Unveiling Faulty User Sequences: A Model-based Approach to Test Three-Tier Software Architectures

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

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

ARTICLE INFO

Keywords:
Software Architecture
Dependency Injection
Stateful Components
Software Dependability
Model-Based Testing
Data Flow Testing.

ABSTRACT

Context. When testing three-tiered architectures, strategies often rely on superficial information, e.g., black-box input. However, the correct behavior of software-intensive systems based on such architectural pattern also depends on the logic hidden behind the interface. Verifying the response process is thus often complex and requires *ad-hoc* strategies.

Objective. We propose an approach to identify faults hidden behind the presentation layer. The model-based approach uses an architectural abstraction called *managed component Data Flow Graph (mcDFG)*. The *mcDFG* is aware of the interactions between all layers of the architecture and guides the generation of tests based on different *mcDFG* coverage criteria to identify faults in the business logic.

Method. To evaluate the approach viability, we consider a three-tiered web application and 32 faults. The fault detection capability is assessed by comparing a set of test suites created by following our method and a set of test suites developed by utilizing traditional testing strategies.

Results. The collected data show that the proposed model-based approach is a viable option to identify faults hidden in the logic layer, as it can outperform standard strategies based solely on the presentation layer while keeping the number of test cases and number of interactions per test case low.

1. Introduction

Software architectures, particularly web applications, are pervasive and used to support even complex and intricate processes. Ensuring the correctness of such applications represents a challenge. Such programs are event-centric and interact with complex and only partially predictable environments (e.g., users or other applications) through presentation interfaces that can range from simple command line interfaces to rich graphical user interfaces (GUIs). Moreover, software systems are often developed under the pressure of meeting tight deadlines, resulting often in inadequate testing before the software is released. Due to the multitude of features and limited time available, testers often tend to verify the main functionalities or execute the primary usage scenarios they deem important. However, this strategy prevents the identification of faults that would be exposed by executing secondary paths of the application.

To face this challenge, various strategies have been proposed throughout the years. For instance, fuzzy testing [31, 26] exercises the system under test by subjecting it to a series of external events. Other approaches produce test cases following the principles of evolutionary algorithms [4, 30].

On the other hand, exploratory testing [23, 22], unlike automated testing, requires testers to manually select and execute tests by leveraging their knowledge of the internal details of the system.

leonardo.scommegna@unifi.it (L. Scommegna); roberto.verdecchia@unifi.it (R. Verdecchia); enrico.vicario@unifi.it (E. Vicario)

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

Among the plethora of strategies, model-based testing [48] emerges as one of the most popular techniques. The tertiary study by Garousi et al. [18] serves as a testament to this popularity. When compared to other methods, model-based testing ranks first in terms of Google hits, with the second most popular method garnering only half as many search results. In addition, from a more academic point of view, model-based testing holds the second position in terms of number of software engineering secondary studies conducted on the topic [18].

Model-based testing utilizes models to guide the creation of test cases and the execution of tests. Specifically, model-based testing techniques aim to identify a model of the system that represents the relevant specifications and mechanisms of the system under test while disregarding unnecessary information.

By using the model of the system, test cases can be derived systematically, covering various scenarios and ensuring thorough test coverage [48]. Since the model is an abstraction of the real system, the information it carries directly influences the type of test cases that will be generated and the type of faults that can be detected. A fine-grained model will indicate test cases that focus on low-level mechanisms, ignoring the overall functioning of software-intensive systems. On the other hand, a coarser-grained model will indicate test cases that focus on high-level features, abstracting away the internal structure of the system. Software architectures are often divided into three layers characterized by specific functionalities: the *presentation* layer, the *business logic* layer, and the *persistence* layer [16]. The presentation layer manages the external interface of the system. The presentation layer is also responsible for forwarding a request to the business logic layer each time an external event is experienced on the interface. The business logic layer is responsible for (i) leading the response process in reaction to specific requests, (i) implementing navigation logic, and finally (iii) managing transient data related to sessions (commonly known as session state [16]). In the context of this study, we focus our research endeavors on *three-tier layered architectures*, i.e., sotware-intensive systems architected by adopting the classic multitier architectural pattern, composed of a *presentation tier*, a *logic tier*, and a *data tier* [6]. During the response phase, if it is necessary to persist data, the business logic interacts with the persistence layer.

The functional and technological differences between the architectural layers require specific testing techniques. For example, database testing [35] focuses on ensuring that data persistency occurs as expected, while front-end testing [27, 25] verifies the functionality of the interface. However, testing the correctness of individual layers in isolation is not enough to verify the overall correctness of the system and it is often necessary to verify the effects of component collaboration. In particular, the response process and the business logic layer interactions with the other layers are crucial for the overall functioning of the system and have not been appropriately investigated in the literature. In fact, the response procedure involves multiple actors and aspects that usually remain partially-hidden also to the developers. More in-depth, the response procedure is managed through internal components of the business logic, sometimes referred to as *software components*, which live in memory for a number of consecutive requests that cannot be predicted in advance. Software components are stateful since they maintain the session state and during the response process they behave and interact with each other depending on their state. The stateful nature of business logic and the variable composition of software components outline an *evolution* of the business logic among multiple requests. The evolution of the business logic, in turn, implies that the response procedure to a request may depend not only on the current request but also on the history of previous requests. An interdependence is then outlined between the business logic and the sequence of external stimuli to which the system is subjected. Given the tight coupling with external events, predicting the evolution of the internal state is difficult and often unfeasible. The problem is further amplified by the fact that the software components are not managed manually but orchestrated by *Inversion of Control containers* (IoC containers) [15], which handle their lifecycle and dependencies (dependency injection) [32].

In this work, we provide a model-based testing strategy aimed at verifying the correct functioning of the business logic of a software architecture. To achieve this, we formalize a model that exploits proper coverage criteria sometimes termed model-flow criteria [45]. This model is capable of identifying sequences of external events that induce specific evolutions and behaviors among the software components. We then show how this abstraction can be obtained automatically by exploiting a generation toolchain that we have developed specifically for the purposes of this paper. Finally, we conduct an experimental proof of concept of the proposed approach on a web application, named *Flight Manager*. We generated a set of test suites for Flight Manager following different coverage criteria. Subsequently, we evaluated whether the generated test suites are able to detect business logic-related faults through a process of mutation testing where we manually injected 32 non-trivial faults into the application and we assessed the fault detection capability of each test suite. To evaluate the viability of our approach, we apply it in the context of a widespread application of three-layered architectures, namely web applications. In this context, to compare our approach, we consider the baseline presentation layer-centric approach that either utilizes as the coverage criterion visiting all pages (page testing) or visiting all navigation (hyperlink testing) of the navigation diagram [40], also referred to in this work as "Page Navigation Diagram" (PND) [24].

This navigational model is often used in literature to identify feasible navigational paths in system testing. For example, Biagiola et al. [7], use a navigational model to identify feasible sequences of interactions in the system that will then constitute the tests. Zheng et al. [51] propose an end-to-end testing framework based on reinforcement learning to identify high-quality interaction sequences, basing the algorithm's choices on a navigational model. Similarly, Mesbah et al. [34], propose a crawler that works with a page navigation diagram for user interface validation.

The main contributions of this research can be summarized as follows:

1. A catalog of faults (i.e., a *fault model*) that may be introduced at coding time while configuring the IoC container, particularly when specifying dependency injection and lifecycle management of software components;
2. A system abstraction called *managed component data flow graph* (*mcDFG*) that takes into account the dynamic evolution of software architecture;
3. The identification of a toolchain that allows the abstraction of the system to be obtained with minimal effort;
4. An implementation of the *mcDFG* Generation for Java-based systems;
5. A complete replication package of the study[1], including, (i) the implementation of the experimental proof of concept, (ii) a reusable experimental subject for model-based testing containing 32 non-trivial faults, (iii) the complete material required to replicate the study, and (iv) the results of the experimental proof of concept reported in the study.

The remainder of the paper is structured as follows: Section 2 outlines the background information on which the study is based. Section 3 discusses the related work. Section 4 presents the set of chain of threats fault types and failure modes considered in this research. Section 5 presents

[1] https://doi.org/10.5281/zenodo.10727674 Accessed 29th February, 2024

the model-based approach introduced with this paper. An experimental poof of concept of the approach is documented in Section 6, accompanied by the presentation of the collected results and their discussion. Finally, conclusions, implications, and research outlook are reported in Section 7.

2. Background

We describe how transient data are managed through *software components* that live in memory (Section 2.1) while software architectures run. We outline the responsibilities and the operations of *IoC containers* (Section 2.2). We then propose a visual representation that makes explicit components dependencies hidden by this practice (Section 2.3). We finally outline pitfalls entailed by these mechanisms (Section 2.4).

2.1. Software Components

Software architectures often deal with big amounts of data. Different natures of information require different management strategies. Long-term and consolidated data are persisted in a database and identified as *record state* [16, 9]. Conversely, transient data are conveniently stored in memory improving access performance and avoiding burdening the database with volatile information. While the record state is visible among multiple sessions, in-memory information is often identified as *session state* since it is usually related to a single business transaction, i.e., session, and it is not shared among other parallel sessions.

Concretely, the session state is managed by typed software objects that live in memory. In the rest of the paper, we will identify all the objects that live in memory with the term *components*. Components live in memory for a bounded timespan and individually maintain a portion of the session state. Part of living components is also responsible for reacting to external events and possibly providing the proper response. Components that are involved in the response process are usually identified as components belonging to the *business logic* layer [16, 8]. Usually, events consist of external stimuli, e.g., interactions, executed on the interface of an application. The stimulus is forwarded by the presentation layer, the module responsible for interface management, to the business logic layer in the form of a request. Once arrived at the business logic layer, a specific component called *controller*, will intercept the request and conduct the response process. The number of controllers and the criterion of request interception may vary. For instance, in web applications [9, 44], the page controller pattern [16] is frequently used. For each page of the web application, the page controller pattern requires the definition of a controller often referred to as *page controller*. A page controller of a specific page is responsible to intercept all the requests arriving from the related page.

Controllers are rarely independent: during the response process, they need to be supported by other components. In case of the necessity of specific business logic functionalities or to simply maintain information in memory, the controller will interact with other business logic components, here identified as *helper* components. Conversely, when read-/write operations to the database are required, components belonging to the underlying *data layer* will be used. Data layer components, therefore, are responsible to implement the functions and provide the entry points to interact with the persistence medium usually identified by databases. Regardless of the types of components, an interaction stipulates a relationship between the involved components that can influence and tie their states. An interaction then establishes a *dependency* between the embroiled components.

Maintaining the session state requires components themselves to acquire a transient nature. Since it is plausible that some transient information should remain longer than others, the life cycles of the components should not be synchronized, identifying components with different life spans. Thus, to properly satisfy the nature of the session state, the business logic is constituted by a set of stateful components that live *concurrently* for a bounded sequence of requests, and establish dependencies with each other. Since requests arrive at runtime and depend on external factors (e.g., the interactions on the interface) the *evolution*, intended as the composition of the living components and the dependencies established at runtime, are hard to predict at static time.

2.2. Component Management through the IoC Container

The complex and intricate nature of business logic makes its development cumbersome and error-prone. To ease the task, development heavily relies on widespread frameworks that provide high-level *containers* able to manage components at runtime. More specifically, containers run alongside the application and perform *component dependencies injection* (DI), and *automated life cycle management* activities. For this reason, they are usually identified as Inversion of Control Containers (IoC Containers) [15]. Component dependencies injection consists of obtaining on the fly the instance of the type specified at coding time and injecting its reference in the dependent component. This also implies dealing with race conditions and determining dynamically if a specific instance should or should not be shared with other instances. Automated life cycle management takes care of component creation and destruction according to the life cycle models specified by the SW developer at development time.

Container behavior is configured through annotations extending the plain code definition of classes with meta-information. To properly manage components and their evolution, the container maintains a runtime representation based on the concepts of scope and context. A *scope* defines a type of policy that the container can enforce in the lifetime and visibility management of required components. Besides, a *context* maintains a collection of references to running objects, often termed *contextual instances*, managed under a common scope. During the runtime, the container maintains a set of contexts and each managed object is associated with a scope specified by the object type.

Though with various terminologies, scopes are traditionally of four types: *request*, *session*, *application*, and *enclosed*. Components with *request* scope are allocated and maintained in memory only for the time between an interaction request and the response. In Web Applications, for instance, scopes are naturally shaped by concepts of the underlying HTTP protocol and its State Management Mechanism [5]. Thus, in web applications, request scoped components live for the equivalent of a single HTTP request. Besides, components with *session* scope maintain their state along a single session of usage. For Web applications, session scoped components will live among multiple HTTP requests, spanning from the initial contact (or login) to when the application is left (possibly with a logout). In the opposite direction, the *application* scope encompasses multiple sessions, along any long-term run from an application startup to shutdown. However, in many interaction scenarios, like use case scenario executions, data need to be maintained along a time span shorter than an entire session but longer than a single request. This is commonly supported by a scope, which we term here *enclosed*, whose boundaries are programmatically demarcated in the code by explicit begin/end operations. In addition to the four traditional scopes, we also consider another scope not always implemented in frameworks of DI and lifecycle management and that is often identified as a pseudo-scope. This pseudo-scope guarantees that a required component assumes the scope of the dependent component where it is injected. We call it *conforming* scope, in contrast with all the other mentioned scopes which will be termed *absolute*.

The system of scopes is organized hierarchically: a *request* context is always contained in a *session* and possibly in an *enclosed* context. An *enclosed* context is always wrapped in a single *session* context and the *application* context wraps all the *session* contexts. Finally, since managed components have a lifecycle, frameworks usually provide the possibility to define *post-construct* and *pre-destroy* actions for each component triggered, respectively, immediately after the creation and immediately before the destruction of the contextual instance.

2.3. A Visual Representation of Concurrency and Coupling among SW Components

IoC Containers orchestrate the execution of multiple concurrent contexts and contextual instances. The container orchestration is determined by the static scope assigned to required components at coding time and by the sequence of requests received at runtime. In this concurrent execution, managed objects belonging to different contexts interact with each other through method invocations that result in implicit data flow coupling.

Figure 1 provides a visual aid to gain concrete insight into how the high-level concepts of presentation, logic, and data tiers considered in this research translate to the more concrete implementation notions used by the approach. As starting point, data needs to be transferred from the presentation tier to the logic tier. This is depicted in Figure 1

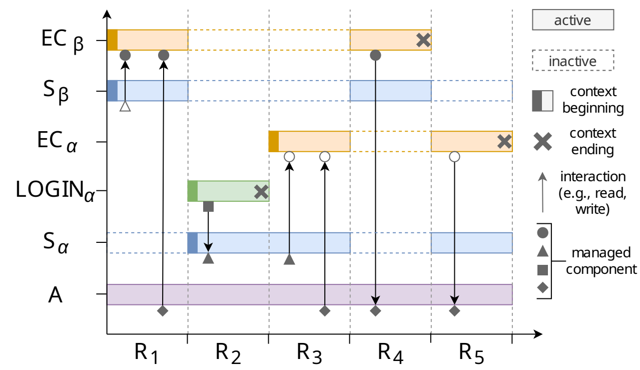


Figure 1: Visual representation of components during a software architecture execution. In the first shown epoch R_1, the application context A continues from the previous activity, the session context S_β and an enclosed context EC_β are started, and the ● instance within EC_β is written, first by △ of S_β and then by ◆ of A. In the subsequent epoch R_2, the session context S_α and a request context $LOGIN_\alpha$ are started, and the instance ▲ of S_α is written by the instance ■ of $LOGIN_\alpha$. Component instances from both EC_α and EC_β, perform read/write operations on shared data from/to the *application* context A.

as the instantiation of managed components at each epoch R_1-R_5, corresponding respectively to the shapes △ and ◆ (R_1), ■ (R_2), ▲ and ◆ (R_3), ● (R_4), and ○ (R_5). The response routine is then executed in the logic tier through the instantiation of contexts and their interaction, depicted in Figure 1 as coloured rectangles. During each epoch, managed components belonging to different context interact between them (see directed arrows in Figure 1). Managed components can belong to two different tiers, either the logic tier (e.g., function calls) or the data tier (e.g., raw data passed from one context to another).

In Figure 1, time is partitioned in a discrete sequence of *epochs*. Each epoch starts when a request arrives and terms when the request is served and the service of the subsequent request can be started. In Figure 1, epochs $\{R_n\}_{n=1}^{N}$ are represented on the horizontal axis.

Within each epoch, the run is characterized by: *i)* the set of *living contexts*, either active or inactive, *ii)* the set of *contextual instances* associated with each context, and *iii)* the *sequence of method invocations* among contextual instances. In Figure 1, living contexts are stacked along the vertical axis. A is the application-scoped context, S_β and EC_β are a session-scoped and an enclosed-scoped contexts, respectively. During R_1, the contextual instance ◆ is associated with context A, △ with S_β, ● with EC_β. Finally, methods of ● are invoked first by △ and then by ◆;

At the beginning of each new epoch, each living context is either *started* (e.g. EC_α in R_3), *continued* (e.g. S_α in R_3 or EC_β in R_3), *inactivated* (e.g. EC_β in R_2) *re-activated* (e.g. EC_β in R_4), or *released* (e.g. EC_β in R_4). At *release*, all instances in the context are destroyed and their state is lost. At *inactivation*, instances maintain their state but they are not visible until the context is activated again. At *creation*, the context starts as new so that each instance will be created from scratch when required. At *continuation*, instances maintain their state and visibility.

Within each epoch, multiple contexts of the same type may be *alive* but only one context for each scope can be *active*. According to this, any *active* contextual instance will be able to directly interact only with instances belonging to its context and its embedding contexts, both of higher and lower level, respecting the hierarchical organisation fixed by definition (i.e., it can only interact with other *active* instances). Since the *request* scope belongs to the lowest level of the hierarchy, for each epoch, the abstraction identifies the set of visible contextual instances and makes explicit the lifetime along which they have maintained their state.

The visual representation makes explicit two interacting mechanisms of cross-context coupling among managed components. On the one hand, concurrent instances active in the same epoch may invoke each other, yielding *direct* coupling between components, both in the same session (e.g. ▲ *intra-session* usage of ● during R_1) and between a session component and the Application context (e.g. ◆ *inter-session* usage ● during R_1). On the other hand, components that maintain their state across multiple epochs may carry *indirect* dependencies between components even when these are not concurrently alive (e.g. ● and ○ transitive coupling intermediated by ◆).

2.4. Problem Formulation

The reaction to an external input of a three-layered architecture is influenced not only by the current request, but also on the past interaction history with the system. In other words, three-layered architectures showcase a stateful behavior, where outputs are provided based on the current interaction with the system, and the internal state reached by the system during its runtime evolution based on the past inputs received. The dynamic nature of this internal state is in fact conditioned also by business logic mechanisms and middleware functionalities, such as Dependency Injection (DI) and automated lifecycle management frameworks (refer to Section 2.2). Within this context, in this research, we aim to identify and evaluate failure-inducing interaction sequences by considering not only the current state of the presentation layer, but also the internal state embedded in the logic and data layers. As further detailed in Section 4, this point of view allows to identify a set of failures that cannot be otherwise identified by considering exclusively the presentation layer state.

Due to components, software architectures can not be considered memoryless systems. It is not guaranteed that a response depends only on the type of request issued and its parametric values. The response process may depend on the transient information encapsulated in one or more components living at the moment of the request. The software architecture can be indeed considered as a stateful system where the *state of the system* is the set of components currently living in memory. To evaluate a stateful system properly, it is fundamental to test it under various state configurations. However, the evolution that the system state experiences at runtime makes it insufficient to test just the state configurations.

More in detail, in software architectures, the system evolves its state in reaction to the events that occur over time. During the usage, it is expected that the transient data required to support the session (i.e., the session state) will change. In software architectures, it is fundamental then to guarantee also that the state of the system will evolve coherently with the sequence of requests that will receive. Even if the system is proven to behave correctly under all the possible state configurations, a wrong evolution during a sequence of requests will still cause a malfunction.

The evolution of the system state represents a fragile part of the architecture. It is guided by configurations defined during the implementation of the architecture but the actual implications can be observed only when the whole application runs. The fragility is further emphasized when DI and automated life cycle management frameworks are used. The actual process of dependency injection of components and the management of their life cycle is managed by the container with logic that remains hidden to the developer that simply exploits the framework. The opacity with which the container works tends to complicate the ability to predict the evolution of the system and to increment unexpected evolution patterns.

Additionally, containers rely on high-level events e.g., in web applications events are HTTP requests while in desktop applications are interactions on the user interface. The high level of the events prevents the standard techniques of unit and integration testing from being used to evaluate how the state evolution affects the runtime behavior. However, neither standard system testing is usually enough. Hence, system testing techniques, usually identify the sequence of events relying on external information provided by the presentation layer. Similarly, also the test case evaluation phase is based on the visible side effects observed considering the system as a black box. In system testing then, the evolution of the system state is only evaluated indirectly and partially, ignoring completely living components, their interactions, and the effects of the container.

Neglecting internal information makes the testing process inefficient for two main reasons. Relying only on external information may lead to selecting poor test cases that neglect the internal processes of both the business logic and the middleware technologies. Moreover, a black box perspective allows assertions only on the external interface of the system under test. This prevents the immediate identification of internal errors and allows the detection of malfunctions only when propagated up to the presentation layer. Even in the case of a test detecting a failure, the subsequent fault detection procedure may be particularly complex due to the complex chain of faults, errors, and failures involved, like for Mandel- and Heisen-bugs [20, 13, 11].

3. Related Work

In this section, we discuss the scientific work related to this study. Specifically, we focus on the closest related work to the approach presented in this study by considering black-box testing strategies (Section 3.1), white- and grey-box testing strategies (Section 3.2), mobile testing strategies (Section 3.3), and Diversity-Based Test Case Selection Strategies (Section 3.4).

3.1. Black-box Testing Strategies

Numerous research studies have been conducted over the years to identify sequences of interactions to exercise the presentation layer of software architectures. Many of these, including ours, rely on an abstraction of the system under test to extract a sequence of relevant interactions. For instance, the work of Biagiola *et al.* [7] proposes a method to generate system-level test cases in web applications. The testing strategy is based on a navigational model of the application where a path represents a list of pages visited by the user during a specific sequence of interactions. Biagiola *et al.* propose a strategy to select the test case on the navigational model guided by a diversity-based metric in order to generate a test suite as heterogeneous as possible. However, the metric proposed by the authors takes into account only the diversity of the interactions involved in the test neglecting the state of the system and its evolution during the execution.

Yousaf *et al.* [50] instead propose an automated model-based test case generation strategy. The process is based on identifying sequences of interactions to be performed on the interface of the application under test. The selection of paths on the interface relies on a model expressed using the Interaction Flow Modelling Language (IFML) formalism [17], a language adopted as a standard by the Object Management Group (OMG), which allows defining the design of web application interfaces. The work presents an interesting application of model-based user interface test case (MBUITC) generation. However, although IFML allows defining some behaviors of the business logic and thus representing dependencies between software components, the lifecycle and role of the container cannot be represented. Therefore, the test cases suggested by the method cannot take into account the faults identified in this work.

3.2. White- and Grey-box Testing Strategies

Previous work of Arcuri [4] has addressed the automatic test case generation for RESTful APIs. The strategy to generate test cases exploits an automated white-box testing approach. Tests are generated through an evolutionary algorithm guided by code coverage and fault-finding metrics. The approach also deals with the well-known hurdle of setting the initial state for test cases. Thus, a test case may require an exact state configuration to observe a specific behavior during the test execution. Setting the initial state of the system is sometimes hard and Arcuri solves the problem through *smart sampling*, a strategy that relies on a predefined set of test case templates. However, smart sampling considers only long-term and consolidated data (the record state) and ignores the transient state of the system. Our approach instead, aims to find a sequence of requests that bring the initial state of the system to the proper one also taking into account the transient data maintained in software components.

The work of van Rooji *et al.* [42] proposes a grey-box fuzzer aimed to discover vulnerabilities in web applications. As in our work, the goal of van Rooji *et al.* is to generate test cases that evaluate the system beyond what is observable in the application response while maintaining a tradeoff in the scalability of the approach. As in our approach, the method is guided by coverage criteria on a high-level representation of the system, however as also outlined by the authors, the faults studied are surface-level bugs. Considered faults in fact do not rely on *"complex internal application state"* or on a series of dependent requests to be triggered.

3.3. Mobile Testing Strategies

Mobile testing, and in particular Android testing, addressed extensively the problem of selecting sequences of interactions to test the correct behavior of the system [28, 2, 47, 37, 21, 29]. Among all the above mentioned papers, the work of Gu *et al.* [21], is very close to our method. The authors propose a fully automated Model-Based automated GUI testing technique. The test case selection is guided by an abstraction that is gradually refined with dynamic information about the system. The dynamic nature of the model allows the method to take into account behaviors that cannot be extracted statically from the application. However, the abstraction can extract and dynamically adapt to behaviors visible from the user interface, this prevents the abstraction from taking into account the evolution of the internal state and suggests paths targeted to trigger the fault that we address in this work.

3.4. Diversity-Based Test Case Selection Strategies

Many other works have addressed the problem of reliability in systems subject to sequences of external events, even without system abstractions. One of the researches that is most closely related to this work is constituted by the Route tool [27]. Route implements a novel strategy of augmentation for system test cases. Starting from a test case consisting of various interactions on the interface, Route suggests alternative cases that verify the same functionality as the original but follow a different path. Taking a different path has the capability to stimulate different dependencies among the underlying components, thus inducing a distinct evolution of the system's internal state.

Although the work remains intriguing and presents innovative heuristics for test case augmentation, the strategy remains blind to internal logic and relies solely on external information, unlike our method, which tackles the problem by employing a grey-box approach.

The work of Leveau *et al.* [25] presents a new approach to suggest rare and diverse sequences of interactions during a phase of exploratory testing of web applications. Although the approach is very interesting and in principle also effective in identifying faulty sequences, the approach measures the diversity and the rarity of a sequence ignoring the internal logic of the application itself. In our method instead, the sequence selection is heavily based on software components information and behavior.

4. Chain of Threats Fault Types and Failure Modes

We characterize the chain of threats affecting the development of the business logic of software architectures by classifying types of coding *faults* (Section 4.1) and *failure modes* that they can produce (Section 4.2).

4.1. Fault Model

We consider a catalog of fault types that can be introduced in annotation or programmatic lifecycle specification, which makes the scope of a managed component unfit for the needs of the point where it is injected.

The catalog was populated by conducting a manual analysis on how the dependency injection and automatic lifecycle management are implemented in the IoC containers. The catalog, therefore, reflects the structural characteristics of annotation-based and programmatic specifications of the lifecycle of software components using IoC containers. To validate the catalog, developers of a complex three-layered architecture implementing an electronic health record in use for several years in a major Tuscan hospital [38, 14] were contacted for feedback. The developers confirmed the list as covering the fault types they experienced in their daily practice.

The resulting catalog of faults considered in this study is reported below.

- **ShorterScope:** a component is assigned an absolute scope *lower* than what would be required.
- **LongerScope:** viceversa, a component is assigned an absolute scope *higher* than what would be required.
- **WrongConformance:** a component is assigned a *conforming* scope while it should have been *absolute*, or viceversa.
- **EarlyOrUndueClosure:** the *end* demarcation of an enclosed context is erroneously added or placed too early in the code.
- **LateOrMissingClosure:** the *end* demarcation of an enclosed context is missing or it is placed too late in the code.
- **LateOrMissingBegin:** the *begin* demarcation of an enclosed context is missing or late in the code.
- **MissingStateClearance:** the code misses a required clear-out or re-initialization of a component, which should be triggered at creation or destruction of some other component as a post-construct or pre-destroy action.
- **ErroneousDynamicInjection:** the type of an injected component is erroneously determined, which may occur when injection types are determined dynamically during the run-time.

The identified faults are insidious and can be inserted by developers with different levels of skill, as can be observed in technical social forums such as *StackOverflow*, *Github*, and *DZone*. An overview of examples of such discussions is reported for completeness in the replication package.

4.2. Failure Modes

Faults in annotation and programmatic specification of managed components lifecycle may result in various kinds of *errors* in the type of injected components or in the logic of the intervals [1] during which they exist, maintain their state, and are shared by multiple dependants. In turn, this may cause various types of deviations in the functional behavior delivered by the presentation layer.

We identify and characterise four types of *failures* occurring when an injected component: does not maintain memory as long as required (*vanishing component*); or, viceversa, it is not renewed when needed (*zombie component*); or it becomes visible at the same time to multiple dependants that should not share it (*unexpected shared component*); or it is created in a wrong type variant (*unexpected injected component*).

Vanishing component. An injected component may not live and maintain its state with continuity along the time interval needed by its dependants, thus resulting in a null pointer exception or a data loss (if the component type is restarted by a new injection), as illustrated in Figure 2.

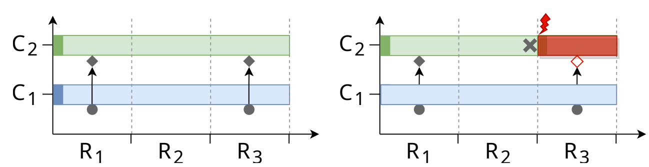


Figure 2: *Vanishing component failure.* (left) the expected correct behaviour in some scenario with two coupled instances ● and ◆ living in distinct contexts C_1 and C_2: ● uses ◆ twice expecting that this maintains its state across subsequent requests. (right) a faulty behaviour: at the beginning of R_3, context C_2 is restarted (instead of continuing) and the IoC container constructs a new instance ◇ of the same component type; the fault is activated at the point marked by ⚡, entering an erroneous state that produces a data loss failure when ◇ is used by ●.

Zombie component. In the opposite situation, an injected component may remain alive with continuity while a dependent component expects that it is destroyed and restarted. This may lead to components that maintain an obsoleted state, as illustrated in Figure 3, or it may also potentially produce an aging failure due to memory leakage [19].

Unexpected shared component. A context may remain continuously active so as to be accessible by two or more concurrent dependent contexts. This may lead multiple dependants to erroneously share the same instance of some required component, causing failures due to interference on the component state, as illustrated in Figure 4.

Unexpected injected component. The type of a required component may be wrongly specified at its injection

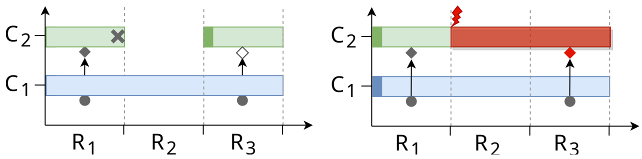


Figure 3: *Zombie component fault.*(left) in a correct implementation, ● should access two distinct instances of ◆. (right) however, since the context C_1 is not closed and restarted, the instance ◆ retains memory also during R_2 and the second access of ● will find an obsoleted and not refreshed state.

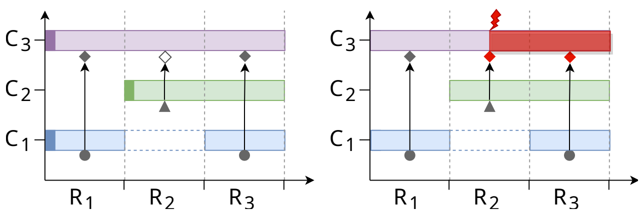


Figure 4: *Unexpected shared component fault.* (left) The ● and ▲ contextual instances expect each one to inject a different instance of the required component (i.e., ◆ and ◇, respectively). (right) yet, the IoC container resolves both dependencies with the same contextual instance, thus producing interference and unpredictable race conditions.

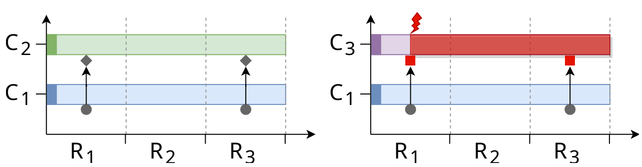


Figure 5: *Unexpected injected component fault.* The IoC container, in R_1 resolves the dependency of ● with a wrong contextual instance (i.e., ■ instead of ◆), thus producing unpredictable behaviours.

point, for trivial coding error or for subtle defects in the static selection of alternative implementations of a type or in the logic of a dynamic programmatic lookup. this may cause a variety of deviations from the expected use case flow, unpredictably leading to fast failure or to complex aging effects [19]. Figure 5 illustrates the concept.

Identified fault and failure types have some typical causal relation, which may direct analysis of root causes: *vanishing components* naturally result from ShorterScope, EarlyOrUndueClosure, and LateOrMissingBegin faults; conversely, a *zombie component* can be easily caused by LongerScope, LateOrMissingClosure, and MissingStateClearance faults; an *unexpected shared component* can be produced by the same faults that cause a zombie component, but with a different process; all failures due to longer or shorter scope can also be due to a WrongConformance, with effects depending on the specific mismatch between conforming and absolute expected components; finally, *unexpected type* typically results from an ErroneousDynamicInjection.

5. Identification of Software Component Faults through Model-Based Testing

We propose a model-based testing approach [48] that jointly involves: (i) the constraints of presentation interface, (ii) the lifecycle specification of software components, (iii) their data-flow dependencies, and (iv) the actual concurrency produced by the effects of container orchestration.

The approach relies on an abstraction, that we call Managed Components Data Flow Graph (*mcDFG* described in Section 5.1). The approach presented in this study is based of a two-phase process, which first involves the *mcDFG* generation, and subsequently generates test cases based on the *mcDFG* model created in the first phase. An overview of the complete process is depicted in Figure 6.

At the highest level the approach, starting from a use case, generates a set of test cases allowing to verify the correct execution of the use case. The presented approach consists of a total of 6 intermediate steps, each one characterized by their own inputs and outputs.

The first phase of the approach, comprising Step 1 and Step 2 (see Figure 6), regards the generation of the *mcDFG* abstraction (see Section 5.2 for more details). In the second phase, starting from Step 3, the procedure exploits the *mcDFG* abstraction to identify and subsequently generate test cases. We describe this latter part in Section 5.3.

Since the *mcDFG* is a technology-agnostic abstraction, the proposed procedure remains valid for generic three-layered architectures with IoC containers. However, for the sake of concreteness and to be able to demonstrate its validity through a proof of concept (Section 6), we implemented the *mcDFG* generation tool for three-layered architectures developed for the Java Enterprise Edition. In the workflow, we have marked both the steps that we have automated and those that we have executed manually. Note, however, that the goal of our proof of concept was to demonstrate the validity of the approach and not to provide a comprehensive tool for practitioners. Thus, we also indicate in the figure the steps that were manually performed during our proof of concept but could easily be automated.

5.1. The Managed Components Data Flow Graph Abstraction

Coverage of couplings across contexts occurring among software components requires a testing approach able to cover the execution paths interconnecting the points where the state of each software component is defined and used. The paths of interest are, therefore, sequences of interactions that occur from the moment a software component is instantiated by the IoC container to the moment a method of the software component is invoked, thus capturing the runtime data flow produced by contextual instances. In principle, execution paths might be abstracted into an *Object-Oriented Data Flow Graph* [46]. However, this would require explicit unfolding and representation of the complex actions performed by the IoC container in the management of contextual instances (e.g., components proxies, aspect-oriented

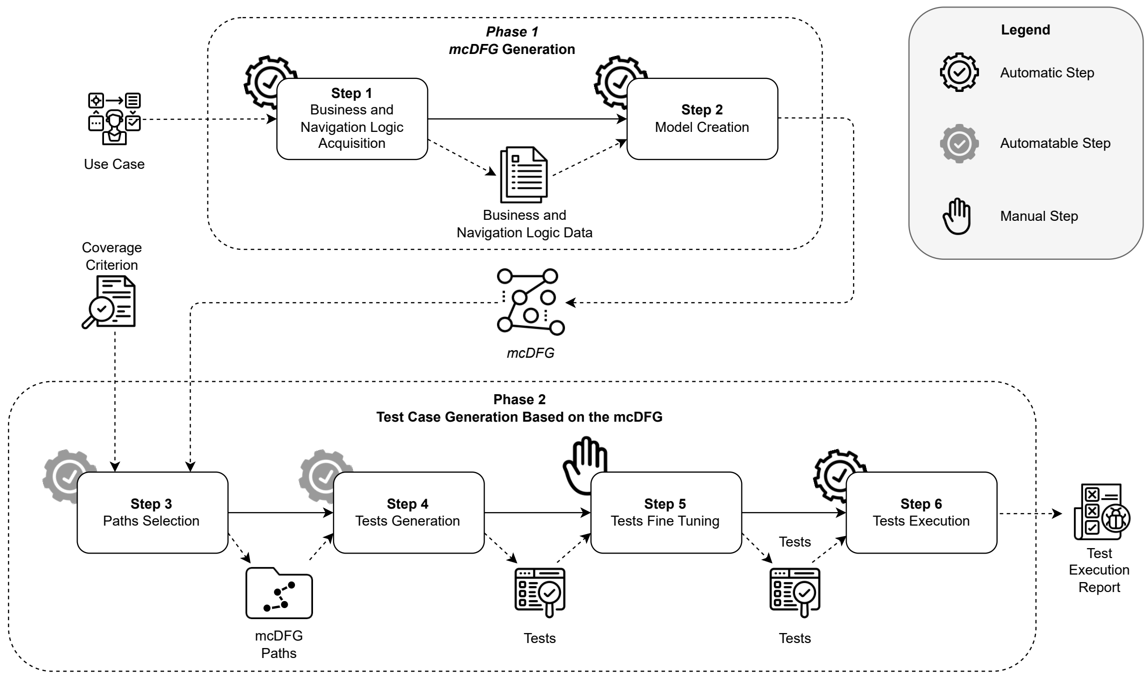


Figure 6: Workflow of the proposed approach

programming techniques), with an explosion of graph elements leading to infeasible dimensions of test suites.

To this end, we propose the *Managed Components Data Flow Graph* (*mcDFG*) abstraction, inspired by the classical DFG and DFT theory [39], which combines elements of structural and functional perspectives by capturing salient characteristics of involved components with their dependency hierarchies and lifecycles together with admissible interactions along designed use cases. Formally, the *mcDFG* is a directed graph, labeled on vertices and edges:

$$mcDFG := \langle \mathcal{V}, \mathcal{V}_{in} : \mathcal{E}, \mathcal{E}_{in}, def, use, \mathcal{P}, Nav, CB \rangle$$

Where $\mathcal{V}$ is the set of vertices, with each $v \in \mathcal{V}$ representing a *basic block*, i.e. a sequence of method invocations and IoC container instantiations that are always executed as a whole. $\mathcal{V}_{in} \subseteq \mathcal{V}$ is the subset of vertices associated with basic blocks that terminate in any state where the interface waits for interactions. $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges, with $< v_i, v_j > \in \mathcal{E}$ iff there exists an execution where the last operation of v_i can be followed by the first operation of v_j. $\mathcal{E}_{in} \subseteq \mathcal{V}_{in} \times \mathcal{V}$ is the subset $\mathcal{E}$ made of the edges that leave a basic block that terminate with the interface waiting for an interaction.

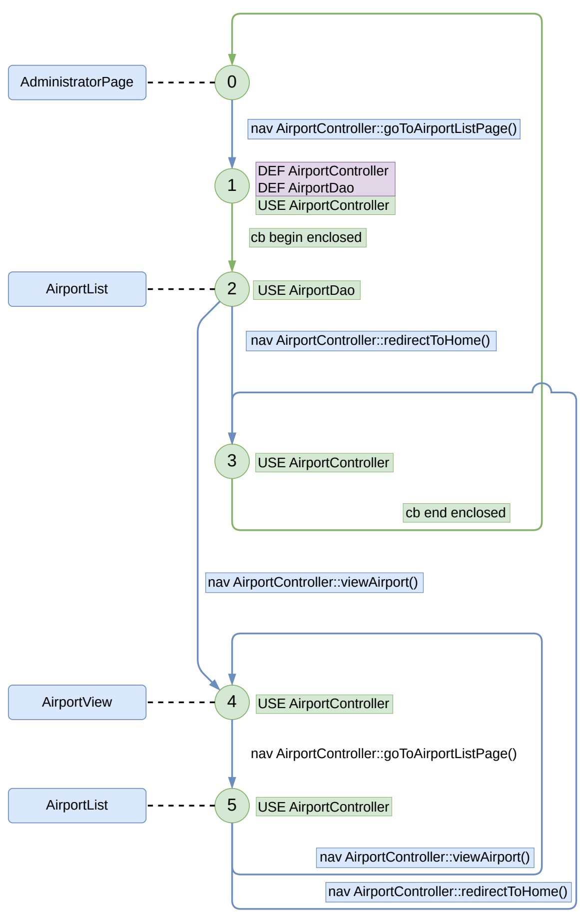
Relations $def : \mathcal{V} \rightarrow 2^{MC}$ and $use : \mathcal{V} \rightarrow 2^{MC}$ associate each vertex with the subset of *used* and *defined* managed components, where MC denotes the set of all managed components, and, for any $c \in MC$, $c \in def(v)$ means that an instance of component c is created during the execution of the basic block associated with vertex v, and $c \in use(v)$ means that an already existing instance of c is used by invocation of any of its methods. As opposed to the classical theory of dataflow testing, the relation of *use* does not distinguish whether the invocation will produce a side effect on the used component. Besides, the relation $\mathcal{P} : \mathcal{V}_{in} \rightarrow$ *presentation layer states* associates each vertex $v \in \mathcal{V}_{in}$ with the specific interface provided by the presentation layer on completion of its associated basic block. The presentation layer state identifies the set of interactions currently allowed on the presentation layer.

The relation $Nav : \mathcal{E}_{in} \rightarrow \{nav\,controller :: sign()\}$ associates each edge $\epsilon \in \mathcal{E}_{in}$ that exits from a vertex $v \in \mathcal{V}_{in}$ with the controller method triggered by the interaction $sign()$. The relation $CB : \mathcal{E} \rightarrow EnclosingActions$ associates edges with any programmatic action of control of an *enclosed* context performed when the edge is traversed, with $EnclosingActions = \{begin, end, end/begin\}$.

To exemplify the concept, Figure 7b reports the *mcDFG* derived from a use case implemented in the online flight booking system *Flight Manager* (more details in Section 6.2). Vertices, associated with basic blocks, are represented as green circles and they are labeled with def and use operations performed in the corresponding basic block, on violet and green background, respectively (e.g. see, vertex 1); vertices in $\mathcal{V}_{in}$ (e.g. vertex 5) are also associated with a pale blue label with the identifier of the presentation layer state in which the three-layered architecture waits for an interaction; output edges from $\mathcal{V}_{in}$ vertices are labeled with the name of controller methods triggered by an interaction (e.g. from vertex 5, AirportController::viewAirport() and AirportController::redirectToHome()) actions for programmatic control of enclosed contexts are labeled on edges where they occur (e.g. on edges $\langle 1, 2 \rangle$ and $\langle 3, 0 \rangle$).

Note that the *mcDFG* is a kind of grey-box abstraction that seams the structure of the navigational model, also known as *page navigation diagram*, of Figure 7a (the pale blue parts) together with lower-level information related to the application code (green parts) and the IoC container behavior (violet parts).

(a) A fragment of the PND of the Flight Manager application.



(b) Managed Component Data Flow Graph.

Figure 7: A snippet of PND and the corresponding *mcDFG* for the administrator use case "View Airports" (UC:A4.2).

5.2. *mcDFG* Generation (Phase 1)

The *mcDFG* provides a powerful abstraction, well tailored to unravel the actual dependencies that result from the intertwined effects of (i) interactions on the presentation layer, (ii) DI specification and method invocations in back-end components, and (iii) orchestration process implemented by the container. However, this effectiveness comes with a corresponding price in the *mcDFG* construction, which involves a significant and error-prone effort for the inherent complexity of integration of different perspectives and for possible misconceptions of the IoC container behavior. Nevertheless, a manual generation process would result time-consuming.

To overcome the hurdle, we resort to a two-phase automatic approach that, starting from a use case will generate the corresponding *mcDFG*.

5.2.1. Business and Navigation Logic Acquisition (Step 1)

The initial input of the presented approach, and hence Step 1, is a user-goal level description of a use case [12].

The first step consists in collecting information related to both navigability and business logic of a use case. To this end, a monitor tool for JEE architectures was implemented to gather effortlessly the required information. More in-depth, the tool operates at runtime and for each interaction issued on the presentation layer, is able to detect (i) the initial presentation layer state where the interaction is performed, (ii) the name and the scope of the components involved in the response process, (iii) and the state that the presentation layer finally reaches.

Concretely, the acquisition process requires that the monitoring tool is executed during the whole execution of the application under test, in order to allow the acquisition of the business logic and data layer runtime information. Once the monitoring setup is in place, the use case needs to be executed *via* the presentation layer of the application under test, and the monitoring tool will observe and collect information on the internal operations. To acquire an exhaustive overview of the underlying logic, this phase requires exercising both the *main success scenario* and their *variations* [12].

The output of this step is a report on the observed response mechanisms of the presentation and business logic layer. This information will be used as input in the next step.

5.2.2. Model Creation (Step 2)

Step 2 merges the information obtained in the previous step - the reachability relationship of the interfaces (i.e., navigability information) and component dependencies - with the details related to the activities of the IoC container in use. This phase represents a crucial part of the *mcDFG* construction: it requires in-depth insights that tend to remain transparent to software architecture developers and that constitute one of the main causes of identified software faults. The identification of *def* and *use* annotations on the vertices of the *mcDFG* indeed requires not only an understanding of the internal workings of the business logic, but also of how the IoC container orchestrates the components (e.g., creation and destruction of contextual instances). In this case, therefore, relying on an automation procedure becomes necessary not only to speed up the *mcDFG* generation but also to ensure a correct result.

As notable features, the tool optimizes the number of final vertexes and implements heuristics that keep the number of cycles as low as possible, with a positive impact on the number of paths that shall then be covered by different test cases.

The output of this step is the *mcDFG* representation of the observed response mechanisms. In addition to the *mcDFG* representation, the next step (Step 3) requires also to specify a coverage criterion (e.g., all nodes), which needs to be manually provided as input (see Section 5.3).

5.3. Test Case Generation Based on the *mcDFG* (Phase 2)

Once the *mcDFG* is obtained through Phase 1, it is used in Phase 2 to identify a set of interaction sequences and construct the tests. The steps composing Phase 2 are described below.

5.3.1. Paths Selection (Step 3)

The *mcDFG* abstraction captures couplings among software component instances under the orchestration of the IoC container according to interactions issued on the interface. Coverage of these coupling comprises a focused and effective means for the identification of faults in annotation-based and programmatic DI specification of back-end components. In so doing, a feasible *mcDFG* path subtends a sequence of interactions on the presentation layer that triggers a specific chain of interactions among software components. We embed a single path in a test case and the set of paths satisfying a chosen coverage criterion in a dedicated test suite. In the following, we provide a suite of criteria inspired to the classical theory of Data Flow Testing [39], while various other coverage criteria could be used as well, e.g., for presentation layers exposing Graphical User Interfaces, coverage metrics such as page or hyperlink coverage could be used, as described in [40].

- **All Nodes** coverage verifies that every reachable basic block is tested at least once, which includes that each *def* (i.e., a component instantiation) and each *use* (i.e., a component method invocation) of any managed component is exercised;
- **All Edges** verifies that every edge is traversed at least once, which implies that each *nav use* from each presentation layer state (i.e., each interaction) is tested;
- **All Defs** verifies that every *def* is tested at least one time, thus exercising each managed component instantiation, reaching one of its *uses* (i.e., one of component method invocations), without traversing intermediate *defs* of the same component;
- **All Uses** verifies that for each *def* all the possible subsequent *uses* are covered, i.e. that: for each component c, and each vertex v_d where c is *def*ined, and each vertex v_u where c is *use*d, *at least one path* that goes from v_d to v_u without visiting any intermediate *def* is exercised;
- **All DU-Paths** verifies that all the possible acyclic paths between each *def* and all its subsequent *use*s are covered, i.e. that: for each component c, and each vertex v_d where c is *def*ined, and each vertex v_u where c is *use*d, *all the acyclic paths* that go from v_d to v_u without visiting any intermediate *def* are exercised.

Criterion	Complexity
All Edges	$\mathcal{O}(N \cdot F)$
All Nodes	$\mathcal{O}(N)$
All DU-Paths	$\mathcal{O}(2^N)$
All Uses	$\mathcal{O}(N^2)$
All Defs	$\mathcal{O}(N \cdot C)$

Table 1
Complexities of *mcDFG* coverage criteria.

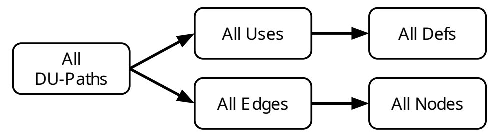


Figure 8: Inclusion relationships among coverage criteria for the *mcDFG* abstraction.

Once a coverage criterion is selected, the approach requires to analyze the *mcDFG* in order to generate a set of *mcDFG* paths that satisfy the coverage criterion. In the proof of concept experimentation (see Section 6) the *mcDFG* coverage of the generated paths is assessed manually. However, in a future implementation of the approach, this process could be automatable by utilizing a graph coverage algorithm.

Inclusion relationships among different criteria are summarized Figure 8. Note that they differ from those of the classical theory of data flow testing in [39] in that *All Uses* coverage does not include *All Nodes* (and not either *All Edges*): in fact, in the *mcDFG*, branching edges from a basic block represent choices in navigation control, not alternative complementary exits of a common guard expression as leveraged in the proof of coverage inclusion referred to the Data Flow Graph in [39].

Theoretical complexity, expressed in terms of the limit number of tests sufficient to implement each criterion, are reported in Table 1, where N is the number vertices in the *mcDFG* abstraction, C the number of distinct managed components, and F the maximum number of choices in the navigation out of any interface within a use case.

The output of this step is a set of *mcDFG* paths that satisfy the selected coverage criterion.

5.3.2. Tests Generation (Steps 4)

To generate the test cases, each path identified in the modeled *mcDFG* (see Step 2), will be translated in a test case, as each path on the *mcDFG* represents a sequence of interactions. The generation of the test is therefore systematically guided by the ordered list of *nav* edges that the path encounters.

Specifically, given the *mcDFG* path, i.e., a sequence of vertexes belonging to the graph, the test instruction of each test case are manually generated by ensuring that all conditions necessary to traverse the vertexes of the graph are met. Given that the test case generation consists of a manual process, the specific technology utilized to implement the test cases is left open by the approach, and depends on the specific development context considered in practice. We note that, building on the presented approach, this step can be automated (see also Figure 6). A test is composed of a set of simulated interactions on the interface of the system under test, which are sequentially evaluated. The evaluation consists of validating the behaviors observable from outside the system (as in the classic cases of black box testing) and the state of the components (i.e., business logic and data persistence layer).

Note that, a test case identified by the *mcDFG* abstraction implies a navigation constraint to verify on the actual implementation and so, step 4 also defines a base oracle, open to be extended by the tester through specific inspections on the state of both the presentation layer and the business logic.

The output of this step is a set of tests covering each *mcDFG* path.

5.3.3. Tests Fine Tuning (Step 5)

Step 5 requires the developer to add, if deemed necessary, additional checks to the tests generated in the previous step. This step is optional, but when combined with the knowledge of the functional requirements of the use case under consideration, it allows for an increase of fault detection capabilities. The *mcDFG* also provides support to the tester in this step. In fact, the *def* and *use* annotations present on the vertexes of the corresponding test path suggest which components undergo side effects and consequently should be checked.

The output of this optional step is the final set of manually tuned tests covering the use case.

5.3.4. Tests Execution (Step 6)

Once the tests are obtained, they can be executed to get the final test outcomes. Given the manual intervention required for the test generation (see Step 4 for more information), the test case execution is not strictly bounded to any specific technology. However, to achieve a good degree of repeatability, tests should be executed automatically. Therefore, it is recommended to implement the tests with technologies that allow automatic execution and subsequent automatic evaluation of the test outcome. For example, in the experimental proof of concept documented in Section 6, test cases were implemented by utilizing Selenium 4.16.1[2] and JUnit 4.13[3].

In the final test execution report provided as output, failing test correspond to triggered failures identified by the approach.

The output of this step is the final test execution report, in terms of tests passed and failed.

6. Experimental Proof of Concept

To confirm the viability of our approach, we conducted an experimental proof of concept to estimate the fault detection capability and assess whether the use of our method provides an advantage over a traditional system testing approach.

During the experimental proof of concept, we are interested in evaluating (i) the fault detection capability of the identified test suites and (ii) the cost in terms of development time that the generation of each test suite implies.

We report results showing how the *mcDFG* provides an effective abstraction for the selection of test cases that are able to: activate faults occurring in the usage of dependency injection and automated management of components lifecycle; and propagate them up to failures in the functional behavior of the presentation layer or in some observable inconsistency of the state of business logic components.

6.1. Research Questions

In order to assess the effectiveness and applicability of the approach, we address the following research questions (RQs):

- RQ_1: *To what extent is our method capable of detecting business logic faults?*
- RQ_2: *How effective is our method in comparison with techniques based on Page Navigation Diagram abstraction?*

With RQ_1 we aim at investigating to which extent the method is able to identify faults of the identified fault model. In particular, we are interested in estimating the fault detection capability of the test suites obtained by applying the different coverage criteria identified. Additionally, we are particularly interested in observing the behavior of the test suites in the presence of non-trivially identifiable faults. By *non-trivially identifiable faults*, we refer to faults that, once activated, do not immediately manifest a failure on the interface.

With RQ_2 we aim to provide a method of comparison with existing strategies. To the best of our knowledge, to date, no literature explicitly targets the correctness of business logic taking into account the evolution inferred over time by sequences of external events and IoC containers. System testing treats the entire architecture as a black box and is unaware of the underneath details of the business logic [36]. However, it subjects the system to sequences of external events and evaluates its functional behavior on the interface. The system test cases thus induce an evolution of the software components and are potentially capable of uncovering failures caused by business logic faults. As a comparison then, we have chosen to rely on a model-based testing strategy based on the Page Navigation Diagram (PND) [7, 24, 34]. The Page Navigation Diagram is an abstraction of the system that is aware of external information (e.g., navigational logic and admissible interactions for each page) but ignores the behavior of the business logic.

6.2. Experimental Object

We conducted our experimental proof of concept on a web application called Flight Manager, developed in-house

[2] https://www.selenium.dev/ Accessed January 4, 2024
[3] https://junit.org/junit5/ Accessed January 4, 2024

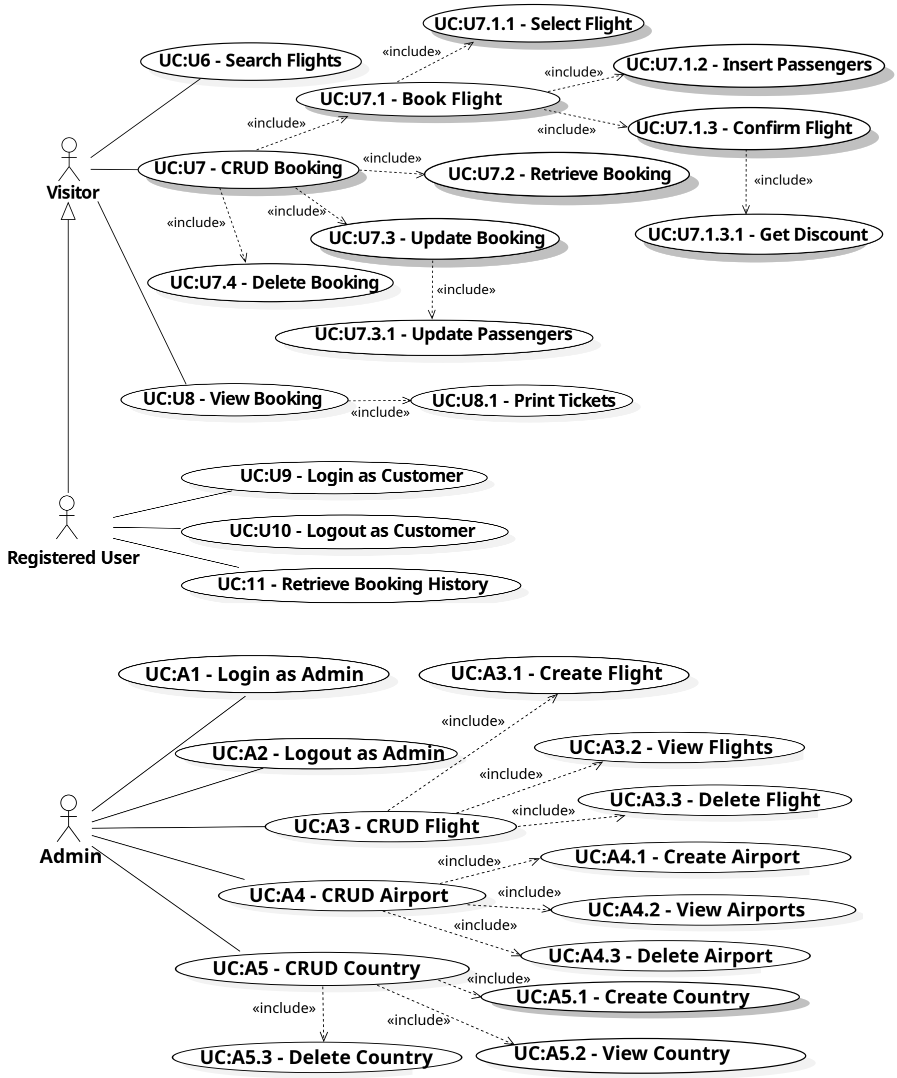


Figure 9: Use case diagrams of *Flight Manager*.

by our laboratory. The research choice of adopting the Flight Manager allowed us to have access to the source code of an enterprise-level application with an adequate number of classes and functionalities. More specifically, to assess the correctness of the proposed approach, we require the experimental subject to (i) leverage dependency injection, (ii) be based on a thee-tier architectural pattern, (iii) explicitly document use cases, (iv) provide a test suite, and (v) be compilable. As the goal of this investigation is to study the theroetical viability of the approach, rather than its generalizability, we focus the proof of concept on Flight Manager, as it results to be an accessible experimental subject satisfying all documented prerequisites while making all source code and related artifacts readily available for scrutiny. Real-world enterprise applications are rarely available as open source, as they often hold economic value for companies, which tend to keep them as proprietary software. The application is made available online for scrutiny and replication purposes as part o the replication package of this study. Specifically, Flight Manager is a stateful web application written in Java and the Java/Jakarta Enterprise Edition Platform. The application focuses on an online flight booking system and, as such, implements use cases common to this type of system (see Figure 9).

As represented in Figure 10, the application follows a 3-tier stateful architecture, consisting of the Domain Model, Data Source, and Presentation Layer. The Domain Model is composed of 10 entity classes. A representation of the domain model in the form of a class diagram is represented in the domain model package of Figure 10. For the sake of conciseness, the class diagram reports only the crucial element of the domain (e.g., no *enum* and *abstact* classes are represented). An exhaustive representation ao the domain model of Flight Manager is available in the replication package. The Data Source is formed by 6 Data Access Object (DAO) which exploits services of an Object Relational Mapping (ORM) framework. The Presentation Layer is made of XHTML pages (roughly, 30 pages), organized as shown in the *Page Navigation Diagram* (PND) of Figure 11. Finally, a Business Logic Layer maintains roughly 30 classes

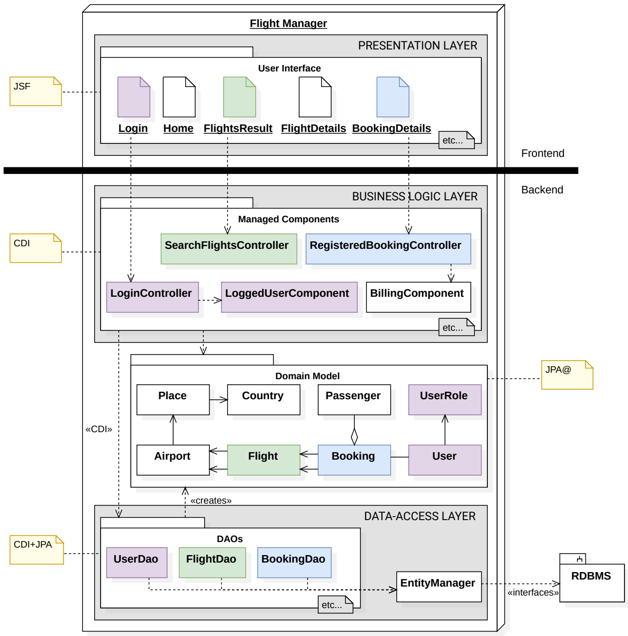


Figure 10: Architecture of *Flight Manager*.

of software components. Flight Manager is composed of 4.6k source lines of code.

6.3. Experimental Proof of Concept Process

To assess the feasibility of our approach, we leveraged the opportunity to access the source code of the experimental subject. Firstly, we constructed the test suites. Each suite is composed of all tests, which are obtained by applying our approach to every use case of the application, using a specific coverage criterion from those indicated in Section 5.3.1. The *mcDFG*s were obtained utilizing an automation tool that was specifically implemented for this proof of concept experimentation (refer to Section 6.6 for more details).

As already discussed in Section 6.1, we aim to compare our method with standard system testing strategies. To do this, we relied on an abstraction frequently used in literature [7, 24, 34], which we identify here with the name of *Page Navigation Diagram* (PND), see Figure 11 as an example. Specifically, this abstraction is concerned with representing the information obtainable through an external analysis of the system, with a particular focus on the acceptable interactions on each individual page and the reachability relationship that exists among different pages. On top of the PND abstraction, we generated two additional test suites exploiting two coverage criteria. Specifically, we considered *All Pages* coverage, which requires that each reachable page is visited at least once, and *All Navigation* coverage, which verifies that each navigation (i.e., each edge of the page navigation diagram) is traversed at least once.

After obtaining the test suites, we estimated their fault detection capability through a procedure similar to mutation testing strategies and we compare the results. More specifically:

1. We create a faulty version of the application, commonly referred to as a *mutant*, using a fault injection procedure.
2. We execute the test suites on the faulty version of the application.
3. For each test suite, we evaluate the final test execution reports. As each faulty version in the proof of concept experimentation corresponded to exactly one mutant, a single test of the test suite fails for that version indicated that the mutant was killed.
4. We define the fault detection capability of a test suite as the percentage of mutants that the suite successfully kills over the total number of mutants considered, namely 32.

The complex nature of faults, their propagation mechanisms, and the laws governing the manifestations of associated failures prevent us from exploiting automated tools for the fault injection phase. Therefore, for this work, faults were injected manually (hand-seeded fault [3]), leading to the generation of 32 faulty versions of the Flight Manager characterized by non-trivial faults.

6.4. Fault Detection Capability Results

Results obtained through the experimental proof of concept are summarized in Table 2. For each coverage criterion, the metrics associated with the corresponding test suite are reported. The "*Avg. # tests per Use Case*" column identifies the average number of tests required to validate a use case provided as input to the approach (see also Figure 6). The "*Interactions per Test Case*" column indicates the average number of user interactions required to complete a test. Lastly, the *Fault Detection Capability* describes the percentage of mutants killed by an abstraction considering a certain coverage criterion over the total number of faults considered, namely 32 faults (see Section 6.3). With these metrics, we are able to assess the quality of our method not only in terms of fault detection capability but also in terms of applicability. In fact, the dimension of the test suite and the number of interactions per test case are two measures that, when considered together, can provide a directly proportional measurement of both the implementation effort and the execution times that the test suite requires.

6.4.1. *RQ_1 Answer (Approach Fault Detection Capability)*

The collected results indicate that our approach is able to successfully identify hidden faults in the business logic. As indicated in particular by the fault detection capability of the test suites obtained with the *mcDFG* abstraction. To explicitly answer the *RQ1*, based on the results of the experimental proof of concept, we can state that the proposed method is capable of identifying faults hidden in the business logic layer. However, we highlight the worst performance of the test suite obtained with the *All Defs* coverage criterion. We explain this as a consequence of the fact that *All Defs* coverage can be implemented by extremely compact paths,

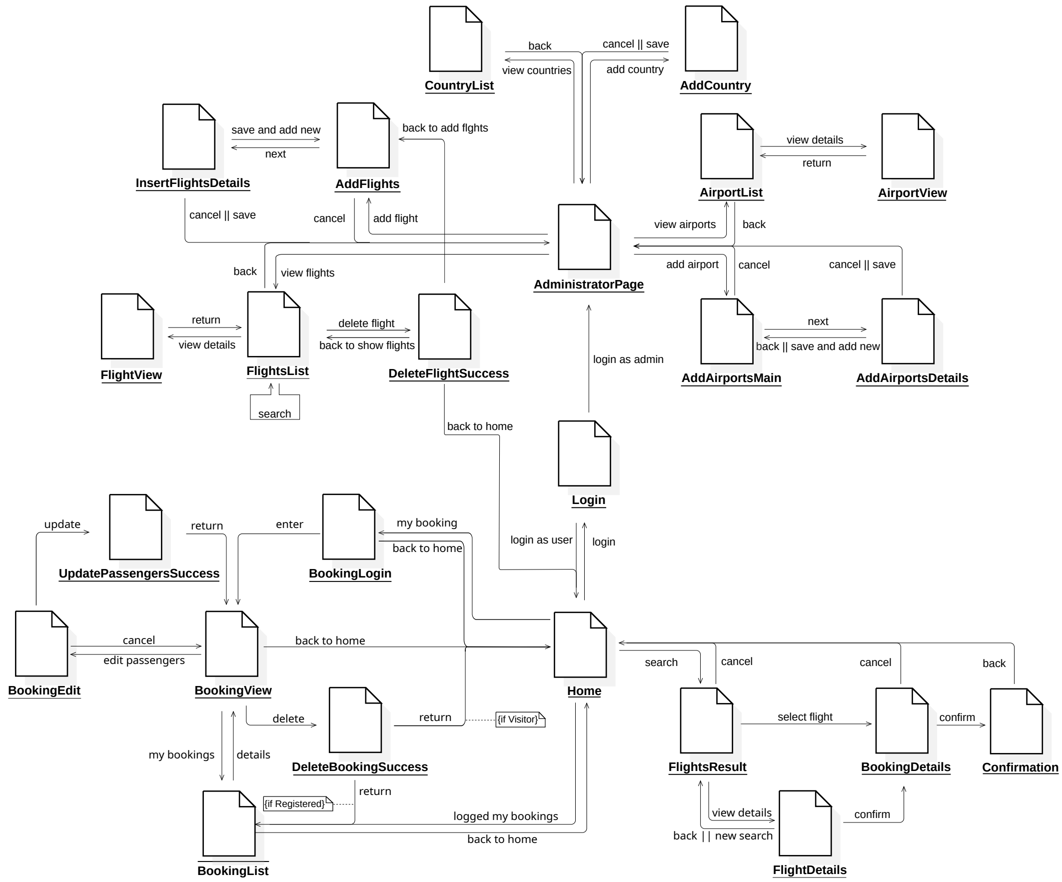


Figure 11: Page Navigation Diagram of *Flight Manager*.

Abstraction	Coverage Criterion	Avg. # Tests per Use Case	Avg. # Interactions per Test	Fault Detection Capability (%)
mcDFG	*All Nodes*	1.18	6.09	100
	All Edges	1.27	9.25	100
	All Defs	1.18	3.09	84.37
	All Uses	2.27	5.04	100
	All DU Paths	3.09	7.76	100
PND	*All Pages*	2	18	28.12
	All Navigation	3	26.33	50

Table 2
Complexity and fault detection of coverage criteria on the 32 faulty versions of *Flight Manager*.

where some component methods may not be exercised at all, as illustrated in Figure 12.

Furthermore, both test suite and number of interactions per test case sizes maintain low values even for expensive criteria, notably for *All DU Paths*. This indicates that the effort required to develop and execute test cases remains low as well. The causes of these low values depend on the high-level perspective of the *mcDFG*, resulting in a sparse graph with a limited number of vertices and edges. Thus the dimension of the *mcDFG* is related by construction to just the number of pages and interactions involved in use cases which is by far lower than what may occur in a conventional DFG expressed in terms of code-level basic blocks.

RQ_1 Takeaways (Fault Detection Capabilities)

Takeaway 1.1: Model-based approaches can successfully identify various faulty interaction sequences in three-tiered layered architectures.

Takeaway 1.2: The high-level perspective of the presented approach allows for the identification of a reduced number of test cases per use case.

Takeaway 1.3: Generated test cases require a low number of interactions with the interface layer.

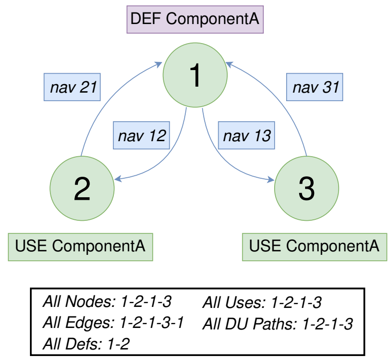


Figure 12: Different coverages on a specific *mcDFG* example.

6.5. Approach Effectiveness Results

Always based on the Table 2, we now want to compare the results obtained with the test suites based on the abstraction proposed by our method (*mcDFG*-based) with the results obtained with the test suites derived from the page navigation diagram (PND-based).

All coverage criteria based on the *mcDFG* show a high fault detection capability, full in most cases, and definitely over-perform test suites based on the *PND* abstraction.

In the comparison of dimensions of *mcDFG* and *PND*-based test suites, the value related to the *mcDFG* represents the average number of test cases needed to satisfy the coverage criterion in a use case, while the *PND*-related is the exact number of test cases needed to test the entire application. In fact, we used each method in its natural way: *mcDFG*-based testing is use-case-wise, as it identifies a different suite for each use case, while *PND*-based testing targets the interface pages of the overall application, which can be covered with a limited number of "long" test cases. However, even if the dimension required to test the *entire* application with the proposed method is still low (the larger test suite is the one related to the *All DU Paths* criterion and consists of 20 test cases), it is possible to include multiple use cases in the same *mcDFG* and then exploit the connectivity between pages to further decrease the test suite dimension. This kind of "*trick*", however, has a drawback: while the number of test cases decreases, due to the redundant navigation actions used, the length of test cases (i.e., the number of user interactions required to carry out the selected navigational path) increases, suggesting that the test suite execution time will not change too much with the use case wise or the application wide approach (see the number of interactions per test case in the Table). As a showcase, we generated an *mcDFG* comprising both the "*search flights*" and the "*book flight*" use cases (*UC:U6* + *UC:U7.1*) obtaining test suites with the same fault detection capability of the two separate diagrams, with overall smaller size longer sequences characterizing each test case (see the details in the repository).

6.5.1. *RQ_2 Answer (Approach Effectiveness)*

Comparing the results obtained with the *mcDFG*-based test suites and those PND-based allows us to answer the *RQ2*. Our method demonstrates to be more accurate in identify hidden faults in business logic in comparison with a Model-Based Testing method aware of only external information. More in detail, the improvement can be explained as due to the ability of the *mcDFG* to extend the purely functional perspective of the *PND* with architectural information, which supports both test case selection and oracle interpretation. On the one hand, test cases identify navigational paths that stress the application not only under the end user functional perspective of page navigation but also under the business logic and IoC container structural perspective. On the other hand, test cases and interpretation of their effects are built so as to be aware both of the user interface and of the business logic components states, enabling detection of a fault even when its propagation does not manifest a failure at the user interface and remains hidden with consequences that are hard to observe and predict [20].

> **RQ_2 Takeaways (Effectiveness)**
> **Takeaway 2.1:** When testing three-tier architectures, considering only the presentation layer does not allow to unveil faulty interaction sequences hidden in the business logic.
> **Takeaway 2.2:** Despite enhanced fault detection capabilities, test suites based on the approach maintain dimensions comparable to those generated *via* plain navigational models.
> **Takeaway 2.3:** Considering interactions between the presentation and logic layers allows for faults to be intercepted even without the manifestation of a failure visible outside the system.

6.6. Applying the Approach in Practice

The presented approach is specifically designed to work with software-intensive systems that are structured using the three-tier architectural design pattern. The amount of work required to adapt this approach for different architectural patterns is uncertain and is not considered within the scope of this study. When applied to other architecture conforming to the three-tier pattern, the approach does not necessitate any prior manual configuration. However, it would require a custom implementation that depends on the specific framework of dependency injection. For Java-based applications using the Context and Dependency Injection (CDI) framework, the proof of concept implementation of the approach, which accompanies this study, can be used immediately without any need for prior implementation or configuration.

Concretely, the tool is a CDI extension. CDI is a popular framework for Inversion of Control and it is the standard for Java/Jakarta EE.[4] Being developed as a CDI extension, the association of the tool with the application is straightforward, as the basic configuration requires only specifying the tool as an extension for the target application. The procedure can be deemed as rather efficient, as it consists only in copying a single plain file inside the metadata directory of the target application. The tool automates the entire Phase 1

[4] `https://jakarta.ee/specifications/cdi/` Accessed January 4, 2024

of the approach (see Figure 6), generating an *mcDFG* output from the input use case.

6.7. Threats to Validity

In this section, we discuss the threats to validity of our study, by following the classification provided by Runeson *et al.* [43] and by considering potential pitfalls of mitigating and documenting threats [49].

1) Construct validity: if the experimental proof of concept we set is appropriate to answer the RQs. To answer to *RQs*, we assessed the fault detection capabilities of various test suites through a mutation strategy. Due to the complexity of the faults, we were unable to rely on automatic tools, and thus the fault injection phase was carried out manually. In principle, defining and injecting manually the faults could potentially influence the estimated fault detection capability: the fault may be not representative or too easy to find for our method. To minimize bias in this phase as much as possible, some faults were proposed by members of our laboratory who were not involved in writing this work. The remaining faults, however, were reproduced by drawing inspiration from real issues about software components reported in technical social forums (*e.g., StackOverflow* and *GitHub*) by developers with different levels of experience and different expertise in language and frameworks. A collection of posts on technical social forums that testify to the difficulty of using IoC containers is reported in the replication package.

2) Internal validity: if the observed results are actually due to the "treatment" and not to other factors. Our experimental proof of concept is conducted on a web application developed in-house for this purpose. Exploiting an application that is not actually used in practice could be an unrealistic assumption.

To mitigate this threat, however, Flight Manager has been developed by software professionals with strong and consolidated experience, following disciplined software development practices. Additionally, Flight Manager implements a widespread combination of reference architectural patterns, largely documented in the professional literature [41, 33, 16], and developed using a language and technology stack (Java and JEE) with primary impact and spread in the practice of complex web applications.

3) External validity: whether and to what extent the observations can be generalized. The results we obtained are derived from an experimental proof of concept that considers a specific architectural style and technology. The results obtained may not be the same on other systems. To mitigate this threat, this work did not rely on a specific technology, instead, it required an analysis of the most popular frameworks that provide IoC containers in *Java, C#*, and *Python* languages. The analysis led to the identification of 5 generic scopes: *request, enclosed, session, application*, and *conforming* (Section 2.2) and the definition of a fault model on which our method is based (Section 4.1). As a reference, Table 3 enlists types of scopes supported by major frameworks analyzed.

Moreover, we have attempted to maintain also our method technology-agnostic by encapsulating technology-dependent steps. In fact, the abstraction of *mcDFG* contains concepts that are pervasive across all the trhee-layered architectures. By changing the technology or architectural style of the system, it will suffice to modify the *mcDFG* generation procedure (see Section 5.2). In particular, the first step required to generate the abstraction is particularly dependent on the system's architectural style, as it needs to know where the business logic is implemented. Instead, the second step depends primarily on the DI and automatic lifecycle management framework used by the system.

4) Reliability: whether and to what extent the observations can be reproduced by other researchers. To ensure independent reproducibility and verifiability of the results, we made available online: the Flight Manager source code, its 32 faulty versions, and all the test suites derived from both the *mcDFG* and the PND abstractions (please refer to the replication package).

7. Conclusions

In the development of software architecture, Dependency Injection and automated lifecycle management play an essential role for productive implementation of the Inversion of Control principle. This supports abstraction and loose coupling, enabling developers to specify components lifecycle models in a choreographic perspective and to delegate to a Container the consequent orchestration. Yet, this also introduces error-prone steps and largely reduces designers control over the actual resulting behavior.

In this work, we characterize the chain of threats affecting the development of software architectures that rely on Dependency Injection and automated lifecycle management, identifying faults that can be introduced in the specification of managed components lifecycles and in their composition, and characterizing mechanisms of fault to failure propagation that result from the interaction of structural characteristics of software components and navigation paths exposed by the presentation layer.

We then propose an abstraction, named managed component Data Flow Graph (*mcDFG*), which unravels concurrency among objects living in the execution of a Use Case and which is derived through an automated procedure.

The *mcDFG* abstraction is here finalized to the implementation of a Model-Based Testing approach, supporting both test case selection and oracle verdict on state errors that would be hard to observe as functional deviations at the application interface. Experimental proof of concept on a mid-sized application with a suite of 32 faulty mutations suggests the viability and capability of detecting faults of the proposed approach.

In terms of implications of the study, from a research perspective, the work presented argues on the limitations of testing three-layered architectures *via* black-box strategies, and lays the groundwork for more sound and comprehensive

Language	Framework	Built-in Scopes				
		request	*enclosed*	*session*	*application*	*conforming*
C#	*Autofac*	✓	✓	✓	✓	✓
	Spring.NET DI	✓		✓	✓	✓
Java	*CDI*	✓	✓	✓	✓	✓
	Spring DI	✓		✓	✓	✓
	Guice	✓		✓	✓	✓
Python	*Dependency Injector*					✓
	Pinject				✓	✓
	Injector	✓	✓	✓	✓	✓

Table 3
Comparison among built-in scopes for main IoC frameworks in high-level programming languages C#, Java, and Python.

testing approaches. As documented in this research, novel viable approaches can be conceptualized and used to integrate information from *both* the presentation layer and the business logic layer by adapting existing black-box model-based testing approaches. From a practitioner perspective, the research serves as a cautionary tale on the impossibility of comprehensively testing a software-intensive system based solely on the state of the presentation layer. During all testing stages, developers must be aware that considering only the presentation layer (e.g., by using solely monkey testing) does not allow to unveil faulty interaction sequences hidden in the business logic of the system under test. In addition, with this research we make available a thorough fault model and a set of failure modes of three-tier architectures with which they can improve their daily testing practice and build upon it. Finally, we make readily available for practitioners a proof of concept implementation outlining how to concretely build a test suite addressing the presented fault model in the companion replication package of this study.

The obtained results are promising, but we consider this investigation as a preliminary step toward the consolidation of the model-based testing through the *mcDFG* abstraction. As future research activities, we plan to mitigate potential threats to validity associated with our findings by conducting empirical experimentation encompassing real-world systems with different architectural styles and technologies. As additional future work, we plan to fully automate the approach (with exception of the optional Step 5, as its nature requires human intervention).

In a wider perspective, this work also aims at providing a contribution connecting patterns in the practice of software architecture with models of concurrency open to analysis and automated verification [10]. The application and its faulty mutations, and their associated models, are part of this aim. In particular, this opens the way to enrich *mcDFG* models with a measure of probability, induced by discrete time characterization of interaction sequences in the execution of use cases.

Acknowledgement

We would like to express our gratitude to Jacopo Parri, Samuele Sampietro, Boris Brizzi, and Nicolò Pollini for their invaluable advice, technical insights, and contributions to this project.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE0000001 - "RESTART").

References

[1] Allen, J.F., 1983. Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843.
[2] Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M., 2014. Mobiguitar: Automated model-based testing of mobile apps. IEEE software 32, 53–59.
[3] Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th international conference on Software engineering, pp. 402–411.
[4] Arcuri, A., 2019. Restful api automated test case generation with evomaster. ACM Transactions on Software Engineering and Methodology (TOSEM) 28, 1–37.
[5] Barth, A., 2011. Rfc 6265-http state management mechanism. Internet Engineering Task Force (IETF) , 2070–1721.
[6] Bass, L., Clements, P., Kazman, R., 2003. Software architecture in practice. Addison-Wesley Professional.
[7] Biagiola, M., Stocco, A., Ricca, F., Tonella, P., 2019. Diversity-based web test generation, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 142–153.
[8] Brown, K., Craig, G., Amsden, J., Hester, G., Berg, D., Pitt, D., Jakab, P.M., Stinehour, R., Weitzel, M., 2003. Enterprise Java Programming with IBM WebSphere. Addison-Wesley Professional.
[9] Buschmann, F., Henney, K., Schmidt, D.C., 2007. Pattern-oriented software architecture, on patterns and pattern languages. volume 5. John wiley & sons.
[10] Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G., 2016. Formal verification with confidence intervals to establish quality of service properties of software systems. IEEE Transactions on Reliability 65, 107–125. doi:10.1109/TR.2015.2452931.
[11] Carrozza, G., Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S., 2013. Analysis and prediction of mandelbugs in an industrial software system, in: 2013 IEEE Sixth international conference on software testing, verification and validation, IEEE. pp. 262–271.
[12] Cockburn, A., 2000. Writing effective use cases, Addison-Wesley Professional. pp. 46–57.

[13] Cotroneo, D., Pietrantuono, R., Russo, S., Trivedi, K., 2016. How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation. Journal of Systems and Software 113, 27–43.

[14] Fioravanti, S., Mattolini, S., Patara, F., Vicario, E., 2016. 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, pp. 297–308.

[15] Fowler, M., 2006. Inversion of control containers and dependency injection pattern. http://www. martinfowler. com/articles/injection. html .

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

[17] Frajták, K., Bureš, M., Jelínek, I., 2015. Transformation of ifml schemas to automated tests, in: Proceedings of the 2015 Conference on research in adaptive and convergent systems, pp. 509–511.

[18] Garousi, V., Mäntylä, M.V., 2016. A systematic literature review of literature reviews in software testing. Information and Software Technology 80, 195–216. URL: https://www.sciencedirect.com/science/article/pii/S0950584916301446, doi:https://doi.org/10.1016/j.infsof.2016.09.002.

[19] Grottke, M., Matias, R., Trivedi, K.S., 2008. The fundamentals of software aging, in: 2008 IEEE International conference on software reliability engineering workshops (ISSRE Wksp), Ieee. pp. 1–6.

[20] Grottke, M., Trivedi, K.S., 2007. Fighting bugs: Remove, retry, replicate, and rejuvenate. Computer 40.

[21] Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., Su, Z., 2019. Practical gui testing of android applications via model abstraction and refinement, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 269–280.

[22] Itkonen, J., Mantyla, M.V., Lassenius, C., 2007. Defect detection efficiency: Test case based vs. exploratory testing, in: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), IEEE. pp. 61–70.

[23] Kaner, C., Falk, J., Nguyen, H.Q., 1999. Testing computer software. John Wiley & Sons.

[24] Kung, D.C., Liu, C.H., Hsia, P., 2000. An object-oriented web test model for testing web applications, in: Proceedings First Asia-Pacific Conference on Quality Software, IEEE. pp. 111–120.

[25] Leveau, J., Blanc, X., Réveillère, L., Falleri, J.R., Rouvoy, R., 2022. Fostering the diversity of exploratory testing in web applications. Software Testing, Verification and Reliability 32, e1827.

[26] Li, J., Zhao, B., Zhang, C., 2018. Fuzzing: a survey. Cybersecurity 1, 1–13.

[27] Lin, J.W., Salehnamadi, N., Malek, S., 2023. Route: Roads not taken in ui testing. ACM Transactions on Software Engineering and Methodology 32, 1–25.

[28] Linares-Vásquez, M., Moran, K., Poshyvanyk, D., 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 399–410.

[29] Liu, Z., Chen, C., Wang, J., Huang, Y., Hu, J., Wang, Q., 2022. Guided bug crush: Assist manual gui testing of android apps via hint moves, in: Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, pp. 1–14.

[30] Mahmood, R., Mirzaei, N., Malek, S., 2014. Evodroid: Segmented evolutionary testing of android apps, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 599–609.

[31] Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M., 2019. The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering 47, 2312–2331.

[32] Martin, R.C., 2000. Design principles and design patterns. Object Mentor 1.

[33] Martin, R.C., 2017. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin Series, Prentice Hall, Boston, MA.

[34] Mesbah, A., Van Deursen, A., 2009. Invariant-based automatic testing of ajax user interfaces, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE. pp. 210–220.

[35] Mishra, C., Koudas, N., Zuzarte, C., 2008. Generating targeted queries for database testing, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 499–510.

[36] Nidhra, S., Dondeti, J., 2012. Black box and white box testing techniques-a literature review. International Journal of Embedded Systems and Applications (IJESA) 2, 29–50.

[37] Nie, L., Said, K.S., Ma, L., Zheng, Y., Zhao, Y., 2023. A systematic mapping study for graphical user interface testing on mobile apps. IET Software .

[38] Patara, F., Vicario, E., 2014. 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. pp. 1–5.

[39] Rapps, S., Weyuker, E.J., 1985. Selecting software test data using data flow information. IEEE transactions on software engineering , 367–375.

[40] Ricca, F., Tonella, P., 2001. Analysis and testing of web applications, in: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, IEEE. pp. 25–34.

[41] Richardson, C., 2006. POJOs in Action, Developing Enterprise Applications with Lightweight Frameworks. Manning.

[42] van Rooij, O., Charalambous, M.A., Kaizer, D., Papaevripides, M., Athanasopoulos, E., 2021. webfuzz: Grey-box fuzzing for web applications, in: Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26, Springer. pp. 152–172.

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

[44] Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., 2013. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. volume 2. John Wiley & Sons.

[45] Shafique, M., Labiche, Y., 2015. A systematic review of state-based test tools. International Journal on Software Tools for Technology Transfer 17, 59–76.

[46] Souter, A.L., Pollock, L.L., Hisley, D., 1999. Inter-class def-use analysis with partial class representations, in: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 47–56.

[47] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z., 2017. Guided, stochastic model-based gui testing of android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 245–256.

[48] Utting, M., Pretschner, A., Legeard, B., 2012. A taxonomy of model-based testing approaches. Software testing, verification and reliability 22, 297–312.

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

[50] Yousaf, N., Azam, F., Butt, W.H., Anwar, M.W., Rashid, M., 2019. Automated model-based test case generation for web user interfaces (wui) from interaction flow modeling language (ifml) models. IEEE Access 7, 67331–67354.

[51] Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., Liu, Y., 2021. Automatic web testing using curiosity-driven reinforcement learning, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 423–435.