Mutation Testing for Task-Oriented Chatbots

Pablo Gómez-Abajo
Pablo.GomezA@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Sara Pérez-Soler
Sara.PerezS@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Pablo C. Cañizares
Pablo.Cerro@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Esther.Guerra@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
Juan.deLara@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

ABSTRACT

Conversational agents, or chatbots, are increasingly used to access all sorts of services using natural language. While open-domain chatbots – like ChatGPT – can converse on any topic, task-oriented chatbots – the focus of this paper – are designed for specific tasks, like booking a flight, obtaining customer support, or setting an appointment. Like any other software, task-oriented chatbots need to be properly tested, usually by defining and executing test scenarios (i.e., sequences of user-chatbot interactions). However, there is currently a lack of methods to quantify the completeness and strength of such test scenarios, which can lead to low-quality tests, and hence to buggy chatbots.

To fill this gap, we propose adapting mutation testing (MuT) for task-oriented chatbots. To this end, we introduce a set of mutation operators that emulate faults in chatbot designs, an architecture that enables MuT on chatbots built using heterogeneous technologies, and a practical realisation as an Eclipse plugin. Moreover, we evaluate the applicability, effectiveness and efficiency of our approach on open-source chatbots, with promising results.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Human-centered computing** → *Natural language interfaces.*

KEYWORDS

Task-oriented chatbots, Mutation testing, Dialogflow, Rasa, Botium

ACM Reference Format:
Pablo Gómez-Abajo, Sara Pérez-Soler, Pablo C. Cañizares, Esther Guerra, and Juan de Lara. 2024. Mutation Testing for Task-Oriented Chatbots. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661220

EASE 2024, June 18–21, 2024, Salerno, Italy

ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661220

1 INTRODUCTION

Conversational agents, or chatbots, are software systems that emulate human conversation. They can be either *open-domain*, like ChatGPT [31], or *task-oriented*. The former are able to engage in conversations on any topic, and are recently being built using generative artificial intelligence, especially large language models (LLMs) [16]. The latter are chatbots designed to solve a specific task, like booking a ticket, ordering a pizza, setting a medical appointment, or obtaining customer support. They are popular to access all sorts of services because of their ease of integration into many channels, such as social networks, websites, or smart speakers.

Unlike open-domain chatbots, building task-oriented chatbots using LLMs is still difficult [42]. Instead, their construction involves an explicit design of the chatbot topics, the conversation paths, the chatbot responses, and the interaction with external information services. There are many technologies to build task-oriented chatbots [33], like Google's Dialogflow [13], Amazon Lex [27], Microsoft Bot Framework [30], IBM Watson assistant [40] or Rasa [34].

Like any other software, task-oriented chatbots require thorough testing to ensure a proper behaviour. This entails testing for accurate recognition of the user intent, adequate continuation of the conversation paths, sensible chatbot responses, and correct calling to information system APIs. To this end, frameworks like Botium [4] and Rasa-test [35] enable the creation of test suites that describe expected user-chatbot interactions and may specify assertions. However, there is currently no way to measure the strength of a given test suite, which can lead to sub-optimal testing processes [7], and thus to buggy chatbots. We aim to fill this gap.

Mutation testing [11] (MuT) is a technique to assess the strength of software test suites. It involves injecting artificial errors into the program, and checking whether the test suite detects them. This way, the *mutation score* – the ratio of detected errors – is used to quantify the quality of the test suite. Faulty programs that remain undetected can be used to improve the original test suite and increase the mutation score. MuT has been applied to programming languages (e.g., Java [10], C [23]) and to other artefacts such as web services [14] or state machines [22], but not to chatbots.

Using MuT for chatbots brings several challenges. First, devising effective MuT operators to inject errors into chatbot designs. Second, reducing the potentially large number of generated mutants to avoid excessive testing times. Third, in practice, the MuT process should work across the many chatbot technologies in use today [33].

To tackle these challenges, we propose a suite of mutation operators for chatbot designs. They stem from an analysis of real faults in open-source chatbots, and the features of existing chatbot construction tools. To reduce the number of mutants, some operators use heuristics based on natural language processing (NLP) and *sentence embeddings* [15] that model different types of errors. Moreover, our operators are technology-independent, as they are implemented over a technology-agnostic chatbot design notation, called Conga [32]. Implementation-wise, we have extended the MuT framework Wodel-Test [17] to support MuT for chatbots. The resulting tool can import chatbots (e.g., built with Dialogflow and Rasa) into the Conga common format for their mutation, and supports the assessment of test suites created with Botium or Rasa-test. We have evaluated our proposal over a set of open-source chatbots and test suites, showing the suitability of our mutation operators, and the effectiveness and practicality of the approach.

2 BACKGROUND

This section provides background on the key elements of our proposal: chatbots (Sec. 2.1), chatbot testing (Sec. 2.2) and MuT (Sec. 2.3).

2.1 Task-oriented chatbots

Task-oriented chatbots are designed to solve particular tasks via conversation in natural language. Fig. 1 depicts their working scheme. The main components of their definition are the *intents*, which are the user intentions that the chatbot aims at recognising. Each of such intents declares a set of *training phrases* that exemplify how users might formulate the intent. Thus, when a user says an utterance in natural language to the chatbot (label 1 in the figure), the chatbot uses NLP to match the most likely intent (label 2). If no intent matches the utterance, the chatbot applies a default *fallback intent* (if it exists) with a predefined response (label 3a). For example, a chatbot for a coffee shop may have two main intents (one to allow users to order coffee, and another informing about the types of coffee served) and a fallback intent.

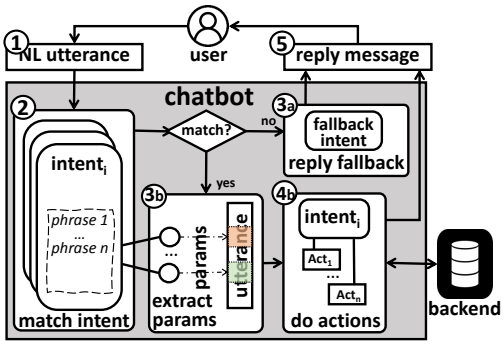


Figure 1: Workflow of task-oriented chatbots.

When the chatbot matches an intent, it may need to extract information from the user utterance. For this purpose, intents may declare *parameters*, with training phrases hinting to them (label 3b). For instance, an intent to order coffee may have two parameters (type and size of coffee), and a training phrase like *"I'd like a regular americano"* illustrates how to provide this information. Parameters are typed by *entities*, which can be predefined (e.g., dates) or domain-specific (e.g., coffee type). In addition, parameters can be optional or required. If the value of a required parameter is missing in an utterance, the chatbot prompts the user for its value.

After extracting the parameter values, the chatbot may need to access a backend to perform *actions* associated to the intent, like storing the order in the shop information system (label 4b). Next, the chatbot provides a reply message to the user (label 5), which is usually textual, but can contain elements like images and links.

Chatbots can also define conversation *flows* that interleave expected intents and chatbot responses. For example, once the user has ordered coffee, the chatbot may ask if the user wants to add something sweet to the order. Then, depending on the user answer, the conversation could follow different *paths*.

2.2 Testing task-oriented chatbots

Frameworks like Botium [4] and Rasa-test [35] give support for testing the conversation flow of chatbots by means of *test scenarios*, also called *convos*. A scenario defines a conversation path that a user and a chatbot are supposed to follow. For example, Listing 1 shows a convo for the coffee shop chatbot, defined with Botium. It specifies the expected responses of the chatbot (lines 3–4 and 7–8) to certain user utterances (lines 1–2 and 5–6). The test execution sends the specified user utterances to the chatbot, and compares its responses with those defined in the test scenario. Depending on the framework, the comparison may be customisable (e.g., exact match or conformance to a regular expression), and can include other assertions such as the expected matching intent or the presence of images in the responses. A scenario can also specify several user utterances at a given step (e.g., line 2), in which case, the convo is executed for each one of them, making all combinations with the user utterances of later steps.

```
1 #me
2 I'd like an americano
3 #bot
4 What size?
5 #me
6 Regular please
7 #bot
8 Your coffee is brewing!
```

Listing 1: Botium convo.

2.3 Mutation testing

MuT is a widely used technique to measure the ability of a test suite to detect faults. It relies on seeding artificial faults into the program under study, thus generating faulty program versions, called *mutants*. The faults are introduced by *mutation operators*, which are rules that produce valid syntactic variations of the program under study. The design of these operators typically follows the *competent programmer* hypothesis, which states that programmers often write code that is close to being correct [11].

The strength of a test suite can then be measured in terms of its ability to detect mutants. A mutant is detected, or *killed*, if the results of applying the same test case to both the original program and the mutant differ. Conversely, if all test cases yield the same result when applied to both the original and mutant programs, the mutant remains *alive*. If the behaviour of a mutant and the original program are indistinguishable, the mutant is said to be *equivalent*. The quality of a test suite is given by its *mutation score* (MS), which is the ratio between the number of killed mutants and the total number of mutants minus the equivalent ones: $MS = \frac{\#Killed}{\#Total - \#Equivalent}$.

The steps to compute the MS of a test suite are as follows. (i) First, the original program must compile successfully. (ii) Next, the test suite is run on the original program. If any test case fails, the program must be fixed. (iii) Once the program passes all test cases, a set of mutants is created by applying the mutation operators on the program, seeding one fault per mutant. (iv) The test suite is run against each mutant, and the results are compared with those of the original program. If the result of a test case differs for a mutant, the mutant is *killed*. (v) Once all mutants have been tested, the MS is calculated. Sometimes, it is necessary to manually analyse the live mutants to check for equivalence. If some mutants are *alive*, new test cases that kill them can be added to the test suite.

3 MUTATION TESTING FOR CHATBOTS

This section describes our proposal of MuT adapted to chatbots. It proceeds as explained in Sec. 2.3 for software systems in general, but using mutation operators specifically designed to inject artificial faults into the chatbot under study. Some of these operators calculate the similarity between (sets of) phrases, for which they use NLP techniques, like sentence *embeddings* [15]. The mutation operators produce a set of mutant chatbots, which are tested against a test suite to identify killed and live non-equivalent mutants, and calculate the MS of the test suite. As explained in Sec. 2.2, a test suite in our context comprises a set of test conversation scenarios with their input test utterances. The live mutants can be analysed to improve the test suite with new cases that kill those mutants.

Next, we focus on the more distinctive ingredient of our proposal: the chatbot mutation operators. We have designed a suite of 19 mutation operators that act on the main components of task-oriented chatbots: training phrases, intents, entities, chatbot actions, and conversation flows. Some of these operators (*DP*, *DPWP*, *DPWL*, *K2P*, *DPP*, *DFI*) emulate real faults identified in the analysis of a dataset of Rasa and Dialogflow open-source chatbots reported in [8] (see also https://github.com/Conga-dsl/ValidationSetup). The rest of operators are designed to cover the features of existing chatbot construction tools [33]. Next, we introduce the set of operators.

Operators for training phrases. According to [8], a common problem when building chatbots is the lack of training phrases that exemplify user utterances. This can lead to poorly trained intents that the chatbot will have difficulty recognising. Hence, we propose 6 operators that emulate this problem by deleting training phrases attending to several criteria. In addition, we propose 2 further operators that move training phrases between intents to simulate placing a phrase in the wrong intent by mistake.

- *Delete training phrase (DP).* This operator deletes a training phrase from an intent. It could be applied exhaustively for every phrase, which may produce many mutants, or randomly a certain number of times, which may prevent some interesting mutants from occurring. Instead, we propose a test reduction technique to select representative mutants attending to the similarity of the phrases within the intent, leading to two variants of the operator. The first variant, called DP_{max}, deletes the most representative phrase of the intent, i.e., the one which is closer to all other phrases in the intent. This emulates a developer who did not define small variations of each training phrase. The second variant, called DP_{min}, deletes the phrase that is most different from all others in the intent. This emulates a developer who made no effort to formulate the intent very differently, or who created the phrases by copy&paste with slight modifications. Phrase similarity can be computed by several means; in our implementation, we use sentence *embeddings* [15] (cf. Sec. 4.1).

  Example 1. Assume an intent to request information with 4 training phrases: *"What kinds of coffee are available?"*, *"What types of coffee can I order?"*, *"What can I drink here?"*, and *"Tell me what drinks there are"*. Using sentence embeddings, the average semantic similarity of each phrase with the others is 0.512, 0.538, 0.475, and 0.474, respectively. Hence, applying DP_{max} produces a mutant by deleting the second phrase, while DP_{min} would delete the last phrase.
- *Delete training phrases with required parameter (DPWP).* This operator removes all training phrases that use a required parameter of an intent. It models a developer who forgets to exemplify the use of a parameter by providing training phrases for it.
- *Delete training phrases with literal (DPWL).* This operator removes from an intent all training phrases that use a specific literal of an entity, modelling that the developer forgot to add phrases exemplifying the use of the literal.
- *Keep two training phrases (K2P).* This operator deletes all but two training phrases of an intent. It emulates the frequent situation of under-trained intents [8]. Following the same reasoning as for the *DP* operator, we propose to reduce the potentially high number of mutants that this operator would generate by heuristically selecting the two training phrases left using text similarity criteria. Thus, a first operator variant, called $K2P_{max}$, leaves the two most representative phrases, and the second one, called $K2P_{min}$, leaves the most different phrases.

  Example 2. For the intent in Example 1, $K2P_{max}$ yields a mutant that leaves the first two phrases of the intent, and $K2P_{min}$ leaves the last two phrases.
- *Move training phrase (MP).* This operator moves a training phrase from one intent to another. It emulates a developer confusing two intents and adding a training phrase to the wrong one. To reduce the combinatorics of cases, the phrase to be moved is selected using text similarity criteria, resulting in two variants of the operator. Given an intent, MP_{min} compares the average similarity of each of its phrases with those of all other intents, and moves its phrase with lowest similarity to the least similar intent. Conversely, MP_{max} moves its most similar phrase to the most similar intent.

Operators for intents. We have designed the following 4 operators that modify intents. They emulate different developer mistakes.

- *Delete intent parameter (DIP).* It removes a parameter from an intent, as well as all training phrases that use the parameter. Unlike operator *DPWP*, *DIP* is applicable to both required and optional parameters, and the parameter is deleted from the intent (*DPWP* preserves it).
- *Delete parameter prompt (DPP).* It deletes the prompt that a chatbot uses to ask for the value of a required parameter when the user does not provide its value. When this prompt does not exist, chatbots often employ an overly generic prompt predefined in the platform.

- *Set required parameter to optional (SPO).* It changes a required parameter to make it optional. Consequently, the chatbot will not ask the parameter value when absent from the user utterances. We do not consider the opposite case (i.e., making an optional parameter mandatory) as it tends to generate easy to kill mutants.
- *Delete fallback intent (DFI).* It deletes a fallback intent, making the chatbot unable to answer when it does not recognise the intent corresponding to a user utterance.

Operators for entities. We propose 2 operators to emulate mistakes in entity definitions. Each considers one of the two alternative ways to define the entity values: implicitly by a regular expression, or explicitly by enumerating the literals that are valid.

- *Change regular expression (CRE).* It changes the regular expression used to define the literals that an entity admits. Any of the mutation operators for regular expressions proposed in the literature can be used for this purpose [1].
- *Delete literal from entity (DLE).* It removes a literal from an entity definition, making the chatbot unable to extract that value from a phrase having a parameter typed by the entity.

Operators for actions. The following 3 operators model mistakes related to the definition of the actions that the chatbot performs.

- *Delete actions (DA).* This operator deletes all the actions to be performed in a chatbot interaction. The actions can be of different nature, such as accessing an external system, showing an image, or producing a text response.
- *Delete a parameter used in a response (DPR).* This operator modifies the textual response of a chatbot, removing the parameter values used in the response.

  Example 3. Upon the utterance *"How much is a large americano?"*, suppose the original chatbot answers *"A large americano is 6$"*, where *large* and *americano* are parameter values extracted from the utterance. If applying *DPR*, the resulting mutant chatbots would answer either *"A large is 6$"* or *"A americano is 6$"*.
- *Swap outputs (SO).* This operator swaps the output that the chatbot produces in two consecutive conversation steps. This is useful for testing multi-turn conversations that must occur in a certain order (e.g., as in Example 4 below).

Operators for conversation flows. The last 2 operators mutate the conversation flow, emulating errors that a developer may inadvertently make. The first operator is applicable to conversation designs that chain several interactions between the chatbot and the user (conversation depth > 1). The second one is for conversation designs that may bifurcate depending on the user response (conversation width > 1).

- *Delete conversation step (DCS).* It deletes a conversation step from a conversation flow, which can be an intermediate step, an initial step, or a final step.

  Example 4. Assume a conversation design that starts with the user intent to order coffee, follows with the intent to add cake to the order, and ends with an intent to pay (i.e., 3 steps). Applying *DCS* would yield three mutants: one that allows to order coffee and pay, another that allows to order cake and pay, and the last one to order coffee and cake but without paying.
- *Delete conversation bifurcation (DCB).* It removes a bifurcation in a conversation flow. This simulates the mistake of forgetting to define a possible user response to a chatbot prompt, which would lead to an alternative conversation flow.

  Example 5. Suppose a conversation design in which the chatbot asks the user for the payment method, so that if the user selects credit card, s/he is asked for the number, but if the user selects cash, s/he is just informed of the amount to pay (i.e., the conversation forks in two). Applying *DCB* would yield two mutants, each of them deleting one of the conversation alternatives.

Overall, the proposed operators cover the main elements of chatbot definitions and emulate mistakes by developers. In Sec. 5, we will evaluate their applicability, effectiveness, and efficiency. Note that we do not claim that this catalogue is exclusive, as more operators – and variants of them – may be added in the future.

4 ARCHITECTURE AND TOOL SUPPORT

We have created a MuT environment for Rasa and Dialogflow chatbots, available at https://gomezabajo.github.io/Wodel/Wodel-Test/. It realises the proposed mutation operators, and computes the MS of test suites built with Botium and Rasa-test. Next, Sec. 4.1 overviews the architecture of our solution, and Sec. 4.2 describes the tool.

4.1 Architecture

Fig. 2 outlines the developed architecture. It builds on Wodel-Test [17], an Eclipse framework for building MuT environments, which we extended for MuT of chatbots. Our solution is agnostic of the chatbot creation platform. This is achieved by parsing the chatbots into a technology-agnostic design notation, called Conga (step 1). Currently, our solution provides parsers from Dialogflow and Rasa chatbots into Conga models. The parsed chatbot model is automatically enriched with metrics on the similarity of its training phrases, computed using TensorFlow's sentence embedding (step 2). The mutation operators are implemented using the Wodel DSL that is available in Wodel-Test. Applying the operators on the annotated chatbot model yields a set of chatbot model mutants (step 3). These mutants are transformed back into chatbot implementations using Conga's code generators (step 4). This way, it is possible to run the test suite under study against them (step 5). Our solution supports test suites defined with Botium or Rasa-test.

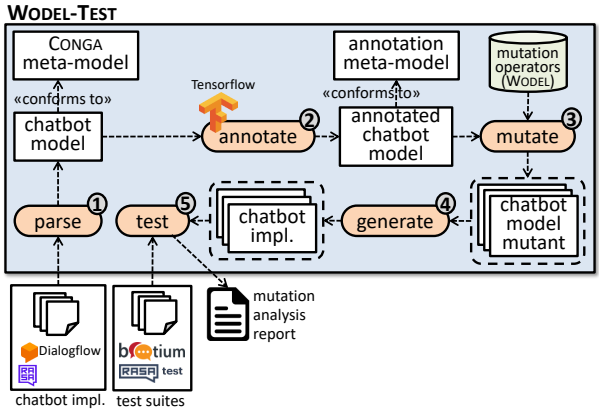


Figure 2: Schema of our solution for MuT of chatbots.

Next, we provide more details on Conga (Sec. 4.1.1), the model annotation process (Sec. 4.1.2), and the DSL Wodel (Sec. 4.1.3).

4.1.1 Conga. Conga [32] is a technology-agnostic notation to design chatbots independently of the chatbot implementation tool. It provides design primitives tailored for task-oriented chatbots (intents, training phrases, parameters, entities, conversation flows...) which abstract away low-level accidental implementation details. Following model-driven engineering principles, its abstract syntax is defined by a meta-model, to which Conga models must conform to. More details about Conga can be found at [32].

To make our MuT environment generic, the mutation operators are defined for Conga models and not for a specific chatbot technology. Adding support for a particular chatbot technology requires building a parser from the technology into Conga, and a code generator from Conga into the technology (steps 1 and 4 in Fig. 2). Currently, both Dialogflow and Rasa are supported.

4.1.2 The semantic similarity annotation model. To reduce the number of mutants and optimise the MuT process with fewer redundancies, our tool enriches the Conga models with information about the similarity between the training phrases of the original chatbot (step 2 in Fig. 2). These similarity annotations conform to the meta-model in Fig. 3. Specifically, *SemanticSimilarity* stores the semantic *similarity* of each pair of *TrainingPhrases* in the chatbot. This is calculated by vectorising the phrases with TensorFlow's sentence embedding and calculating the cosine similarity between vectors. In addition, based on the average similarity, *IntentIntentValue* records the two *TrainingPhrases* from each intent (*intent1*) that have the highest (*max*) and lowest (*min*) similarity to every other intent (*intent2*), while *IntentValue* (a particular case of the latter, reified for efficiency) stores the two most/least similar phrases of each intent.

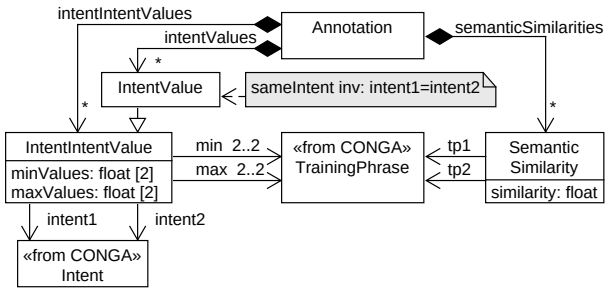


Figure 3: Semantic similarity annotation meta-model.

4.1.3 Wodel. We have used the Wodel DSL [18] to specify the operators of Sec. 3 and apply them to the annotated Conga models (step 3 in Fig. 2). Wodel provides mutation primitives to *select*, *create*, *clone*, *modify*, *retype* (i.e., modify the type of an object to a sibling type), and *delete* elements from models. It features two execution modes: *exhaustive*, which generates all possible mutants for the given mutation operators, and *stochastic*, which stablishes a maximum number of mutants to be generated.

As an example, Listing 2 shows the DP_{max} operator, which deletes the most representative phrase of an intent. Line 1 specifies the exhaustive execution mode, the output folder in which to store the mutants, and the input folder that contains the seed models. Line 2 declares the URI or location of the meta-model the models to be mutated conform to (Conga's meta-model in this case). Lines 3–4 declare the name (*nlp*) and meta-model of the annotation model. Finally, lines 7–12 define the DP_{max} operator, which selects one *IntentValue* object (line 8) that annotates one intent (line 9). Then, the operator removes the phrase with maximum average similarity to all other phrases, pointed by reference *max[0]* (line 11, cf. Fig 3).

```
1  generate exhaustive mutants in "data/out/" from "data/model/"
2  metamodel "http://botGenerator/1.0"
3  with resources from
4    { nlp = "data/model/Annotation" metamodel="http://botAnnot" }
5
6  with blocks {
7    DP_max "Deletes the most representative phrase of an intent" {
8      tpi = select one IntentValue from nlp resources
9      i = select one Intent where self == tpi→intent1
10     li = select one LanguageIntent in i→inputs
11     remove one TrainingPhrase from li→inputs where self == tpi→max[0]
12   }
13 ... }
```

Listing 2: Wodel program for operator DP_{max}.

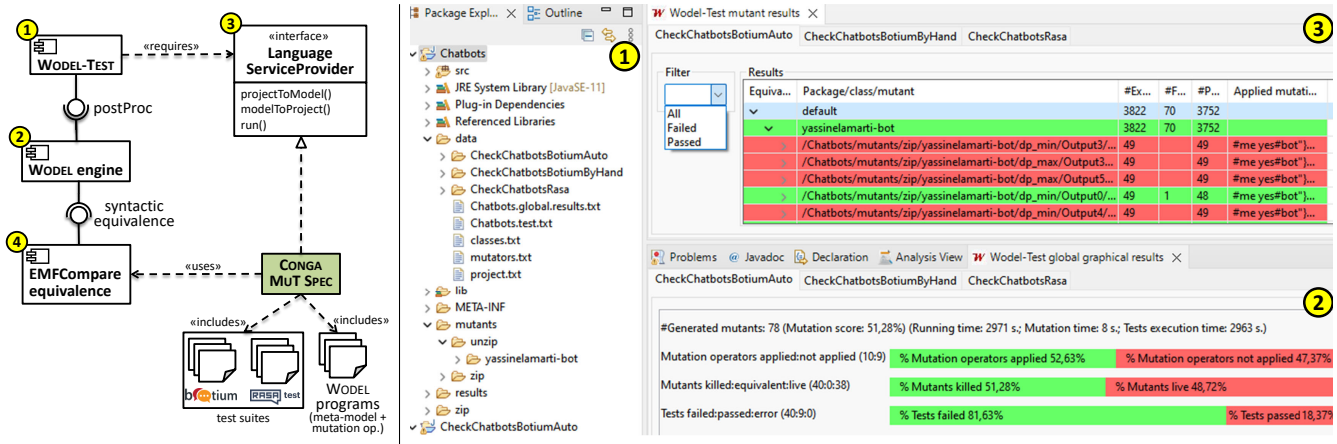
4.2 Tooling

Fig. 4a shows the technical architecture of Wodel-Test, and our extension for MuT of chatbots. Wodel-Test (label 1) is a post-processing extension to the Wodel engine [18] (label 2), built as an Eclipse plugin. To extend Wodel-Test for our purposes, we had to instantiate its extension point `LanguageServiceProvider` (label 3), which requires implementing three methods:

- `projectToModel`: to convert the artefact to be mutated (a chatbot) into a model. We use the Conga parser for this purpose.
- `modelToProject`: to convert a model into a domain artefact. In this work, we use the Conga generator to serialise a model conformant to the Conga meta-model into the corresponding Rasa or Dialogflow implementation.
- `run`: to specify how to execute the programs against the test suite. In this work, we support the execution of Rasa-test and Botium test scenarios, and allow applying multiple test suites to the same set of mutants in the same execution.

To speed up the MuT process, we have extended Wodel-Test to run the mutants in parallel. Currently, the detection of equivalent mutants is purely syntactical by model comparison (label 4). We plan to provide additional automated support in future work.

Fig. 4b shows a screenshot of Wodel-Test, displaying the results after applying the MuT process to a Rasa chatbot. Label 1 corresponds to the Eclipse project explorer, which contains the project under test and the used test-suite projects. Label 2 shows the *global view* of results which reports the MS, the running time of the MuT process, the number of applied mutation operators, the number of killed, equivalent and live mutants, and the number of failed and passed tests. This view also displays this information in percentages using coloured bar graphs (green for killed/failed, red for live/passed). Label 3 shows the *mutants view*, with the results of the tests passed and failed by each mutant. Both views offer interactive actions, such as a pop-up window to show the applied mutation operators or the live mutants, in the case of the *global view*, or a



(a) Architecture of Wodel-Test

(b) The chatbot MuT tool built with Wodel-Test

Figure 4: Mutation testing of chatbots with Wodel-Test.

filter to show only the failed or the passed tests, in the case of the *mutants view*. In addition, these views group the information by tabs, one for each test-suite project included in the MuT process. Wodel-Test also enables persisting these results in a plain text file.

5 EVALUATION

To assess the value of our proposal for MuT for chatbots, we aim at answering the following research questions (RQs):

RQ1: *How applicable are the defined mutation operators?*
RQ2: *How effective are the mutation operators?*
RQ3: *How effective is the mutation testing process?*
RQ4: *How efficient is the mutation testing process?*

RQ1 is important to understand how many mutants of a chatbot are expected to be generated, which operator types are more applicable, and which ones generate the most mutants. The execution time of the MuT process is a concern, so it may be necessary to prioritise or filter out mutants from operators that generate many of them. As for RQ2, its goal is to determine the usefulness of each operator type, as operators that produce mutants that are always killed are not useful. RQ3 is concerned with the MS (i.e., the quality) of realistic test suites. If MSs were always close to 100% because most mutants are killed, this would indicate that MuT is of little value. Finally, RQ4 has practical relevance, as it measures the time of the whole MuT process.

5.1 Experiment setup

To answer the RQs, we applied the MuT process to a dataset of 15 third-party chatbots and their available test suites.

We selected chatbots from GitHub, favouring those with a test suite available. For simplicity, we only selected Rasa chatbots since Dialogflow chatbots must be deployed in Google's cloud, so their testing cannot be executed locally. The chatbot selection criterion was maximising their heterogeneity, considering aspects like chatbot size, conversation length, or vocabulary. In Table 1, columns 2–9 summarise the features of the selected chatbots: number of training phrases (and the average per intent), intents (and those that are fallback), entities, intent parameters (and how many of them are required), conversation flows, average length of the conversations, number of conversation bifurcations, and number of chatbot actions (and how many of them are text messages).

Columns 10 to last of Table 1 characterise the test suites for the selected chatbots. All chatbots – except `bikeShop` and `Covid19_tracer` – had a test suite built by a developer available, written either with Botium (column *Botium by hand*) or with Rasa-test (column *Rasa-test*). In addition, Botium permits generating an initial test suite from a chatbot automatically, which testers can modify and increase. We have generated this test suite for all chatbots (column *Botium automatic*). For each test suite, the table shows the number of convos (in parenthesis, the number of user-chatbot interactions of the longest conversation scenario), and the number of test cases (as each convo is executed for every test user utterance provided). The *Botium automatic* test suites have just 1 user-chatbot interaction, while the human-written Botium and Rasa-test suites reach up to 4 interactions. The *Botium automatic* test suites use the training phrases as test utterances, and so, the number of tests is higher than for Rasa-test. Still, the manually created Botium test suites have the highest number of tests (except for `Spaceonova`).

5.2 RQ1: Applicability of mutation operators

To measure the applicability of the operators, we applied them in exhaustive mode to all chatbots in the dataset. Table 2 shows a summary of the results. The column group *Applicability* reports on the number of mutants generated (*#Mutants*), the number of chatbots where the operator produced at least one mutant (*#Chatbots*), and the ratio of generated mutants per applicable chatbot (*Ratio*).

Applicability. Seven operators were applicable on all chatbots: DP_{max} and DP_{min} (which delete one phrase from intents), $K2P_{max}$ and $K2P_{min}$ (which delete all but 2 phrases from intents), MP_{max} and MP_{min} (which move phrases between intents), and *DA* (which deletes chatbot actions). This was to be expected, since all chatbots have training phrases and actions, so that operators deleting or moving phrases, or deleting actions, are applicable.

	Chatbots								Test Suites					
	Train. Phr.	Intents		Param.		Avg Path	Path	Actions	Botium automatic		Botium by hand		Rasa-test	
Chatbot	(Avg)	(Fallb.)	Entities	(Req.)	Flows	Length	Bifur.	(Text)	Convos (len.)	Tests	Convos (len.)	Tests	Convos (len.)	Tests
256644	63 (5.7)	11 (0)	0	1 (0)	10	1.10	0	12 (11)	11 (1)	63	10 (2)	107	-	-
bikeShop	13 (2.6)	5 (1)	1	3 (1)	4	1.25	0	8 (6)	4 (1)	13	-	-	-	-
Covid19_tracer	18 (2.25)	8 (0)	0	1 (0)	4	1.5	0	7 (5)	3 (1)	18	-	-	-	-
data-mining	1,128 (26.86)	42 (0)	0	0 (0)	38	1.12	2	43 (41)	42 (1)	1,128	40 (3)	1,801	-	-
diagrams2ai	39 (5.5)	7 (1)	0	0 (0)	3	2.2	2	6 (5)	6 (1)	39	4 (4)	2,212	-	-
dusbot	406 (31.23)	13 (1)	0	0 (0)	13	1	0	17 (7)	11 (1)	401	-	-	6 (3)	6
e2e-bot	33 (8.25)	4 (0)	0	0 (0)	2	3.5	0	3 (3)	4 (1)	33	-	-	4 (4)	4
Email-WhatsApp	48 (5.3)	9 (1)	0	0 (0)	4	2	3	11 (6)	8 (1)	48	10 (3)	1,319	-	-
h4h-chatbot	100 (10)	10 (0)	0	0 (0)	10	1	0	10 (10)	9 (1)	100	-	-	9 (1)	9
lankbanfinance	542 (10.04)	54 (1)	0	0 (0)	51	10.2	0	55 (5)	52 (1)	542	-	-	47 (1)	47
legal-alien	270 (7.5)	36 (0)	0	1 (0)	32	1.12	2	37 (36)	7 (1)	72	-	-	3 (3)	3
personal-bot	91 (9.2)	10 (0)	1	5 (0)	5	1.6	0	11 (8)	6 (1)	63	-	-	3 (2)	3
Rasa-demo	87 (7.9)	11 (0)	0	1 (0)	7	15.56	2	10 (7)	11 (1)	87	-	-	7 (3)	7
Spaceonova	149 (9.31)	16 (0)	0	0 (0)	14	1	0	15 (14)	16 (1)	147	13 (1)	131	-	-
yassinelamarti	49 (5.44)	9 (0)	0	0 (0)	4	2	2	8 (7)	9 (1)	49	6 (3)	725	7 (3)	7
Average	202.4	16.33	0.13	0.8	13.4	3.08	0.87	16.87	13.27	186.87	13.83	1,049.17	10.75	10.75

Table 1: Measurements of the selected chatbots and test suites.

	Applicability			Resilience		Stubbornness	
Operator	#Mutants	#Chatbots	Ratio	#Killed	MS	#Tests	%
Operators for training phrases							
DP$_{max}$	229	15	15.3	90	39.3	1,828	19.9
DP$_{min}$	229	15	15.3	111	48.5	2,973	32.4
DPWP	3	1	3	2	66.7	3	23.1
DPWL	5	2	2.5	3	60	1	1.3
K2P$_{max}$	218	15	14.5	167	76.6	5,945	64.8
K2P$_{min}$	222	15	14.8	162	73	6,774	73.8
MP$_{max}$	228	15	15.2	178	78.1	3,771	41.1
MP$_{min}$	229	15	15.3	183	79.9	3,357	36.6
Operators for intents							
DIP	9	6	1.5	6	66.7	98	22.5
DPP	3	1	3	0	0	0	0
SPO	3	1	3	0	0	0	0
DFI	5	5	1	2	40	31	0.7
Operators for entities							
CRE	0	0	0	0	-	0	-
DLE	4	2	2	2	50	1	1.3
Operators for actions							
DA	262	15	17.5	199	76	7,470	81.3
DPR	7	3	2.3	1	14.3	60	21.7
SO	38	12	3.2	34	89.5	5,061	60.3
Operators for conversation flows							
DCS	39	12	3.3	34	87.2	5,083	60.6
DCB	24	6	4	23	95.8	5,018	67
Average	92.5	8.2	11.3	63	68.2	2,498.7	50.5

Table 2: Operator applicability, resilience and stubbornness.

The only operator that could not be applied was *CRE* (which changes a regular expression). The operators applied on 1/3 of the chatbots or less were *DPWP* (deletes a phrase with required parameters), *DPP* (deletes a parameter prompt) and *SPO* (sets a required parameter to optional), applied on 1 chatbot; *DPWL* (deletes a phrase with literals) and *DLE* (deletes a literal), applied on 2 chatbots; *DPR* (deletes a parameter used in a response) applied on 3 chatbots; and *DFI* (deletes a fallback intent) applied on 5 chatbots. These operators target more specific chatbot features, like required parameters (*DPWP*, *DPP*, *SPO*), entities with literals (*DPWL*, *DLE*), fallback intents (*DFI*), or chatbot answers that use parameter values (*DPR*). Hence, the operators were applied on chatbots with those features (cf. Table 1).

Generated mutants. The operator that produced the highest ratio of mutants per chatbot was *DA*, with a ratio of 17.5. Other operators with high mutant generation ratios are *DP*$_{max}$ (15.3), *DP*$_{min}$ (15.3), *MP*$_{min}$ (15.3), *MP*$_{max}$ (15.2), *K2P*$_{min}$ (14.8) and *K2P*$_{max}$ (14.5). The number of mutant chatbots generated with these operators is close to the number of chatbot intents, or slightly lower as these operators do not apply to intents that have two or less training phrases or are fallback. The operators with the lowest mutant generation ratios are *DFI* (ratio 1), *DIP* (1.5), *DLE* (2), *DPR* (2.3), and *DPWL* (2.5).

Operator type. The operators for training phrases generated the highest number of mutants (77.6% of the total), followed by those for actions (17.5%), flows (3.6%), intents (1.1%) and entities (0.2%). The operators for phrases also had the highest average mutant generation ratio (12), followed by those for actions (7.7), flows (3.6), intents (2.1) and entities (1).

Answer to RQ1. All our operators are applicable in chatbots, with the operators for training phrases being the most applicable within and across chatbots.

5.3 RQ2: Operator effectiveness

We evaluate the effectiveness of the operators by measuring their *resilience* (i.e., the MS achieved by the operator) and *stubbornness* [19] (i.e., % of test suites that kill the mutant). An operator yielding a high MS implies low resilience. A mutant is stubborn if very few test cases kill it. An operator producing stubborn mutants is stubborn.

Table 2 displays the results, which are computed jointly for all available test suites of each chatbot (cf. Table 1). The column group *Resilience* shows the number of mutants killed, and the MS. The column group *Stubbornness* shows the number of tests that killed each mutant, and the percentage this number represents relative to the total number of tests executed on those mutants.

Resilience. MSs range from 0% (for *DPP* and *SPO*) to 95.8% (for *DCB*, which deletes a conversation bifurcation). The average MS is 68.2%. Some operators yield a MS higher than 80%, so their resilience is low. This is the case of *DCB* (95.8%), *SO* (89.5%) and *DCS* (87.2%). In contrast, other operators produce hard-to-kill-mutants, like *DPP* (0%), *SPO* (0%), *DPR* (14.3%), DP_{max} (39.3%) and *DFI* (40%). Regarding operator variants, MP_{max} and MP_{min} yield similar MSs (78.1% and 79.9%, a difference of 1.8), while the difference for the other variants is more significant: 3.6 for *K2P* (76.6% vs 73%) and 9.2 for *DP* (48.5% vs 39.3%).

Stubbornness. The percentage of tests that kill each mutant ranges from 0% (for *DPP* and *SPO*) to 81.3% (for *DA*), with an average of 50.5%. The stubbornest operators (low % values) are *DPP* (0%) *SPO* (0%), *DFI* (0.7%), *DLE* (1.3%), *DPWL* (1.3%), and DP_{max} (19.9%). The least stubborn operators (high % values) are *DA* (81.3%), $K2P_{min}$ (73.8%), *DCB* (67%), $K2P_{max}$ (64.8%), *DCS* (60.6%) and *SO* (60.3%). Looking at operator variants, the difference between MP_{max} and MP_{min} is smaller (4.5%) than the one for *K2P* (9%) and *DP* (12.5%).

Answer to RQ2. Different operators show different effectiveness. The operators for intents produce the hardest-to-kill mutants (overall MS of 40%), followed by those for entities (50%), phrases (65.7%), actions (76.2%) and flows (90.5%). The operators for entities produced the stubbornest mutants (1.3%), followed by intents (2.6%).

5.4 RQ3: MuT effectiveness

We assess the effectiveness of the MuT process by computing the MS of the test suites. Table 3 presents the results. Columns 2–4 show the MS of the test suites for each chatbot. The column *Overlap* shows the percentage of mutants that all available test suites killed (when there is more than one test suite). The column *Overall* displays the MS resulting from the combination of all available test suites.

We observe that only 4 test suites obtained a 100% MS (all of them of type *Botium-by-hand*). The test suites created manually with *Rasa-test* obtained the lowest scores (with 6.3% as the lowest value). Generally, the test suites created by hand with Botium had the highest MS (with an average of 94.3%), followed by those generated automatically by Botium (45.5%) and manually built with Rasa-test (19.6%). We found good correlation (0.67) between the number of test cases of each test suite (cf. Table 1) and the obtained MS.

The overlapping between *Botium-by-hand* and *Botium-automatic* test cases is relatively high (peaking at 66.4%), but low between *Botium-automatic* and *Rasa-test*, as 5 cases have no overlapping or is close to 0 (0.3% for `lankbanfinance`). This is interesting as it means that the latter test suites are complementary. This is likely because the *Botium-automatic* suites focus on testing intents with their training phrases but not the conversation flow (the length of all convos is 1, cf. Table 1). Instead, the *Rasa-test* suites focus more on the conversation paths (with lengths up to 4, cf. Table 1).

Answer to RQ3. The results show that MuT is effective for chatbots. Only 4 out of 29 test suites had a 100% MS, with an overall MS of 68.2%. MuT can help understanding the strength of test suites. For instance, in our dataset the tests generated by Botium were weaker than those manually built, and complementary to the Rasa tests.

Chatbot	Botium auto.	Botium hand	Rasa-test	Overlap	Overall
`256644`	52.3	78.9	-	52.3	78.9
`bikeShop`	21.3	-	-	-	21.3
`Covid19_tracer`	37.5	-	-	-	37.5
`data-mining`	63.7	100	-	63.7	100
`diagrams2ai`	56	100	-	56	100
`dusbot`	53	-	10.6	0	63.6
`e2e-bot`	46.2	-	35.9	0	82.1
`Email-WhatsApp`	41.1	100	-	41.1	100
`h4h-chatbot`	63.4	-	15	0	78.4
`lankbanfinance`	41.4	-	12.9	0.3	54
`legal-alien`	11.4	-	6.3	0	17.6
`personal-bot`	40.3	-	73.6	30.6	83.4
`Rasa-demo`	49.5	-	20.5	1.1	68.9
`Spaceonova`	75.8	76.9	-	66.4	86.4
`yassinelamarti`	51.3	100	48.8	15.4	100
Average	45.5	94.3	19.6	23.2	68.2

Table 3: Mutation score (%) by test suite.

To calculate the MS, we automatically checked and discarded syntactically equivalent mutants. Moreover, we randomly sampled a subset of the live mutants and manually checked if they were semantically equivalent. We found that the sampled mutants created by operators for intents, entities, actions and conversation flows were not equivalent. In the case of mutants produced by training phrase operators, we assessed that deleting or moving phrases from an intent reduced the likelihood (*confidence*) that the NLP engine would select that intent when saying the deleted phrases to the chatbot. For most intents, the confidence decrease was between 30–60%, so we hypothesise that there are test utterances (other than those in the test cases) that would cause the chatbot to select a different intent and kill the mutant, implying that the mutant is not semantically equivalent. Instead, for intents related to greetings/farewell, which have many similar short phrases (e.g., "Hello", "Hi", "Hey"), the decrease was lower. We argue that this uncertainty would not distort the results much, as most live mutants produced by phrase operators (73%) come from only two chatbots (`lankbanfinance` and `legal-alien-chatbot`). In future work, we plan to incorporate automatic heuristics for detecting semantically equivalent mutants based on confidence decrease.

5.5 RQ4: MuT efficiency

To evaluate the efficiency of MuT for chatbots, we measured the execution time. The MuT process comprises 3 phases: a preprocessing phase that computes the similarity annotation model used by the heuristics, a chatbot mutation phase, and a testing phase that runs the test suites against the mutants and calculates the MS. Columns 2–4 of Table 4 show the execution time for each phase, and column 5 shows the total MuT time. Then, columns 6–8 show the number of mutants, test cases, and the time per mutant (TPM), respectively. Times are shown in seconds. We run the experiment on a windows machine with an i9-13900 2CPU, 32GB of RAM, and using dynamic parallelisation with 16 threads.

Chatbot	Prepr.	Mutat.	Testing	Total	#Mut.	#Tests	TPM
256644	18	464	2,825	3,307	90	170	37
bikeShop	2	89	380	471	47	13	10
Covid19_tracer	1	3	250	255	32	18	8
data-mining	5,940	25,668	39,879	71,487	308	2,929	232
diagrams2ai	11	6	5,286	5,303	59	2,251	90
dusbot	799	2,365	6,861	10,026	85	407	118
e2e-bot	5	5	557	568	39	37	15
Email-WhatsApp	11	7	26,289	26,308	78	1,367	337
h4h-chatbot	49	470	3,089	3,608	60	109	60
lankbanfinance	1,418	8,777	46,555	56,750	365	589	156
legal-alien	337	208	16,818	17,364	256	75	68
personal-bot	36	378	1,988	2,403	72	66	33
Rasa-demo	34	430	2,765	3,229	93	94	35
Spaceonova	90	39	2,011	2,140	95	278	23
yassinelamarti	11	8	2,963	2,983	78	781	38
Average	584	2,595	10,568	13,747	117.2	612.3	117

Table 4: MuT efficiency. TPM = time per mutant. Times in s.

The total MuT time ranges from a few minutes (4 for Covid19_tracer) to almost 20 hours (for data-mining), with 10 chatbots finishing in less than 90 minutes. The per-mutant time ranges from 8 seconds (for Covid19_tracer) to 337 seconds (for Email-WhatsApp).

Regarding each phase, the preprocessing time has quadratic dependence with the number of training phrases (adjusted by $4.95 \times \#phrases - 61.63 \times \#phrases$, with R^2=0.9). This is expected, as this phase calculates the similarity between each two phrases. This is typically the fastest phase (except for diagrams2ai, Email-WhatsApp, legal-alien, Spaceonova and yassinelamarti). Mutation is the second fastest phase, with times from 3 seconds (for Covid19_tracer, generating 32 mutants) to more than 7 hours (for data-mining, generating 308 mutants). The most expensive phase is testing, ranging from 4 minutes (for Covid19_tracer) to almost 13 hours (for lankbanfinance). As expected, there is high correlation (0.71) between the testing phase time and the number of test executions (given by the product of the numbers of tests and mutants).

Answer to RQ4. In line with the results of MuT for other artefacts, like code [24], MuT for chatbots is time-consuming. Still, its cost may be admissible in practice, as the whole MuT process took less than 90 minutes for 67% of the analysed chatbots. Some of the most time-consuming chatbots had also the largest test suites (e.g., lankbanfinance, dusbot, legal-alien, cf. Table 1), but their MSs were not particularly high (54, 63.6 and 17.6), which shows the potential of MuT to improve the test suites.

5.6 Threats to validity

Construct validity. We rely on parsing chatbots into Conga, and generating an implementation back. Conga's Rasa parser detects when the chatbot behaviour is not preserved [32]. As an additional sanity check, we applied an *empty* mutation to all chatbots, checking that they behaved as the original in all test cases.

Internal validity. To ensure that our parsers and generators to translate between Rasa and Conga are correct, we applied differential testing to them, and confirmed that the translation preserves the chatbot information and behaviour.

External validity. We only consider Rasa chatbots due to the availability of test suites that can be run locally. Hence, our results need to be confirmed with chatbots of other platforms, like Dialogflow. Similarly, the experiment used a reduced dataset of chatbots, so its results could be strengthened by using more chatbots. Nevertheless, compared to the bigger dataset used in [8], our chatbots have similar size in terms of intents (average 16.3 in our case vs 15.4 in [8]) and entities (1.13 vs 0.39), and higher conversation complexity (13.4 vs 6.96 average flows).

6 RELATED WORK

Mutation testing. MuT was initially proposed for source code, and so, mutation operators for many programming languages exist, such as C [23], or Java [10]. MuT has also been applied to other artefacts like state machines [22], or model transformations [20]. Overall, the relevance of MuT is evinced by the diversity of domains where it has been used, including cyber-physical systems [39], cloud and HPC [9], and blockchain [2]. We follow this line of works but for a novel domain: chatbots. Since chatbots manage specialised concepts (e.g., training phrases, intents, flows), we have defined mutation operators specifically for them, assessing their value.

MuT can produce a large number of mutants, leading to long testing times. Test case reduction techniques decrease this number by selecting the most relevant mutants. For example, Wei *et al.* [41] apply spectral clustering to discard mutants based on their similarity. Likewise, we measure the distance between training phrases to consider the closest or farthest ones when generating the mutants.

Equivalent mutants can affect the efficiency and accuracy of the MuT process. Some techniques for their detection are data flow patterns [26], trivial compiler equivalence [25], and automata language equivalence [12] (see [29] for a survey). In the future, we will study techniques for detecting equivalent chatbots. In addition, MuT approaches for deep neural networks [38] may help us manage the stochasticity of the NL models in chatbots to enable more informed decisions on whether or not a mutant is killed.

Testing for chatbots. Chatbots can be tested at different levels [7]. The most basic one is *unit testing*, which seeks to ensure that intents and their training phrases are properly defined. Most commercial platforms, like Dialogflow, offer a console to test individual intents manually. Since defining test data is time-consuming, researchers have devised ways to generate test utterances that simulate user phrases. For example, Charm [6] provides mutation operators for sentences that emulate misspellings and rephrase them by using synonyms, swapping numbers for words, or translating back and forth into a pivot language. Applying these operators to the intent training phrases generates test utterances. Božić [5] mutates input phrases using metamorphic relations, and generates input test utterances from an initial test suite. Similarly, DialTest [28] applies synonym replacement, back translation and word insertion, and identifies the test cases most likely to find defects in a chatbot. Overall, all these works use operators to enlarge the set of test utterances by generating variants of the input test phrases. Instead, our operators are specific to MuT, covering aspects of the chatbot design like conversation, interaction, or parameters.

Complementarily, some frameworks help to test the conversation flow. Most permit running test scenarios and are specific for a

chatbot technology, such as Rasa-test for Rasa chatbots, or Bot Tester [3] for chatbots created with Bot Builder. In contrast, Botium is independent from the underlying chatbot technology, providing connectors to several platforms. Instead of testing, Silva *et al.* [37] use model checking on the conversation design to ensure properties of interest in the conversation. This can be done at the design phase, but still requires from testing to ensure a proper handling of the user utterances. In our case, our mutation operators are technology-independent since they are defined atop a neutral chatbot design language. Our tool supports test scenarios defined with Botium and Rasa-test, but it can be extended to support other frameworks.

Other chatbot aspects that can be tested include performance, security or usability [7]. Usability testing often requires human judgement [36] (which is difficult to automate) or the existence of a record of real user-chatbot dialogues [21]. Alternatively, some researchers propose design metrics that may hint potential usability problems [8]. Since MuT is oriented to assess the quality of test cases, our work does not cover these other kinds of testing.

Overall, we can find different proposals and tools to test chatbots. However, to the best of our knowledge, there are no proposals that apply MuT to assess the quality of chatbot test suites.

7 CONCLUSIONS AND FUTURE WORK

Task-oriented chatbots have raised in popularity, and many technologies for their construction have appeared. However, there is currently no mechanism to assess the quality of their test suites. To fill this gap, we have proposed an approach to MuT of chatbots, including a catalogue of 19 MuT operators emulating faults in training phrases, intents, entities, chatbot actions, and conversation flows. We have implemented a MuT environment for chatbots that is technology-independent and supports test scenarios from Botium and Rasa-test. We have assessed the applicability, effectiveness and efficiency of the approach on a dataset of 15 chatbots and 29 test suites, obtaining positive results. Overall, we found room for improvement in 86% of the test suites, with costly but acceptable running times (less than 90 minutes for 67% of the chatbots).

In the future, we will automate the detection of semantically equivalent mutants (e.g., using confidence decrease heuristics), and the synthesis of tests able to kill live mutants. Finally, we will adapt our approach to LLM-based agents.

DATA AVAILABILITY

The datasets of chatbots and test suites, the experiment data, and the results are available at https://zenodo.org/records/10938786.

ACKNOWLEDGMENTS

Work funded by the Spanish MICINN with projects TED2021-129381B-C21, PID2021-122270OB-I00, and RED2022-134647-T.

REFERENCES

[1] P. Arcaini, A. Gargantini, and E. Riccobene. 2019. Fault-based test generation for regular expressions by mutation. *Softw. Test. Verification Reliab.* 29, 1-2 (2019).
[2] M. Barboni, A. Morichetta, and A. Polini. 2022. SuMo: A mutation testing approach and tool for the Ethereum blockchain. *J. Syst. Softw.* 193 (2022), 111445.
[3] Bot Tester. last access in 2024. https://github.com/microsoftly/BotTester.
[4] Botium. last access in 2024. https://www.botium.ai/.
[5] J. Bozic. 2022. Ontology-Based Metamorphic Testing for Chatbots. *Softw. Qual. J.* 30, 1 (2022), 227–251.
[6] S. Bravo-Santos, E. Guerra, and J. de Lara. 2020. Testing Chatbots with Charm. In *QUATIC (CCIS, Vol. 1266)*. Springer, 426–438.
[7] J. Cabot et al. 2021. Testing Challenges for NLP-intensive Bots. In *BotSE@ICSE*. IEEE, 31–34.
[8] P. C. Cañizares, J. M. López-Morales, S. Pérez-Soler, E. Guerra, and J. de Lara. 2024. Measuring and clustering heterogeneous chatbot designs. *ACM Trans. Softw. Eng. Methodol.* 33, 4 (2024), 90:1–90:43.
[9] P. C. Cañizares, A. Núñez, and M. Merayo. 2018. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *J. Syst. Softw.* 143 (2018), 187–207.
[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *ISSTA* (Saarbrücken, Germany). ACM, 449–452. https://doi.org/10.1145/2931037.2948707
[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41.
[12] X. Devroey et al. 2018. Model-based mutant equivalence detection using automata language equivalence and simulations. *J. Syst. Softw.* 141 (2018), 1–15.
[13] Dialogflow. last access in 2024. https://dialogflow.com/.
[14] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. 2008. Mutation operators for WS-BPEL 2.0. In *ICSSEA*.
[15] D. Cer *et al.* 2018. Universal Sentence Encoder. *CoRR* abs/1803.11175 (2018). arXiv:1803.11175 http://arxiv.org/abs/1803.11175
[16] W. X. Zhao *et al.* 2023. A Survey of Large Language Models. *CoRR* abs/2303.18223 (2023). https://doi.org/10.48550/ARXIV.2303.18223 arXiv:2303.18223
[17] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. Merayo. 2021. Wodel-Test: A model-based framework for language-independent mutation testing. *Softw. Syst. Model.* 20, 3 (2021), 767–793. https://doi.org/10.1007/s10270-020-00827-0
[18] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. 2018. A tool for domain-independent model mutation. *Sci. Comput. Program.* 163 (2018), 85–92.
[19] L. Gonzalez-Hernandez et al. 2018. Using Mutant Stubbornness to Create Minimal and Prioritized Test Sets. In *QRS*. IEEE, 446–457.
[20] E. Guerra, J. Sánchez Cuadrado, and J. de Lara. 2019. Towards Effective Mutation Testing for ATL. In *MODELS*. IEEE, 78–88.
[21] X. Han et al. 2023. Democratizing Chatbot Debugging: A Computational Framework for Evaluating and Explaining Inappropriate Chatbot Responses. In *CUI*. ACM, Article 39, 7 pages.
[22] R. M. Hierons and M. G. Merayo. 2009. Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.* 82, 11 (2009), 1804–1818.
[23] Y. Jia and M. Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *TAICPART*. 94–98. https://doi.org/10.1109/TAIC-PART.2008.18
[24] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.
[25] M. Kintis et al. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Trans. Software Eng.* 44, 4 (2018), 308–333.
[26] M. Kintis and N. Malevris. 2015. MEDIC: A static analysis framework for equivalent mutant identification. *Inf. Softw. Technol.* 68 (2015), 1–17.
[27] Lex. last access in 2024. https://aws.amazon.com/en/lex/.
[28] Z. Liu, Y. Feng, and Z. Chen. 2021. DialTest: Automated Testing for Recurrent-Neural-Network-Driven Dialogue Systems. In *ISSTA*. ACM, 115–126.
[29] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Sof. Eng.* 40, 1 (2014), 23–42.
[30] Microsoft Bot Framework. last access in 2024. https://dev.botframework.com/.
[31] OpenAI. last access in 2024. https://openai.com/.
[32] S. Pérez-Soler, E. Guerra, and J. de Lara. 2020. Model-Driven Chatbot Development. In *ER (LNCS, Vol. 12400)*. Springer, 207–222.
[33] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, and J. de Lara. 2021. Choosing a Chatbot Development Tool. *IEEE Softw.* 38, 4 (2021), 94–103.
[34] Rasa. last access in 2024. https://rasa.com/.
[35] Rasa test. last access in 2024. https://rasa.com/docs/rasa/testing-your-assistant.
[36] R. Ren, J. W. Castro, S. T. Acuña, and J. de Lara. 2019. Evaluation Techniques for Chatbot Usability: A Systematic Mapping Study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.
[37] G. R. S. Silva, G. N. Rodrigues, and E. D. Canedo. 2023. A Modeling Strategy for the Verification of Context-Oriented Chatbot Conversational Flows via Model Checking. *J. Univers. Comput. Sci.* 29, 7 (2023), 805–835.
[38] F. Tambon, F. Khomh, and I. Antoniol. 2023. A probabilistic framework for mutation testing in deep neural networks. *Inf. Softw. Technol.* 155 (2023), 107129.
[39] E. Viganò, O. Cornejo, F. Pastore, and L. Briand. 2023. Data-Driven Mutation Analysis for Cyber-Physical Systems. *IEEE Tr. Sof. Eng.* 49, 4 (2023), 2182–2201.
[40] Watson. last access in 2024. https://www.ibm.com/cloud/watson-assistant/.
[41] C. Wei, X. Yao, D. Gong, and H. Liu. 2021. Spectral clustering based mutant reduction for mutation testing. *Inf. Softw. Technol.* 132 (2021), 106502.
[42] J. D. Zamfirescu-Pereira et al. 2023. Herding AI Cats: Lessons from Designing a Chatbot by Prompting GPT-3. In *DIS*. ACM, 2206–2220.