in each visited state; and, the evaluation is limited to observe only the tests hit by at least one arrival. Under these conditions, the probability of multiple errors during the same test case is negligible, and the arrival time is distributed according to a truncated exponential that can be closely fit by a uniform distribution.

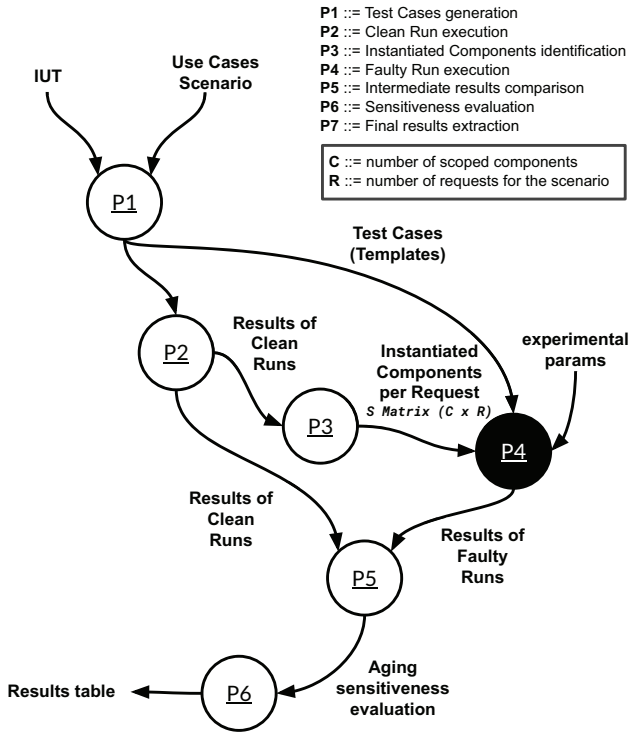

Fig. 1. Data flow diagram representation of the proposed accelerated testing approach that compares results of the execution of "faulty" runs subject to fault injection strategies with that of a "clean" run interpreting the expected IUT behaviours.

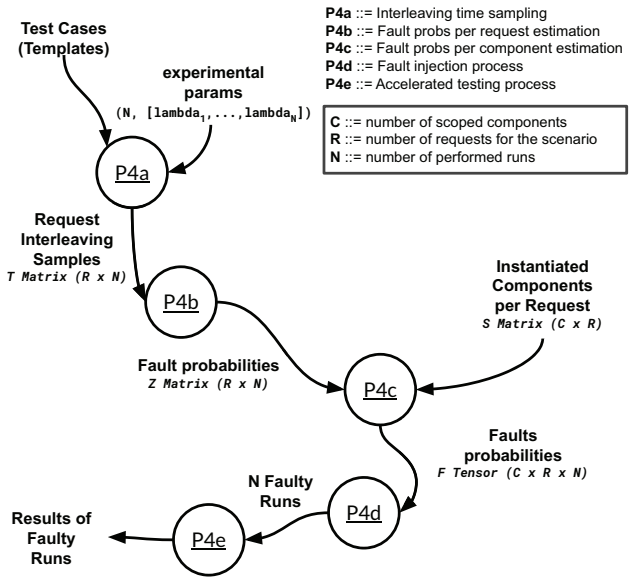

Fig. 2. Data flow diagram detailing inner sub-processes of the core *P4* process of Fig. 1, outlining the stochastic characterisation of the approach.

According to this, the injection of a fault in a run is obtained: *i)* by sampling interleaving times of subsequent requests; *ii)* by deriving the probability that the fault is injected between them proportionally to the width of the interval/interleaving time itself; and *iii)* by equally distributing the probability among involved *in-memory* components. In so doing, an event-driven simulation is enabled where the time is not linearly accelerated by "waiting" and "sleeping" mechanisms so as to speed up the flow of time, but it is continuously carried forward to the instant corresponding to the nearest future event (i.e., the lowest sample).

A conceptual representation of the approach is summarised in the data flow diagram of Fig. 1, where main involved elaboration processes are depicted. The IUT and the sequence of selected use cases (i.e., the monitored usage scenario) act as the primary inputs of the overall approach for generating in output a table of results: *P1* performs the test case generation, obtaining a set of concrete test case template implementations; *P2* executes a "clean" run over the IUT, executing the test case in absence of faults; *P3* identifies components which are alive and instantiated before each request in order to determine an $(C \times R)$ binary matrix, named S, where C is the number of scoped components and R is the number of requests fixed for the simulation scenario; *P4* executes N "faulty" runs of the selected test case, each one subject to a different injected fault; *P5* compares the results of the N faulty runs with the results of the "clean" run in order to evaluate the aging sensitiveness of the involved components; finally, *P6* elaborates the output evaluation of *P5*, extracting the final format.

Fig. 2 details the sub-flow of the core process (i.e., *P4*) for the whole approach. Specifically, the sub-process *P4a* exploits experimental parameters together with the generated test case templates so as to enabling a sampling stage of request interleaving times between subsequent requests. In particular, different lambda rate parameters can be adopted for each request, in order to emulate the human user interaction time with the UI (e.g., filling a text area field within a form takes different times wrt clicking an action button). *P4a* generates a $(R \times N)$ matrix, named T, whose $t_{i,j}$ element is the sampled interleaving time of the *i-th* request in the *j-th* run, and N is the total number of performed runs. *P4b* elaborates a $(R \times N)$ matrix, named Z, representing for each request of each run the probability that the fault is injected before the arrival of the request. The probability $z_{i,j} = \frac{t_{i,j}}{\sum_{\xi=1}^{R} t_{\xi,j}}$ is directly proportional to the width of the interval sample of the *i-th* request wrt all requests samples of the considered run. As a consequence, the $\sum_{i=1}^{R} z_{i,j} = 1$. *P4c* mixes fault probabilities values of the Z matrix, with information about instantiated components of the binary S matrix (where $s_{i,j}$ is equal to 1 if and only if an instance of the *i-th* component is alive in the *j-th* request), for obtaining the $(C \times R \times N)$ tensor, named F. The probability $f_{i,j,k} = \frac{z_{j,k} \cdot s_{i,j}}{\sum_{\xi=1}^{C} s_{\xi,j}}$ is computed as the probability that the

injected fault affects the *j-th* request in the *k-th* run, divided by the number of instantiated and alive components. *P4d* launches N test runs, injecting faults in compliance with the probabilities of the F tensor; finally, *P4e* elaborates the output results.

B. Concrete implementation in the JEE ecosystem

The simulation of a real usage of the IUT requires to exploit a *web driver* - in this work implemented with the automation testing framework, named *Selenium Web Driver* [35] - for mocking user interactions on the UI (e.g., click of a button, fill of a text field) and for facilitating the inspection of displayed values (making assertions on displayed texts or on the existence of specific UI widgets). Usually, web drivers are not capable of accessing the actual alive components instances, preventing a wide range of *grey box* approaches. To overcome this limitation, the *Arquillian* [36] ecosystem has been adopted in conjunction with the *Arquillian Warp* extension for accessing all the contextual instances living server-side before, or after, a simulated user interaction.

Our implemented JEE module is designed for retrieving all living contextual instances, managed in background by the CDI framework, so as to perturb them during the execution of the simulation scenario. In so doing, fault injections can be programatically triggered before the concrete execution of a user interaction on the UI, thus enabling actuation of request arrival times and errors activation, as described in Sect. II-A.

For each executed simulation of a scenario, different indicators are monitored and collected:

- *the final state of the application*; we interpret it as the collection of the living contextual instances after the completion of the last HTTP request of the scenario;
- *the state of the perturbed contextual instance*; during the execution, the fault injection is triggered at a defined time instant (see Sect. II-A for details) perturbing one of the living components. Due to the implemented rejuvenation strategy or to the error propagation mechanism, it may be difficult to retrieve the original perturbed component simply by knowing of the final application state. To this end, after the injection, the component fully qualified name, its scope and the type of perturbation are saved;
- *the number of failures manifested during the simulation*; in a FEF chain perspective, only top-level failures (i.e., failures manifested on the UI and visible to the end-user) have been considered, thus neglecting failures of intermediate components.

```
1  @Test
2  public void testScenario1() {
3   ...
4   // A user interaction
5   Warp.initiate(new Activity() {
6     @Override
7     public void perform() {
8       homePage = new SimpleAppHome(driver);
9
10      if (!homePage.checkPageCorrectState())
11        failureOccurred(runInfoFilePathStr);
12
13      homePage.clickButton();
14    }
15  }).inspect(new InjectionInspection() {
16
17    String path = testDirStr;
18
19    boolean executeFaultInj = shouldInject();
20
21    @Inject
22    BeanManager bm;
23
24    @BeforeServlet
25    public void injectFault(){
26     if (executeFaultInj)
27       injectFault(bm,
28             path + "/errorInjected");
29    }
30
31    // Invoked only in the final interaction
32    @AfterServlet
33    public void saveFinalState() {
34     InstanceFinder.gsonPrint(
35         InstanceFinder.retrieveState(bm),
36         finalStateDir);
37    }
38  });
39  ...
```

Listing 1. Java code snippet representing how a simulation scenario is concretely implemented through the proposed JEE module.

Listing 1 reports an illustrative code snippet related to a test case for illustrating the implementation style of a simulation scenario. The `Warp.initiate()` method (*line* 5) defines what happens in a single user interaction from the end-user perspective, defined in turn by the `Activity` class definition through the overriding of the `perform()` method (*line* 7). Assertions and behaviours from the server-side perspective are defined in the `inspect()` method (*line* 15) through the `InjectionInspection` class definition; since considered scenarios are composed by a sequence of user interactions, their implementation will be a sequence of invocations of `initiate()` and `inspect()` methods. The snippet also shows the invocation of the prescribed user action (i.e., `clickButton()` at *line* 13) only after the evaluation of the occurrence of a top-level failure (i.e., `checkPageCorrectState()` at *line* 10) within the current page (i.e., `homePage`). The `InjectionInspection` class has been defined for setting up some action hooks, through *ad hoc* annotation decorators (i.e., `@BeforeServlet` and `@AfterServlet`), where invoke specific methods (e.g., `injectFault()` and `saveFinalState()`). Specifically, the `injectFault()` method (*line* 25) is responsible for triggering the fault injection, if required (i.e., `shouldInject()` evaluation at *line* 19), and save the state of perturbed components, according to the probability matrix obtained from the tensor $F = [F]_{i,j,k}$ where k is constrained to the current run, as described in Sec II-A. Finally, `saveFinalState()`

```
1  @Deployment(testable = true)
2  public static WebArchive createDeployment() {
3    WebArchive war = ShrinkWrap
4    .create(WebArchive.class, "deployment.war")
5    .addPackages(true, "appPackageName")
6    .addClass(InjectionInspection.class);
7    .addClass(FaultInjector.class)
8    .addClass(InstanceFinder.class)
9    .addClass(ObjectConverter.class)
10   .addClass(CtxInstanceExclusionStrg.class)
11
12   return war;
13 }
```

Listing 2. Java code snippet of an exemplary deployment configured and generated through *ShrinkWrap* for the *Arquillian* test suite.

method at *line* 33 retrieves all the component instances in order to save them within a file in the JSON format.

As a significant note, through the *Arquillian* deployment mechanism, which adopts *ShrinkWrap* [37] for the creation of Java archives files, our implemented module does not need to be integrated within the production source code of the IUT, thus enabling an experimentation stage with "no modifications" on the Web Application under test.

Listing 2 represents the deployment of the archive referred to the application (i.e., usually a *.war* file) that is essential for the configuration of the test environment. This is accomplished by implementing the *public static method* annotated with `@Deployment` that returns the archive. As mentioned above, the approach is so powerful and flexible that allows to insert dynamically classes into an application without modifying the real source code of the IUT; in this way, it enables the design of tailored archives with only the classes needed for the test suite, thus obtaining lighter deployments and, consequently, speed up tests executions. As can be seen at *line* 5, it is possible to add entire packages to the deployment (in this snippet, the root package of the IUT is added), but also to add single classes, as in *lines* $6-10$. Specifically, the `InjectionInspection` class is a custom extension of the `Inspection` Warp class, exposing the implementation of the `injectFault()` method (invoked at *line* 25 of Listing 1). The `FaultInjector` class provides the specific faults for a run, relying on the `InstanceFinder` class for retrieving server-side components instances, and on the `ObjectConverter` class which converts a random number in other base types. The `InstanceFinder` is also used in Listing 1 at *lines* 34 and 35 in order to generate the JSON file and save the final state. Lastly, the `CtxInstanceExclusionStrg` class defines custom serialisation strategies (e.g., during the serialisation process from object to JSON string, it selects only the state of a subset of contextual instances).

Our concrete module randomly chooses a contextual instance among the available living ones and, then, randomly picks one of its fields before applying a perturbation (i.e., if the field is a native numerical or literal type, a random number is assigned, otherwise if the field is a structured object instance, its internal state is perturbed in compliance to its inner fields).

The implementation of the fault injector is opened to further integrations for additional behaviours, such as components/-fields blacklists (i.e., lists of components/fields that should be ignored by the injector), components (or fields) specific extraction probabilities (also depending on their state), or special perturbation strategies (i.e., it could be defined a specific perturbation for a component or a group of components[1]).

III. Experimentation

For the experimentation stage, the behaviour of the IUT has been analysed: activating a fault does not necessarily causes an immediate failure, it could rather lead to an erroneous internal condition (i.e., the erroneous state) remaining silent for an unpredictable period of time or evolving into further errors (i.e., error propagation). Under these assumptions, a failure could be caused by a set of events occurring over a long-term period of time, making the failure detection and the subsequent fault removal phase extremely difficult [39].

For all these reasons, the experimentation has been based not only on failures caused by the fault injection but also on the runtime internal state of the IUT; raw data obtained during each performed simulation with the JEE module (described in II-B) is exploited so as to derive further information about the effects of simulated error activations. Each run is classified with one of the following categories:

- **manifested failures** (i.e., top-level failures manifested during the run) highlighting that errors have deviated some external states of the system also affecting functionalities and services offered through the UI, thus implying that the rejuvenation policy was "too soft". This information is derived, addressing the number of failures observed by the JEE module during each simulation;
- **latent errors** (i.e., errors that do not contribute in top-level failure manifestations neither are corrected by the rejuvenation policy) remaining hidden during the run. This information is derived comparing the final state of the "clean" run with the final state of the "faulty" run (if at least one component state differs from its counterpart obtained after a "clean" run and there are no manifested failures in the simulation, then a latent error occurs);
- **corrected errors** (i.e., errors that are automatically corrected by the Dependency Injection container) outlining the success of the rejuvenation strategy defined through the designed component scope. This information is derived by comparing the final state obtained after a "clean" run with the final state of the simulation (if there are no

[1]For example, if the field extracted is an *ordered list*, it could be defined a type of perturbation that changes the order of the list items, alternatively, if the field is a *string* that requires a specific input format (e.g., a mail address) it could be defined a perturbation that changes the address with another random one, still respecting the constraint (e.g., email regex [38]).

TABLE I
SCOPE-WISE EXPERIMENTATION RESULTS: DIFFERENT ASPECTS RELATED TO SENSITIVITY TO ERRORS OVER TIME ARE REPORTED FOR EACH STATEFUL SCOPE DURING AN EXPERIMENTATION MADE OF 200 RUNS.

Scope	Manifested failures (%)	Latent errors (%)	Corrected errors (%)
application	42.4	57.6	0
session	48	36	16
conversation	12	24	64

differences and if no failures have occurred, then the error has been corrected successfully).

The proposed approach for accelerated testing has been experimented in two modes: *i)* **scope-wise experimentation**, with the aim of studying how the perturbation of a component of a certain scope can affect the overall behaviour of the IUT, thus demonstrating that long-living components are more error-prone than short-living ones; *ii)* **policy-wise experimentation**, with the aim of comparing two different - *but functionally equivalent* - versions of a simple Web Application designed under two distinct principles for the lifecycle design of scoped components (i.e., data long retention principle vs lower scope principle) in terms of reliability.

The results of the *scope-wise experimentation* are outlined in Tab. I; as can be seen, the wider scopes (i.e., *session* and *application*) tended to manifest failures and to show the presence of latent errors more frequently than the narrower one (i.e., *conversation*) which, in general, demonstrated a greater robustness to errors over time.

Note that *application* scope never corrects error, this is due to the fact that no rejuvenation strategy is applied. As a remark, the *request* scope is not reported in the table since its components instances live only within single HTTP requests and, so, their state is continuously refreshed almost preventing errors over time to perturb them.

It is also interesting to examine how components associated with wider scopes are more likely to be picked up by the fault injector (our experimentation has shown a sampling probability of 0.625, 0.25 and 0.125 for *application*, *session* and *conversation*, respectively, after 200 runs). This intrinsically reflects the stochastic characterisation of the approach, adopted in sampling times and choosing components for the fault injection mechanism, described in Sect. II-A: components living more during the simulation have more chance to be chosen for the *soft errors* injection.

The *policy-wise experimentation* adopted the following principles for components lifecycle design:

- **data long retention principle**, which encourages the use of wide scopes (e.g., *application*, *session*), allowing to store in-memory information for extended period of times avoiding to retrieve or re-compute them; as a drawback the memory is majorly occupied by data that are essentially not useful in the current computation;
- **lower scope principle**, which promotes the usage of scopes as narrow as possible (e.g., *request*), minimising the memory occupation while requiring a dedicated place to store runtime data (e.g., the local or session storages in the client-side or dedicated databases in the server-side), thus producing an overhead in computation.

TABLE II
POLICY-WISE EXPERIMENTATION RESULTS: RESULTS OF THE COMPARISON BETWEEN A VERSION OF THE UNDER MONITORED WEB APPLICATION WHICH FOLLOWS THE LOWER SCOPE PRINCIPLE AND A VERSION WHICH FOLLOWS THE DATA LONG RETENTION PRINCIPLE. RESULTS ARE OBTAINED WITH 100 RUNS PER VERSION.

Principle	Manifested failures (%)	Latent errors (%)	Corrected errors (%)
data long retention	42	46	12
lower scope	30	26	48

TABLE III
ERRORS PROPAGATION METRICS OBTAINED ALONGSIDE THE SCOPE-WISE EXPERIMENTATION. THE PERCENTAGE OF "PROP. RATIO" COLUMN IS OBTAINED AS THE RATIO BETWEEN THE VALUES WITHIN "MEAN PROPAGATED ERRORS" AND "MEAN TOUCHED COMPONENTS" COLUMNS.

Scope	Errors propagation (%)	Mean propagated errors	Mean touched components	Prop. ratio (%)
application	44	1	4.6	21.7
session	32	1.33	2.5	53.2
conversation	16	1.25	1.7	73.5

The results of the *policy-wise experimentation* are outlined in Tab. II; as can be seen, values are consistent with the *scope-wise experimentation* since under the *lower scope principle* is plausible to assume that components are predominantly *conversation* or even *request* scoped, while under the *data long retention principle* the scopes will be mainly *session* and *application*.

While outperforming the *data long retention principle* in these results, the *lower scope principle* cannot be considered the best solution for every implementation; indeed, the selection of a specific policy implies a trade-off between the ability of correcting *Aging Related Bugs* (ARB) [40] and the related computation overhead.

If the rejuvenation is rarely applied, ARBs could be activated and propagated. Conversely, if the rejuvenation is frequently applied, the components state need to be stored in other dedicated places, also affecting the performance of the IUT. Moreover, narrower scopes may also not guarantee the problem resolution: some use cases in Web Applications need to retain data across many HTTP requests, so the data must be maintained within databases or frontends, which in turn are not protected from *soft errors* over time. In so doing, the problem may just have been moved elsewhere.

Besides, we also collected information about **errors propagation** (i.e., errors which manifest and propagate failures in external components dependent on directly affected ones, also producing the activation of external faults and consequent errors) that increase the number of the possible sources of failures. Specifically, we considered only propagations affecting dependent components that are in an erroneous state at the end of the run (neglecting cases in which the propagation is corrected by a rejuvenation of the component). This information is derived comparing the final state of the "clean" run with the final state of the "faulty" run (i.e., if other components near by the one, affected by the injection, are in an erroneous state - different from the state obtained after a "clean" run - then the error propagation occurs).

Errors propagation is a crucial parameter for the overall sensitivity to *soft errors* over time: it gives rise to the error accumulation problem [41], complicating the correction (i.e., more than one component has to activate the rejuvenation strategy to fix the error, also generating "chain reactions" where more components could cause failures and propagate errors*, recursively*). In Tab. III, we report propagation-related measurements obtained during the *scope-wise experimentation*; in particular, we provide the percentage of runs with at least one error propagation (i.e., the *Errors propagation* column), also providing the mean number of components (excluded the perturbed one) with an erroneous state at the end of each simulation (i.e., the *Mean propagated errors* column). Although these measures give an idea on the actual error propagation for each scope, they only partially express how components could spread errors throughout the IUT and to complete this observation, we also monitored the propagation capacity for each scope. In order to assess this measure, we added a "dirty bit" on each instance which indicates whether a component has been exposed to the possibility of being contaminated with an erroneous state by another instance, during the run, every time a component, with the dirty bit set to true, interacts with another component the dirty bit of the latter becomes true. Note that, the component interaction involves both sides of the dependency between scoped components: a dirty component (with the dirty bit enabled) could use a method of a component with the dirty bit disabled and then dirt it, on the other side, a clean component (with the dirty bit disabled) could call a method of a dirty component and, consequently, getting itself dirty. Even if, the *application* scope seems to have, on average, a lower number of propagated errors, the propagation occurs more frequently with *application* scoped components (44% of the times against the 16% of the *conversation* scoped ones) and in addition, the mean touched components for each scope are directly proportional with the lifecycle of the scope.

We also evaluated the ratio (i.e., the *Prop. ratio* column) between the "Mean propagated errors" and "Mean touched components" that represents how many components have been effectively affected in average wrt the components that might have been affected (i.e., touched components with a set dirty bit). For some kind of reason, while - *in the absolute sense* - wider scoped components results in propagating much more errors, the narrower scoped ones seem to have a higher rate in deviating the runtime behaviour of interacting components (i.e., activating some external faults in dependent components). We expect this to be due to the fact that implementations of narrower scoped components are strictly related to the inner business logic of use cases, thus generating stronger relationships and implementation couplings between components.

IV. Conclusions

In this paper, we presented an accelerated testing approach for evaluating *software aging* effects in *stateful* Web Applications subject to *micro-rejuvenation* policies, implicitly controlled through Dependency Injection and automated lifecycle management mechanisms. The approach leverages stochastic injection of faults simulating the occurrence of *soft errors* over time, which enables efficient event-driven simulation of long-running scenarios made by several client-server interactions subtending sequences of use cases.

We reported experimental results in two perspectives: on the one hand, through a *scope-wise experimentation* we outlined how long-living components are more error-prone; on the other hand, through a *policy-wise experimentation* we highlighted how the principle of minimising scopes (i.e., *lower scope principle*) increases the overall reliability of the system, reducing the probability that errors are propagated, with respect to the principle of adopting wide scopes (i.e., *data long retention principle*).

As a limit case, lifting to the extreme the *lower scope principle*, if all components were associated to the narrower scope (i.e., the *request* scope), then at each HTTP request the state of the whole application would be refreshed (i.e., the rejuvenation would act at the end of each request). In this case, the Web Application would actually implement a *stateless* behaviour (i.e., the state of each component is dependent only to the data enclosed to a single HTTP request and to the information persisted within dedicated data storages) thus collapsing in a kind of RESTful architecture [42]. At first glance, the *stateless* behaviour seems to provide rejuvenation rates high enough to avoid almost any kind of software aging, but actually, in this scheme, the complexity of the problem is rather delegated to external management systems, such as client-side frontends or *in-memory* databases, which are not necessarily less exposed to soft errors. In this perspective, also considering that backend servers should be more robust and stable than mobile devices (e.g., smartphones, tablets) usually hosting decoupled UIs, our research will focus on a deep qualitative and quantitative evaluation on how server-side micro-rejuvenation policies may affect external environments in the propagation of *soft errors*.

Further development is ongoing to strengthen results in various aspects with respect to various threats to validity. Though current results were obtained on a realistic state-of-the-art application, further development is ongoing to

experiment with a stable Web Application using real data collected in a *production* environment. This objective is largely facilitated by the structure of the fault injection module, which is designed to be easily pluggable to any JEE-based source code. Further experimentation will also address the dependency of results on the number of dependencies and the type of monitored components, so as to permit a classification of the error propagation proneness of a component through analysis of its scope and state.

Last but not least, experimentation will also address monitoring of resource demand to characterize the trade-off between efficiency and fault tolerance enabled by longer or shorter scopes, respectively.

REFERENCES

[1] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture, on patterns and pattern languages*. John wiley & sons, 2007, vol. 5.

[2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013, vol. 2.

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

[4] A. Cockburn, "Structuring use cases with goals," *Journal of object-oriented programming*, vol. 10, no. 5, pp. 56–62, 1997.

[5] P. D. Manuel and J. AlGhamdi, "A data-centric design for n-tier architecture," *Information Sciences*, vol. 150, no. 3-4, pp. 195–206, 2003.

[6] P. C. Linskey and M. Prud'hommeaux, "An in-depth look at the architecture of an object/relational mapper," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 889–894.

[7] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol–http/1.0," 1996.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

[9] D. R. Prasanna, "Dependency injection," 2009.

[10] M. Seemann, *Dependency injection in. NET*. Manning, 2012.

[11] A. Sabot-Durand, "Jsr 314: Contexts and dependency injection for javatm 2.0," 2017. [Online]. Available: https://jcp.org/en/jsr/detail?id=365

[12] V. P. Koutras and A. N. Platis, "Applying software rejuvenation in a two node cluster system for high availability," in *2006 International Conference on Dependability of Computer Systems*. IEEE, 2006, pp. 175–182.

[13] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and analysis of software rejuvenation in a server virtualized system," in *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*. IEEE, 2010, pp. 1–6.

[14] J. Alonso, R. Matias, E. Vicente, A. Maria, and K. S. Trivedi, "A comparative experimental study of software rejuvenation overhead," *Performance Evaluation*, vol. 70, no. 3, pp. 231–250, 2013.

[15] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono, "Software aging analysis of the android mobile os," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 478–489.

[16] J. Xiang, C. Weng, D. Zhao, A. Andrzejak, S. Xiong, L. Li, and J. Tian, "Software aging and rejuvenation in android: new models and metrics," *Software Quality Journal*, pp. 1–22, 2019.

[17] V. Sundaram, M. T. Creti, R. K. Panta, and S. Bagchi, "Component-dependency based micro-rejuvenation scheduling," in *Fast Abstract in the Supplemental Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Anchorage, Alaska, USA*. Citeseer, 2008.

[18] A. Avritzer, R. Pietrantuono, and K. Trivedi, "Future directions for software aging and rejuvenation research," in *Handbook Of Software Aging And Rejuvenation: Fundamentals, Methods, Applications, And Future Directions*, 2020, pp. 355–362.

[19] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging and rejuvenation: Where we are and where we are going," in *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*. IEEE, 2011, pp. 1–6.

[20] ——, "A survey of software aging and rejuvenation studies," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, pp. 1–34, 2014.

[21] K. Moore and N. Freed, "Use of http state management," RFC 2964, Network Working Group, Tech. Rep., 2000.

[22] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[23] W. Q. Meeker and L. A. Escobar, "A review of recent research and current issues in accelerated testing," *International Statistical Review/Revue Internationale de Statistique*, pp. 147–168, 1993.

[24] L. A. Escobar and W. Q. Meeker, "A review of accelerated test models," *Statistical science*, pp. 552–577, 2006.

[25] S. Limon, O. P. Yadav, and H. Liao, "A literature review on planning and analysis of accelerated testing for reliability assessment," *Quality and Reliability Engineering International*, vol. 33, no. 8, pp. 2361–2383, 2017.

[26] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. IEEE, 2000, pp. 417–426.

[27] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[28] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[29] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *16th International Reliability Physics Symposium*. IEEE, 1978, pp. 33–40.

[30] ——, "Alpha-particle-induced soft errors in dynamic memories," *IEEE transactions on Electron devices*, vol. 26, no. 1, pp. 2–9, 1979.

[31] F. F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2017, pp. 169–176.

[32] A. Azizimazreah, Y. Gu, X. Gu, and L. Chen, "Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs," in *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2018, pp. 1–10.

[33] M. Kishani, M. Tahoori, and H. Asadi, "Dependability analysis of data storage systems in presence of soft errors," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 201–215, 2019.

[34] F. Cerveira, R. A. Oliveira, R. Barbosa, and H. Madeira, "Evaluation of restful frameworks under soft errors," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 369–379.

[35] S. Gojare, R. Joshi, and D. Gaigaware, "Analysis and design of selenium webdriver automation testing framework," *Procedia Computer Science*, vol. 50, pp. 341–346, 2015.

[36] J. D. Ament, *Arquillian Testing Guide*. Packt Publishing, 2013.

[37] J. R. Hat, "Shrinkwrap: Java api for archive manipulation," 2016. [Online]. Available: http://arquillian.org/modules/shrinkwrap-shrinkwrap/

[38] D. Crocker *et al.*, "Standard for the format of arpa internet text messages," 1982.

[39] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[40] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting aging-related bugs using software complexity metrics," *Performance Evaluation*, vol. 70, no. 3, pp. 163–178, 2013.

[41] T. Dohi, K. S. Trivedi, and A. Avritzer, *Handbook of Software Aging and Rejuvenation: Fundamentals, Methods, Applications, and Future Directions*. World Scientific, 2020.

[42] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, 2000, vol. 7.