FuzzE, Development of a Fuzzing Approach for Odoo's Tours Integration Testing Plateform

Gabriel Benoit*, François Georis†, Géry Debongnie†, Benoît Vanderose* and Xavier Devroey*
*NADI, University of Namur, Namur, Belgium. Emails: benoit.vanderose@unamur.be, xavier.devroey@unamur.be
†Odoo S.A., Grand-Rosière, Belgium. Emails: fge@odoo.com, ged@odoo.com

***Abstract*—For many years, Odoo, an open-source add-on-based platform offering an extensive range of functionalities, including Enterprise Resource Planning, has constantly expanded its scope, resulting in an increased complexity of its software. To cope with this evolution, Odoo has developed an integration testing system called *tour* execution, which executes predefined testing scenarios (i.e., tours) on the web user interface to test the integration between the front, back, and data layers. This paper reports our effort and experience in extending the tour system with fuzzing. Inspired by action research, we followed an iterative approach to devise *FuzzE*, a plugin for Odoo's tour system to create new tours. FuzzE was eventually developed in three iterations. Our results show that mutational fuzzing is the most effective approach when integrating with an existing testing infrastructure. We also reported one issue to the Odoo issue tracker. Finally, we present lessons learned from our endeavor, including the necessity to consider testability aspects earlier when developing web-based systems to help the fuzzing effort, and the difficulty faced when performing triage and root cause analysis on failing tours.**

***Index Terms*—software integration testing, fuzzing, Odoo, action research**

I. Introduction

Odoo [37] is a highly configurable platform with a large offer of modules to rapidly build and customize web-based applications supporting the day-to-day of a company [23], [25]. For instance, Odoo offers modules for classical Enterprise Resource Planning (ERP) tasks like managing customers, sales, invoices, etc. With its rapid development, Odoo is facing new challenges, including the automated execution of integration tests designed to test the correct integration of the front-end, developed in JavaScript and TypeScript, and the back-end, developed in Python. For that, Odoo has developed an in-house solution called *tours execution* [35]. A *tour* is an integration test, starting an Odoo instance and executing a pre-defined test scenario on the web interface. The development and growing usage of the tours for integration testing offered us an opportunity to develop new exploratory testing approaches. In our case, we seek to incorporate fuzzing into the tour system.

There are several automated test case generation approaches for various kinds of systems. Search-based can generate unit [15], [21], [24], [43], integration [19], [20], [40], [45], [48], and system tests [7], [31], usually by relying on instrumentation of the source code under test that can be automated or not. Similarly, fuzzing has been used to generate test input values for various purposes [30], including enhancing code coverage [39], [41], [50], triggering specific crashes [10], or adopt an exploratory testing approach for system testing [8], [22]. Search-based and fuzzing approaches have been applied to various systems, including web-based systems like the Odoo platform [3], [49]. Our goal in this paper is not to develop a new approach but to enhance exploratory testing of the Odoo platform by generating *tours* [35].

However, applying fuzzing to test a web-based system yields several challenges [4], [5]. First, the existence of dynamic interface elements and dynamic code means considering interface changes after every action and even in the absence of actions. Second, it must be possible to mimic a user's behavior, particularly when filling in forms with fields requiring specific formats. Failing to do so prevents the exploration of other parts of the application. Finally, in some cases, it is necessary to be able to fill in fields with predetermined values, for example, in the case of authentification [18]. This is particularly true for Odoo, given the data-management nature of many different features.

We developed our plugin for the tours execution system called FuzzE at Odoo S.A. in Grand-Rosière, Belgium, by building on previous work testing graphical web user interfaces [3], [32], [33], [51]. Inspired by action research, we iteratively defined, implemented, and evaluated our solution. Due to the time constraints of the internship at Odoo S.A., we limited ourselves to three iterations [12]. We evaluated different configurations of FuzzE on the Odoo platform and reported a bug in the GitHub issue tracker. The FuzzE plugin is open-source and available [11], and we provide a replication package for our evaluation [13].

The rest of this paper is structured as follows: first, we provide an overview of Odoo S.A.'s context at the end of this section. Then Section II describes the necessary background information, while Section III gives an outline of the Odoo platform. Section IV describes the development of the FuzzE approach and its implementation. The evaluation of FuzzE 3.0 is described in Section V. Finally, we provide and discuss lessons learned from our endeavor in Section VI and conclude in Section VII.

Odoo's Context: Odoo (previously OpenERP) [25], [37] is an open-source addon-based platform offering a large range of

This research was funded by the CyberExcellence by DigitalWallonia project (No. 2110186), funded by the Public Service of Wallonia (SPW Recherche).

```
<email>     ::= <localpart> "@" <domain>
<localpart> ::= <string>
<domain>    ::= <string> "." <tld>
<string>    ::= <character> <string> | <character>
<tld>       ::= "com" | "net" | "org" | "io"
...
```

Listing 1. Excerpt of the valid e-mail BNF grammar of FuzzE

functionalities, including Enterprise Resource Planning (ERP), like customer relationship management, sales management, stock management, etc., but also a content management system (CMS), business intelligence, etc. Odoo's core developer and maintainer is *Odoo S.A.*, a Belgian company with more than 2,800 employees, responsible for developing and maintaining the core Odoo platform and its add-ons and the Odoo development framework. Odoo's development framework allows many external contributors to develop addons to extend and customize Odoo to fit specific needs. To give an order of magnitude, Odoo S.A. counts more than 5,000 external partners and 12 million Odoo users. We use Odoo version 17.0, which counts 1,190.4 kLOC (i.e., thousands of lines of code) of Python code, 1,662.0 kLOC of JavaScript/TypeScript code, and 595.0 kLOC of XML code.

II. Background and Related Work

Among the different possible testing strategies, we focus on fuzzing and related web-based testing through the user interface (i.e., a web page) for black-box testing, as we do not assume access to the source code for additional instrumentation.

A. Generation, Mutation, and Grammar-based Fuzzing

Fuzzing consists in generating input values (called *fuzz*) pseudo-randomly to trigger unexexpected behaviors in a system under test [29], [30], [51]. Existing fuzzing approaches are usually tailored for a particular system or target specific types of faults, e.g., fuzzing for security testing [10], [14]. We can distinct two categories of fuzzers [29]: *generation-based fuzzing*, producing fresh fuzz, and *mutation-based fuzzing* producing fuzz by modifying either provided input values or previous fuzz. For both categories, the fuzzer can generate invalid fuzz that can be of little interest when testing systems having highly structured input values.

Fuzzers rely on one or more grammars to compactly represent valid inputs to prevent invalid values from being generated. *Grammar-based fuzzing* can be applied to both generation and mutation-based fuzzers [17], [27], [28], [46], [51]. For instance, Listing 1 presents an excerpt of the grammar used in our implementation to represent valid e-mail addresses. In our case, we only consider `.com`, `.net`, `.org`, and `.io` domain extensions (as specified by the `<tld>` in Listing 1).

Grammars offer a powerful tool for generating and mutating values. A generation-based fuzzer can derive valid fuzz following the grammar's production rules. In contrast, a mutation-based fuzzer can modify an existing value by parsing it and modifying the abstract syntax tree according to the production rules. In both cases, the grammar can also guide the fuzzing effort, for instance, by checking that the different production rules are covered. In our case, we will use grammar-based fuzzing for two distinct purposes: (i) generating valid input values for the different input fields of Odoo and (ii) representing the different states of the Odoo web-UI, as explained in the following sub-section.

B. Automated Testing Through Web User Interface

Previous attempts to test Odoo through its web user interface include TESTAR [49]. TESTAR is a tool for automating exploration testing at the user interface level of (initially) Desktop applications. It relies on the operating system's Accessibility API to scan the graphical user interface (i.e., the system's current *state*). From a scan, TESTAR identifies widgets (e.g., buttons, input fields, etc.) to identify possible *actions* to execute. Each action execution potentially leads the system to a new state that is checked for validity.

Almenar et al. [3] applied TESTAR to Odoo with a few adjustments to consider web application specificities (web latency, exclusion of the browser from the elements to test, etc.). Testing Odoo in a black-box setting (i.e., without access to the source code), they used the number of states, the longest path, and the minimum and maximum coverage per state (i.e., the ratio between the number of actions executed and the total number of actions on a screen) as metrics to assess the coverage achieved by TESTER. Their approach faced limitations, including the inability of the accessibility API to detect confirmation questions in the form of emerging windows and interactions coded via the CSS, resulting in a mix between actions available in emerging panels and those in the windows under them. In our case, we rely on Odoo's tour execution system, which leverages jQuery to select elements in the HTML page.

More generally, TESTAR and other related approaches [9], [32] like Crawljax [33] and QExplore [47] build a state machine-like model representing the different states of the graphical user interface (e.g., one screen corresponds to a state) with the various actions that can be performed (e.g., clicking a specific button or fill in a field). Following Zeller et al.'s examples [51], we will consider automating the exploration of the Odoo interface to build a state machine describing different possible valid tours. Given that tours are defined using a Domain Specific Embedded Language (DSEL), we will encode this state machine as a grammar that can be used by our fuzzer. We provide more information about the Odoo platform and the tour execution system in the following section.

III. The Odoo Platform

Odoo follows a three-tiers architecture [36]: the presentation tier (*front-end*), here, a web client, relies on the Odoo Web Library (OWL) framework [38] written in Typescript; the logic tier (*back-end*) is implemented as Python modules; and the data tier (*persistence layer*) relies on a PostgreSQL database and is abstracted using an Object Relational Mapping (ORM).

```
 1 /* Here are imports */
 2 registry.category("web_tour.tours").add('main_flow_tour'
       , {
 3   test: true,
 4   url: "/web",
 5   steps: () => [
 6       /* Here is a step */, {
 7       mobile: false,
 8       trigger: ".o_field_widget[name=partner_id] input",

 9       extra_trigger: ".o_breadcrumb .active:contains('Mj
            \?\}\-\#')",
10       content: _t('Select a seller'),
11       position: 'top',
12       run: "text 1b\{\#\|\~",
13     }, {
14     /* Here is another step */
15   }]
16 });
```

Listing 2. Excerpt of `main_flow.js` tour.

A. High Configurability

Odoo's high configurability relies on a modular approach [23], [36]. The default installation includes several modules (also called *add-ons* in Odoo's ecosystem), which can be extended. The Odoo platform includes an add-on management system, which is in charge of installing and de-installing the module and its dependencies. A module's design follows the same three-tier architectural vision (a front-end, a back-end, and a persistence layer), and it can modify existing modules using different mechanisms. In other words, each module can have side effects (i.e., desired feature interaction) on the already installed modules. Given the highly configurable nature of the Odoo platform, advanced testing techniques are required to ensure the quality of the platform and its various add-ons.

B. Integration Testing

Odoo relies on *tours* executed by a *test runner* to test the integration between the three tiers of the different modules of the application [35]. A tour is an ordered sequence of *steps*, where each step is an action simulating an interaction between a user and the web interface. Each tour (i.e., an integration test) is described using a Domain-Specific Embedded Language (DSEL) implemented as a set of JavaScript functions. Each step composing a tour has the following properties:

- a **trigger**, defined as a jQuery 3, to select an element to run an action on. Given the dynamic nature of the Odoo web interface, the runner will wait for the element to be visible before executing the action;
- an optional **extra-trigger**, which can serve as an additional precondition (without executing the action on the selected element);
- a **run** property, defining the action to execute on the element selected by the trigger, which can be a function or one of the base functions (click, double click, fill with a given text, etc.);
- a **timeout**, specifying how long the step can wait before executing the run action; etc.

Listing 2 presents an excerpt of the `main_flow.js` tour, exercising many of Odoo's functionalities. The line 8 specifies the trigger that selects an input field with a name equal to `partner_id`, with a precondition specified at line 9 testing the presence of a random string (here, `Mj\?\}\-\#`) in the active component. The action specified at line 12 will fill the random string `1b\{\#\|\~` in the `partner_id` input field. More information is available in the Odoo documentation [35].

For a given tour, the test runner will spawn an Odoo instance with a given set of modules installed (i.e., the core modules by default) and create a PhantomJs browser, point it to the proper URL, and simulate the click and inputs, according to the sequence of steps. In our context, one limitation of the *tour testing* system is the impossibility of dynamically generating tours: all the steps must be defined before starting the Odoo server.

IV. Development of the FuzzE Approach

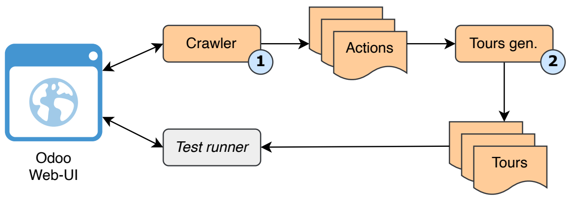
We aim to further enhance testing practices at Odoo by **integrating fuzzing into the existing integration testing infrastructure** (*Req.1*). This is a hard requirement that must be respected for (practical) reasons: first, the existing tours system is itself included in different continuous integration workflows and cannot be modified. Second, the setup and configuration of an Odoo instance are not trivial, and relying on the existing tours system eases the process. The second hard requirement is that **FuzzE has to be written in Python** (*Req.2*) to ease the integration into the Odoo ecosystem.

In other words, with FuzzE, we seek to generate tours corresponding to potential user interactions that trigger unexpected failures. Inspired by *action research*, we will follow an iterative process to define and implement our approach and evaluate it in the Odoo context. Each iteration will include a preliminary evaluation and a retrospective on observations made so that these can be taken into account in the next iteration. In total, the development of our approach counts three iterations detailed hereafter.

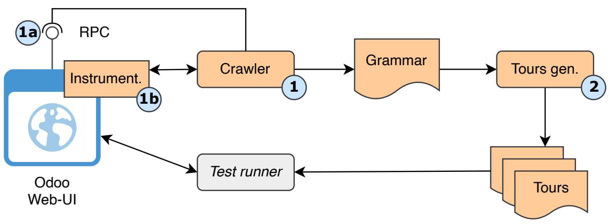
A. Iteration 1: Random Exploration of the Odoo Web UI

As mentioned in Section III, the main limitation of the tour testing system is that one needs to restart Odoo to register new tours to execute. This means that any exploration of the highly dynamic web UI has to be done separately to identify possible user actions from which we can generate tours using *generational fuzzing*. This is depicted in Figure 1a: first, a *crawler* (1), i.e., a software capable of exploring the pages of a website by extracting links, clicking on the page buttons and filling in the fields, explores the UI; then, the sequences of actions serve as input to the *tours generator* (2) that will produce tours to be executed on the Odoo *test runner* Our initial aim is, therefore, to implement a crawler for dynamically exploring the web UI, based on the Crawljax technique used by Mesbah et al. [33] and TESTAR technique used by Almenar et al. [3], both written in Java, to derive tours from the collected sequences of actions.

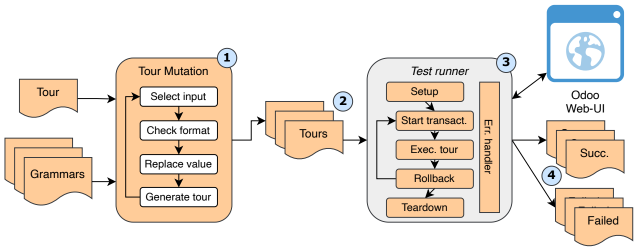
a) Implementation: To respect *Req.2*, we implement our crawler in *Python* and *Selenium* based on the *fuzzing book*'s [51] implementation. For this first iteration, we use the most



(a) Iteration 1: FuzzE 1.0 - Random generational fuzzing.

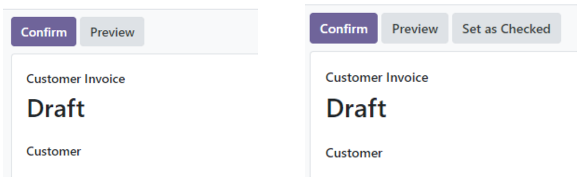


(b) Iteration 2: FuzzE 2.0 - Grammar-based generational fuzzing.



(c) Iteration 3: FuzzE 3.0 - Mutational fuzzing.

Fig. 1. FuzzE conceptual evolution over three iterations.



(a) Checkbox is checked. (b) Checkbox is unchecked.

Fig. 2. Example of the high reactivity of the Odoo web interface: update of the Form header when the state of a related checkbox changes.

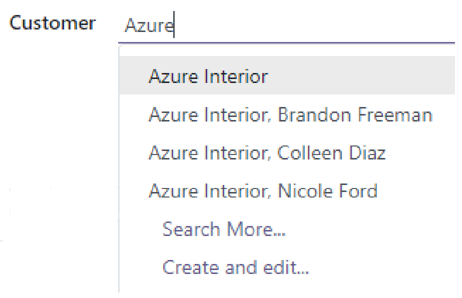

Customer
Azure
Azure Interior
Azure Interior, Brandon Freeman
Azure Interior, Colleen Diaz
Azure Interior, Nicole Ford
Search More...
Create and edit...

Fig. 3. Example of `Many2One` field available in Odoo forms.

naive method: we randomly explore the web UI and extract the different components encountered during the exploration (here, links, buttons, and input fields). The goal of this first iteration is to familiarize ourselves with Odoo and identify potential issues.

The crawling process proceeds as follows: (i) a Chrome client instance is launched via Selenium and goes to the Odoo instance page. (ii) The crawler logs in as an administrator and iterates until no new elements are found, in which case, it generates a tour from the sequences of actions performed. (iii) For each iteration, it extracts links, buttons, and input fields from the current page, then, randomly selects the next component. (iv) If the next component is a link, it verifies that this is an internal link and clicks on it. (v) If the component is an empty input field, it generates a random string to fill it. (vi) And if the component is a button, it clicks on it. Due to the high reactivity of the web UI implemented using the OWL framework, we need to repeat the extraction of the different components after each interaction with the web UI, as elements might dynamically appear or disappear. Figure 2 presents an example of a highly reactive web interface with the availability of the button `Save as Checked` in the form header, depending on the state of a related checkbox.

b) Observations: Following the implementation, we conducted a small experiment on the crawler for 1,000 executions on the standard Odoo *Community* version (v17.0) to identify potential issues. First, the crawler does not wait for a possible UI update before repeating the extraction process, which crashes the process when the update occurs between extraction and execution of the chosen action. Second, as selection is totally random, it is practically impossible to go beyond the form-filling stages, as required fields are not filled in. Finally, random input strings do not pass validation on the client side and require additional information about format or valid input values.

B. Iteration 2: Grammar-based Tours Generation

In this iteration, we solve the issues from the first iteration. As illustrated in Figure 1b, we add a set of methods specific to the Odoo context (1a and 1b) and a mechanism in the crawler (1) for recording actions performed in the form of a state machine encoded as a grammar, as proposed by Zeller et al. [51]. This grammar is then reused to generate tours (2).

a) Odoo-specific Interactions: In practice, an additional session connecting to the Odoo API via a Remote Procedure Call (RPC) system (1a in Figure 1b) is set up during the exploration to collect meta-information describing the various fields. Odoo-specific action sequences are then dynamically defined, e.g., filling non-standard input fields based on the meta-information, like dates, account numbers, etc.

For this iteration, we implemented a macro for the `Many-2One` field type (see Figure 3 for an example), a dropdown field allowing to choose a value from a list of proposals, with autocompletion. This macro clicks on the field to display the list of values, extracts the values, and randomly selects one value. Values include `Search More...` and `Create and edit...` options, which allow exploring on-the-fly creation and advanced search forms.

This approach, however, has (significant) limitations: (i) there is no filtering mechanism to collect only information about the fields of the current *view* (i.e., the current page displayed to the user); (ii) the information about the *model* (i.e., the data linked to the different fields) is extracted via a URL *query argument*. However, Odoo's specifications stipulate that *query arguments* will be removed from URLs in future versions; (iii) the OWL framework, used to define the web interface, allows defining and extending components (e.g., forms), which means that the naming of the input fields is not unique and requires additional instrumentation on the web-client side to collect the XPath from the web-page's root, slowing down the exploration process.

Also, using that information for setting up a macro for each field type, if it were to be exhaustive, would prove very cumbersome to maintain, as Odoo implements over 50 different field types, including, fields with advanced interactions such as drag and drop, making simulation of such movements very complex.

b) Odoo's Web UI High Reactivity: We partially solved the web UI update detection issue identified in the previous iteration by adding instrumentation on the client side to detect updates on the web page (1b in Figure 1b). More specifically, we rely on the native JavaScript function `MutationObserver` to detect changes in the web page's Document Object Model (DOM).

Remaining limitations include the impossibility to predict whether or not an action will result in a change to the DOM, requiring to add a timeout to the wait mechanism. Also, execution of the macro for `Many2One` fields generates changes notification that should be ignored. Finally, the major challenge is related to error detection: when an error occurs, there is sometimes an irregular delay before the modal error window appears, which is problematic because, during this delay, actions continue to be executed, making it harder to detect the action that led to the error.

c) Observations: Again, following the implementation, we conducted a small experiment for 1,000 executions on the standard Odoo *Community* version (v17.0) to identify potential issues. First, despite meta-data provided by the Odoo API, forms remain challenging to fill automatically as one needs to synthesize highly structured data with dependencies between the different fields.

Second, regarding identifying states, we first considered using a method similar to that proposed by Mesbah et al. [33]: i.e., comparing the DOM tree at each iteration with previously saved trees. This method is very costly regarding memory but also impacts performance using Python and *Selenium*. Also, using a web framework such as OWL, which allows for component recycling, leads to false positives and negatives when identifying the states. As Mesbah et al. [33] explain, the DOM trees are compared using a tolerance threshold. In our case, if we do not set it to 0, we might end up with so many similar components (and therefore DOM parts) that a false positive is generated. Conversely, if we set the threshold to 0, all DOMs are considered different, and we end up with linear flows in which all states are considered distinct. Such situations render the derived grammar unusable, as only one path is possible. An alternative way of approximating the similarity of the two states would be to compare all the data stored in the Odoo database. However, given the size and complexity of the database, even for the standard version, it seems unrealistic.

In conclusion, tour generation using random or grammar-based generational fuzzing cannot be implemented and integrated into the current infrastructure without heavy additional development and refactoring. This is due to various factors specific to Odoo's implementation (OWL, RPC API access, etc.) but also linked to contextual constraints (database access, highly structured data generation, etc.), making it complex to solve problems encountered by Vos et al. [3] and Mesbah et al. [33].

C. Iteration 3: Tours Mutation

Similarly to Almeida et al., [2], [42], where an existing test case is mutated to create new test cases, this third iteration focuses on mutating an existing tour. So, instead of generational fuzzing, requiring to separate the exploration of the web UI from the generation of the tours, we focus on *mutational fuzzing*, starting from a given tour to generate new tours by applying mutations. This reduces the scope of testing as it depends on the initial tour (i.e., the initial seed), but meets our objectives as it leads to a working prototype, integrated into Odoo's testing infrastructure. We can mutate a tour in different ways: we can mutate the actions of the tour by mutating steps, or we can focus on the input data used during the tour's execution. Based on previous observations, the first option will pose several challenges as it requires considering the interdependencies between the different actions during mutation. Therefore, we chose the second option and focused on mutating input data within a tour.

As illustrated in Figure 1c, FuzzE 3.0's Python implementation relies on ANTLR [44] and works as a two steps process: first, it takes as input a *tour* (and input values grammars) and mutates its inputs (1). It produces variants (2) that will be executed in a second step by the Odoo *test runner* (3). A report (4) details which tours succeeded and failed during the execution. We detail the different steps of the mutations in the following paragraphs.

a) Tours Mutation: For a given tour, the different inputs are identified, thanks to the command `text` used in a *run* property to indicate to the runner to fill the corresponding *trigger* with the given text. Mutation consists of replacing the input value with another generated value. To avoid issues related to structured input fields encountered during iteration 1, FuzzE 3.0 supports formats encoded as grammars and provided as inputs (here, integer, natural, ream, e-mail, phone, date, and string). If the original input can be parsed according to one of the formats, the corresponding grammar will be used to generate a new input. Also, to account for dependencies between different steps, FuzzE replaces all references to the original text with the corresponding new value in the tour.

b) Input Selection: FuzzE can be configured to select the inputs to replace based on one of the following policies: the *all strategy* selects all the inputs; the *selection strategy* selects a specific subset of all fields; the *random strategy* selects a random subset of all fields following a normal distribution (centered reduced $\mathcal{N}(0,1)$ by default). By default, an input has a probability of 0.5 to be selected; the *random selection strategy* selects a random subset of a given set of all inputs. Each time an input is selected and replaced, FuzzE records the newly mutated tour.

c) Integration in the Odoo Test Runner: We rely on the `subTest` functionality provided by the `unittest` module. A `subTest` is a method for including several small tests within a larger test. The goal is to avoid re-executing the time-consuming `setup` and `teardown` parts for each generated tour. As illustrated in (3) in Figure 1c, each `subTest` corresponds to the execution of a single mutated tour. If a mutated tour fails, the error is recorded with the step that caused it and the mutated input. The database consistency restoration between two tours is handled using transactions: the database is set when starting the overall execution, and a new transaction starts for each new `subTest` (i.e., each new tour) with a corresponding rollback, executed at the end of the `subTest`.

d) Observations: The small experiment for 1,000 executions on the standard Odoo *Community* version (v17.0) on this third iteration provided encouraging preliminary results. We used FuzzE 3.0 to carry out a larger empirical evaluation, described in the following section.

V. Evaluation

Our primary goal was to enhance exploration testing for the Odoo platform. To that end, we developed FuzzE 3.0, a plugin for the tour execution system, and evaluated it on Odoo version 17.0. We focused on failing tours as they might denote the presence of faults.

A. Evaluation Setup

We evaluated FuzzE 3.0 using the `main_flow` tour as input. This tour is the largest available tour in the Odoo test suite and exercises most of the default Odoo features. We used the four different selection strategies described in Section IV-C: *all*, selection (*sel*), random (*ran*), and random selection (*rse*).

a) Parameters Configuration: For the selection (*sel*) and random selection (*rse*) strategies, we excluded 10 inputs from the 35 that can be selected. Excluded inputs are either fixed values (i.e., enumeration) or quantities directly validated by the front end. A free input field that serves as a control indicator for the tour is also excluded. We run the evaluation with a budget of 1,000 iterations. Given that the `main_flow` tour takes ~ 2 minutes to execute, the execution time upper-bound for the evaluation is $1,000 \times 2 \times 4 \simeq 5.5$ days. FuzzE 3.0 is open source [11], and the details of our evaluation and the results are available in our replication package [13].

b) Data Analysis: We compare the different strategies based on the percentage of failed tours (4 in Figure 1c). We complete our analysis by manually triaging failed tours and investigating the potential causes (i.e., underlying faults) of those failures. For that, we identify the failing steps in the generated tours with their corresponding input data.

TABLE I
Failure rates for the different strategies.

Selection strategy	Failures	Rate
All (*all*)	1,000	100%
Selection (*sel*)	45	4.5%
Random (*ran*)	1,000	100%
Random selection (*rse*)	28	2.8%

B. Evaluation Results

As shown in Table I, the *all* and *ran* strategies result in a failure rate of 100%. The *sel* and *rse* strategies have a failure rate of 4.5% and 2.8%, respectively. These two percentages seem more likely to represent cases where potential bugs are found. Those numbers can be explained by the fact that some fields in the `main_flow` tour only accept a specific set of values, causing failures for the *all* and *ran* strategies. Such fields have been excluded for the *sel* and *rse* strategies, as explained in Section V-A. To confirm our results further, we proceeded with a manual analysis of the failing tours for the different strategies.

a) All Strategy: The failures occurred at the same step for the 1,000 iterations (`.o-form-buttonbox`) at the tour's beginning. Our manual investigation shows that one of the prior steps requires selecting a type of product concerned by the action specified in the `.o-form-buttonbox` step. There is a dependency between those two steps, requiring the action and products to be compatible, which is not the case in the 1,000 iterations.

b) Selection Strategy: The step `.o_field_widget-[name=email] input` accounts for 27 failures out of 45 failures. After re-executing the failing tours, we noticed that when encoding a new customer or salesperson, creating one on the fly is possible using only an e-mail address. However, it turns out that if there are commas (`','`) in the text entered, only the first valid e-mail address detected between the commas is saved. For example, an entry of the form `test-1,test@test.com,test-2,test2@test.com` saves `test@test.com` in the database. We opened an issue in the Odoo Github to report the problem: https://github.com/odoo/odoo/issues/168068.

The steps `.o_data_row:has(.o_data_cell:contains(<text>)) .o_data_cell:first`, where the placeholder `<text>` is the content of a mutated field, accounts for 18 out of 45 failures. After re-executing the failing tours, we saw that, by default, all names of the corresponding fields have been converted to lower cases. When the placeholder `<text>` in the query `contains(<text>)` contain upper-cases letters, it triggers an error. This could be solved by refining the input grammars provided to FuzzE 3.0 to distinguish upper- and lower-only cases.

c) Random Strategy: Similar to the *all* strategy, the failures occurred at the same step for the 1,000 iterations (`.o_field_widget[name=project_id] input`). Again, after manual investigation, we identified interdependencies between this step and prior steps, requiring compatible data.

d) Random Selection Strategy: The 28 failures occur at the `.o_field_widget[name=email] input` step, same as for the selection strategy. Our analysis showed that it has the same root cause.

C. Threats to the Validity

This subsection presents the threats to the validity of our evaluation using FuzzE 3.0.

a) Internal Validity: We performed our parameters configuration using a preliminary evaluation to identify problematic input fields that should be excluded from selection for mutation for the *sel* and *rse* strategies. This was done in collaboration with the Odoo developers to confirm the relevance of the remaining input fields.

b) External Validity: Our evaluation focuses on Odoo 17.0 using one tour as input. This default tour is used for integration testing as it exercises most of the default Odoo features. We cannot guarantee that it is representative of all possible tours, but as it is the primary tour used by Odoo developers, it was the most relevant one in our case.

c) Reliability: The first author, a junior developer, manually analyzed the failing tours. The second and third authors, both senior developers at Odoo S.A, validated the conclusions. The failure identified as a bug was reported in the Odoo GitHub issue tracker, but unfortunately, the issue has not gone through triage as of the time of writing this paper.[1]

VI. Discussion

Although it is possible to implement fuzzing techniques on Odoo's web interface, the task is far from trivial. In addition to the responsiveness of web applications [33], new challenges, like the usage of highly reactive web frameworks like OWL, make the application of fuzzing more complex. In our case, combining existing tests (i.e., tours) with mutational fuzzing allowed us to test different values for the input fields used within a tour and show unexpected behavior.

A. Generational vs Mutational Fuzzing

As observed when evaluating FuzzE 1.0, not considering the Odoo specificities produced an inefficient tool that could only explore a few pages. As stated by Al Salem and Song [1], it is essential to consider the context to implement an effective and efficient fuzzing approach.

Generational fuzzing requires a dynamic exploration of the Odoo interface or a grammar representing valid tours. In FuzzE 1.0 and 2.0, we attempted to adapt the TESTAR approach from Almenar et al. [3] to the Odoo tour system. However, unlike tours, which rely on HTML code, TESTAR relies on the operating system's graphical API, making it impossible to query the structure of the HTML document to identify *triggers* for the tour system.

Although simpler, *mutational fuzzing* allowed us to respect *Req.1* and integrate FuzzE 3.0 with the Odoo infrastructure, and run fuzzing campaigns. Thanks to the different input grammars, the tool can generate formatted strings and, as pointed out by Alsaedi et al. [5], minimizes validation errors at the web client level.

Lessons Learned. For a system with an existing test execution environment, we recommend using *mutational fuzzing* first, as it is easier to adapt and put into practice. Mutational fuzzing can then serve as a starting point to further enhance the fuzzing effort.

B. Selection Strategies for the Inputs to Mutate

a) All Strategy: This strategy, selecting all the inputs of a tour, has the advantage of not requiring supervision. However, it was ineffective in our evaluation: all the mutated tours failed at the same step, requiring filling a specific field from a set of enumerated values. To avoid this situation, one can choose or specifically design an initial tour referencing only free input fields.

b) Selection Strategy: Unlike the previous strategy, the *selection* strategy allows configuring which input fields' values will be mutated. Our evaluation showed that this strategy was effective. However, it requires a deeper knowledge of the system under test to identify the input fields to select. In our case, we selected the fields by removing input fields requiring specific values on a trial-and-error basis. Other usages of the selection strategy also include selecting a small subset of fields to deepen the testing of particular tour steps.

c) Random Strategy: Like the *all* strategy, the *random* strategy does not offer enough control over the input fields selected for mutation. Our evaluation showed that it was ineffective in our case.

d) Random Selection Strategy: This strategy relies on the second and third strategies, inheriting their strengths and weaknesses. Only a subset of the specified fields is selected, which could be interesting when the time budget is limited. Our evaluation results showed that the second strategy found the same bug.

Lessons Learned. Generally, when testing complex web user interfaces, a strategy providing more control over the input fields to mutate will give better results. The *selection* and *random selection* strategies (if the time budget is limited) allow either to avoid complex fields requiring specific values or test specific parts of the tour. However, the *all* strategy can serve as a starting point with a simple input tour, only requiring to fill input fields corresponding to one of the input grammars.

C. Testability of the System

Our evaluation showed that additional constraints on input fields might break the tours during mutation. This was especially true for the *all* and *random* strategies. One possible lead in solving the limitations we encountered could be to enhance *testability* of the Odoo platform itself [6], [16], [26], [34].

As explained in Section IV-B, we used the Odoo API to collect meta-information (e.g., data input format, input data

[1]https://github.com/odoo/odoo/issues/168068

dependencies, etc.) about the different fields. However, to be effective, this API would require filtering mechanisms to collect information about fields specific to the view that is currently displayed, as well as support for extracting full data (i.e., the *model*) about these fields. The different input fields would also require unique identifiers that do not rely on additional instrumentation.

Lessons Learned. When using fuzzing on web user interfaces, it is interesting to consider testability aspects during the implementation of the system to enable the usage of advanced fuzzing approaches. For instance, by defining unique identifiers for every graphical element of the web interface or by providing readily available APIs to collect additional meta-information, like data formats and additional constraints.

D. Failed Tours Root Cause Analysis

Failing tours require a post-test analysis to understand the origin of a failure. During our evaluation, we observed that the major challenge was identifying the steps causing the failure. Indeed, in most cases, the step where the tour crashes is linked to a previously entered value that has caused a chain reaction leading to the failure. Identifying this chain to proceed to a *root cause analysis* is not easy: it requires replaying the tour to observe what happens. An alternative could be to record the different data (i.e., fields' inputs and database records) and compare those to those recorded when executing the original tour. This alternative requires further development and is left for future work.

Lessons Learned. Except when the failure is caused by direct validation of an input value, the mutated step is not directly causing a failed execution. It is rather a step that triggers depending on the correct execution of the mutated step. This makes the analysis process more complex.

E. Limitations and Future Work of FuzzE 3.0

Our method also has its shortcomings. First, as for generational fuzzing, tours in FuzzE 3.0 are relatively long to execute, requiring extended periods to collect results. Second, the Odoo tour system does not allow loading new tours once the test runner has been launched (3 in Figure 1c). The mutants must, therefore, be generated beforehand (1 in Figure 1c), which prevents using feedback from the tours' execution to drive the mutation process. We made this choice as the *setup* and *teardowm* steps (3 in Figure 1c) of tours' execution is time-intensive. An alternative could be to have a feedback loop connecting the results of tours' execution to the tour mutation (from 4 to 1 in Figure 1c) at the expense of multiple executions of the *setup* and *teardowm* steps. Extending FuzzE 3.0 to include this feedback loop is part of our future work.

Regarding the analysis and triage of the tours' execution, additional instrumentation could be used to collect additional information about the system's state. As explained in the previous section, it could ease the root cause analysis process, but it could also help collect additional feedback to drive the fuzzing effort and provide testing adequacy information (like coverage). Similar to what we implemented for grammar-based generational fuzzing in FuzzE 2.0 (see Section IV-B), this additional instrumentation could be adapted to be included in FuzzE 3.0.

VII. Conclusion

We aimed to develop a fuzzing approach incorporated into Odoo's tour integration testing infrastructure. To achieve this goal, we applied an iterative process inspired by action research to develop FuzzE in three iterations. We tried to apply generational fuzzing for the first and second iterations, requiring exploring the Odoo web interface to collect the possible actions. Unfortunately, this approach proved ineffective due to various factors specific to the Odoo implementation but also linked to contextual constraints, requiring heavy additional development and refactoring to solve the various challenges.

For the third iteration, we applied mutational fuzzing on existing tours by mutating the input values used for the various input fields. We devised different input selection strategies and showed that the *selection* and *random selection* strategies are more effective due to the higher control they offer over selecting the input fields to mutate. Our evaluation also identified an issue reported in the Odoo issue tracker: https://github.com/odoo/odoo/issues/168068. Finally, we discussed several lessons learned, both from the devise and implementation of FuzzE 3.0 and its application on Odoo.

Our future work includes further developing Fuzze 3.0 by investigating possible feedback loops from the tours' execution to the tour mutation to improve the fuzzing process's guidance. Additional instrumentation for data collection during tour execution could also ease the root cause analysis process and the identification of false positives when analyzing failing mutated tours. We intend to see how differential data analysis approaches could help in that regard.

Acknowledgment

We warmly thank Odoo S.A. and the Odoo collaborators for their invaluable help.

References

[1] H. Al Salem and J. Song, "A review on grammar-based fuzzing techniques," *International Journal of Computer Science & Security (IJCSS)*, vol. 13, no. 3, pp. 114–123, 2019.

[2] S. Almeida, A. C. R. Paiva, and A. Restivo, "Mutation-Based Web Test Case Generation," in *Quality of Information and Communications Technology*, M. Piattini, P. Rupino da Cunha, I. García Rodríguez de Guzmán, and R. Pérez-Castillo, Eds. Cham: Springer, 2019, pp. 339–346.

[3] F. Almenar, A. I. Esparcia-Alcázar, M. Martínez, and U. Rueda, "Automated testing of web applications with testar," in *Search Based Software Engineering*, F. Sarro and K. Deb, Eds. Cham: Springer, 2016, pp. 218–223.

[4] A. Alsaedi, A. Alhuzali, and O. Bamasag, "Black-box fuzzing approaches to secure web applications: Survey," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 5, 2021.

[5] ——, "Effective and scalable black-box fuzzing approach for modern web applications," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 10 068–10 078, 2022.

[6] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella, "Improving Web Application Testing using testability measures," in *2009 11th IEEE International Symposium on Web Systems Evolution*. Edmonton, AB, Canada: IEEE, Sep. 2009, pp. 49–58.

[7] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, Feb. 2019.

[8] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler : Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758.

[9] A. Bainczyk, A. Schieweck, M. Isberner, T. Margaria, J. Neubauer, and B. Steffen, "ALEX: Mixed-Mode Learning of Web Applications at Ease," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, ser. LNCS, T. Margaria and B. Steffen, Eds. Cham: Springer, 2016, vol. 9953, pp. 655–671.

[10] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Computers & Security*, vol. 120, p. 102813, Sep. 2022.

[11] G. Benoit, "FuzzE 3.0," https://github.com/snail-unamur/FuzzE.

[12] ——, "Fuzzing highly-configurable web user interface: a odoo case study," Master's thesis, University of Namur, 2024.

[13] G. Benoit and X. Devroey, "Replication package of FuzzE, development of a fuzzing approach for Odoo's tours integration testing platform resources (1.0.0)," https://doi.org/10.5281/zenodo.14605997, 2025.

[14] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. ACM, 2011, p. 575–586.

[15] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "SUSHI: A Test Generator for Programs with Complex Structured Inputs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, May 2018, pp. 21–24, issue: i.

[16] M. Bruntink and A. van Deursen, "Predicting class testability using object-oriented metrics," in *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE, 2004, pp. 136–145.

[17] W. H. Burkhardt, "Generating test programs from syntax," *Computing*, vol. 2, no. 1, pp. 53–73, Mar. 1967.

[18] V. Dallmeier, B. Pohl, M. Burger, M. Mirold, and A. Zeller, "Webmate: Web application test generation in the real world," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 413–418.

[19] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic Data Flow Testing of Object Oriented Systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, May 2015, pp. 947–958, iSSN: 02705257.

[20] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. Van Deursen, "Generating Class-Level Integration Tests Using Call Site Information," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2069–2087, Apr. 2023.

[21] X. Devroey, A. Gambi, J. P. Galeotti, R. Just, F. Kifetew, A. Panichella, and S. Panichella, "JUGE: An infrastructure for benchmarking Java unit test generators," *Software Testing, Verification and Reliability*, vol. 33, no. 3, p. e1838, May 2023.

[22] P. Di, B. Liu, and Y. Gao, "MicroFuzz: An Efficient Fuzzing Framework for Microservices," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. Lisbon Portugal: ACM, Apr. 2024, pp. 216–227.

[23] Z. El Idrissi, "Reverse engineering variability for configurable systems using formal concept analysis: The odoo case study," Master's thesis, University of Namur, 2022.

[24] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, ser. ESEC/FSE '11. ACM Press, 2011, p. 416.

[25] A. Ganesh, K. Shanil, C. Sunitha, and A. Midhundas, "OpenERP/Odoo - An Open Source Concept to ERP Solution," in *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, 2016, pp. 112–116.

[26] V. Garousi, M. Felderer, and F. N. Kiluçaslan, "A survey on software testability," *Information and Software Technology*, vol. 108, pp. 35–64, Apr. 2019.

[27] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458.

[28] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[29] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[30] V. J. M. Manes, H. Han, C. Han, s. k. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 5589, no. c, 2019.

[31] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pp. 94–105, 2016, iSBN: 9781450343909.

[32] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* Victoria, BC, Canada: IEEE, 2003, pp. 260–269.

[33] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling ajax by inferring user interface state changes," in *2008 Eighth International Conference on Web Engineering*, 2008, pp. 122–134.

[34] S. Mouchawrab, L. C. Briand, and Y. Labiche, "A measurement framework for object-oriented software testability," *Information and Software Technology*, vol. 47, no. 15, pp. 979–997, Dec. 2005.

[35] Odoo, "Testing odoo," https://www.odoo.com/documentation/17.0/developer/reference/backend/testing.html#integration-testing, 2024, last access 15/02/2024.

[36] Odoo S.A., "Architecture odoo," https://www.odoo.com/documentation/17.0/developer/tutorials/server_framework_101/01_architecture.html, 2024, last access 12/05/2024.

[37] ——, "Open source ERP and CRM Odoo," https://www.odoo.com/, 2024, last access 13/09/2024.

[38] ——, "OWL framework," https://odoo.github.io/owl/, 2024, last access 19/09/2024.

[39] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating highly-structured input data by combining search-based testing and grammar-based fuzzing," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Dec. 2020, pp. 1224–1228.

[40] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, vol. 2. ACM Press, 2007, p. 815.

[41] R. Padhye, C. Lemieux, M. Papadakis, and Y. L. Traon, "Semantic Fuzzing with Zest," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, Beijing, China, 2019, pp. 329–340.

[42] A. C. R. Paiva, A. Restivo, and S. Almeida, "Test case generation based on mutations over user execution traces," *Software Quality Journal*, vol. 28, no. 3, pp. 1173–1186, Sep. 2020.

[43] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb. 2018, iSBN: 0098-5589 VO - PP.

[44] T. Parr, *The Definitive ANTLR 4 Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2013.

[45] M. Pezzè, K. Rubinov, and J. Wuttke, "Generating effective integration test cases from unit ones," *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, pp. 11–20, 2013, iSBN: 978-0-7695-4968-2.

[46] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, Sep. 1972.

[47] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "QExplore: An exploration strategy for dynamic web applications using guided search," *Journal of Systems and Software*, vol. 195, p. 111512, Jan. 2023.

[48] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2013, pp. 370–379.

[49] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015.

[50] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One Fuzzing Strategy to Rule Them All," in *ICSE '22: Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1634 – 1645.

[51] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," https://www.fuzzingbook.org/, 2024, last access 18/01/2024.