# Coverage-based Strategies for the Automated Synthesis of Test Scenarios for Conversational Agents

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

Daniel Ávila
Universidad Autónoma de Madrid
Madrid, Spain
rdavilao@outlook.com

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

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

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

## ABSTRACT

Conversational agents – or chatbots – are increasingly used as the user interface to many software services. While open-domain chatbots like ChatGPT excel in their ability to chat about any topic, task-oriented conversational agents are designed to perform goal-oriented tasks (e.g., booking or shopping) guided by a dialogue-based user interaction, which is explicitly designed. Like any kind of software system, task-oriented conversational agents need to be properly tested to ensure their quality. For this purpose, some tools permit defining and executing conversation test cases. However, there are currently no established means to assess the coverage of the design of a task-oriented agent by a test suite, or mechanisms to automate quality test case generation ensuring the agent coverage.

To attack this problem, we propose test coverage criteria for task-oriented conversational agents, and define coverage-based strategies to synthesise test scenarios, some oriented to test case reduction. We provide an implementation of the criteria and the strategies that is independent of the agent development platform. Finally, we report on their evaluation on open-source Dialogflow and Rasa agents, and a comparison against a state-of-the-art testing tool. The experiment shows benefits in terms of test generation correctness, increased coverage and reduced testing time.

## CCS CONCEPTS

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

## KEYWORDS

Testing, Test suite generation, Task-oriented conversational agents

*AST 2024, April 2024, Lisbon, Portugal*
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

**ACM Reference Format:**
Pablo C. Cañizares, Daniel Ávila, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2024. Coverage-based Strategies for the Automated Synthesis of Test Scenarios for Conversational Agents. In *Proceedings of 5th International Conference on Automation of Software Test (AST 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Conversational agents are experiencing a fast evolution and adoption in our society. Nowadays, it is common to find conversational agents that provide a natural language user interface to assist in all sorts of online services. One of the reasons for this widespread adoption is their successful integration into daily life services such as education, banking, shopping and restaurants [1, 12], as well as their availability in engaging environments such as social networks (e.g., Telegram, X/Twitter, Facebook Chat), smart speakers (e.g., Amazon Alexa, Google Assistant, Siri) and websites accessible from different devices (e.g., tablets, smartphones).

Depending on their scope, conversational agents can be categorised into *open-domain* or *task-oriented*. Open-domain agents – like ChatGPT[1] or Google Bard[2] – build on large language models (LLMs) to provide a realistic and fluid conversation on arbitrary topics. However, given their generative approach, they may suffer from *hallucinations* [2], which are false or misleading responses to a given question. Currently, this makes it difficult to adopt them for conducting critical tasks or specific use cases [24, 35].

In contrast, task-oriented agents focus on specific tasks, such as booking tickets, ordering food, or making medical appointments, among many other possibilities. These agents are not prone to artificial hallucinations because both the conversations they are able to react to and the agents' responses are carefully designed for the task at hand (i.e., they are not based on generative artificial intelligence). There are many technologies on the market for designing and implementing task-oriented conversational agents, such as Google's Dialogflow[3], Microsoft's Bot Framework[4], Amazon Lex[5] or Rasa[6].

In this paper, our focus is on task-oriented conversational agents. They are software systems, and therefore, their proper behaviour must be ensured. In this respect, testing is a widely accepted technique for assessing the correct behaviour of systems [3, 31, 33]. Specifically for conversational agents, some approaches and tools –

[1] https://openai.com/chatgpt
[2] https://bard.google.com/chat
[3] https://dialogflow.com/
[4] https://dev.botframework.com/
[5] https://aws.amazon.com/en/lex/
[6] https://rasa.com/

like Botium[7] – enable the specification of test scenarios (expected sequences of user utterances and agent responses) that can be automatically executed. However, such approaches still face the following challenges: (i) the manual specification of test scenarios is costly, since agents may support hundreds or thousands of different conversation paths; (ii) assessing the quality of an agent test suite is an open question, as there is currently no reinterpretation in the context of conversational agents of accepted criteria for measuring and monitoring the testing activity, like test coverage metrics; (iii) testing a conversational agent is extremely time-consuming, so test reduction techniques ensuring a certain test coverage would be most helpful. Indeed, test suite reduction is one of the open challenges of testing conversational agents [9].

To fill this gap, in this paper, we present a classification of coverage criteria for task-oriented conversational agents, attending to factors like conversation design (conversation entry/exit points, conversation paths, conversational steps) and vocabulary. These criteria enable the measurement of agent test suites. Moreover, we propose test suite generation strategies with different levels of exhaustivity, which ensure the defined coverage criteria. Some strategies apply reduction techniques principles to alleviate the high cost associated with the test suite execution.

We have implemented our proposal on the basis of a technology-neutral design language for conversational agents, called Conga [26]. This way, the coverage metrics and test suite generation can be performed on agents developed with different technologies. To analyse the suitability of our coverage-based strategies, we have applied them to generate test suites for the Botium testing platform, for six open-source agents created with Dialogflow and Rasa. Our evaluation shows that the generated test suites are suitable (i.e., correct) and more complete than Botium's test generation facilities, and that reduction techniques help decreasing the testing time.

*Paper organisation.* Section 2 provides background on the architecture of task-oriented conversational agents, and on testing with Botium. Next, Section 3 analyses the state of the art. Section 4 details our coverage metrics and the strategies for the synthesis of test scenarios based on them. Section 5 describes our tooling, and Section 6 reports the results of the experimental study. Finally, Section 7 presents conclusions and prospects for future work.

## 2 BACKGROUND

This section provides background on task-oriented conversational agents (Section 2.1) and their testing using Botium (Section 2.2).

### 2.1 Task-oriented conversational agents

Task-oriented conversational agents (hereafter referred to as *agents*, in short) are targeted to solve particular tasks or to interact with specific software services, via conversation in natural language.

Fig. 1 outlines the elements that an agent specification comprises, using an agent for a pizzeria as an illustration. As depicted in the figure, agents define a collection of user *intents* that the agent aims at recognising. Each intent declares a set of *training phrases*, which exemplify how users could express the intent. This way, when the user states an utterance in natural language to the agent, this latter matches the most likely intent for the utterance with a certain

[7] https://www.botium.ai/

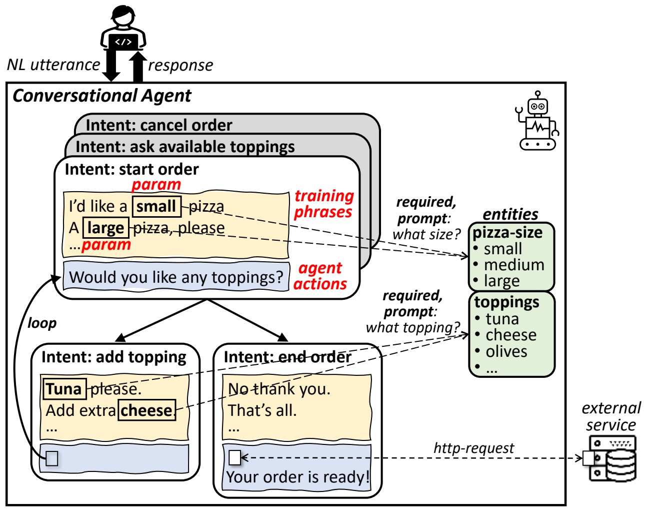


**Figure 1: Schema of task-oriented agent specification.**

probability. If no intent matches above a threshold, the agent applies a default *fallback intent* (if defined) to ask the user to rephrase the utterance. In the figure, the agent declares 5 intents, none of which are fallback. The figure also shows 2 training phrases for the intent *start order*: *"I'd like a small pizza"*, and *"A large pizza, please"*.

When the agent matches an intent, it may need to extract information from the user utterance. This way, intents may declare *parameters*, with training phrases hinting to that information. For instance, intent *start order* requires one parameter (pizza size), and phrases like *"I'd like a small pizza"* exemplify the provision of such information. Parameters are typed by *entities*, which can be predefined (e.g., dates) or agent-specific (e.g., pizza size). Moreover, parameters can be optional or required. If an utterance misses the value of a required intent parameter, then the agent asks the user for its value using the specified prompt. For example, upon the user utterance *"I'd like a pizza"*, the agent will prompt the question *"What size?"*, since the pizza size is a required parameter.

Chatbots also define the *actions* that the agent must perform when the intent is matched. A typical action is accessing an external service via a webhook to manage the user intent. This is what intent *end order* in the lower-right part of Fig. 1 does: it accesses the pizzeria information system to store the order. The last action associated to an intent is usually a response to the user, often textual, but which can contain other items such as images and links.

Finally, full-fledged conversations are designed by interleaving expected intents and agent actions, in user and agent turns. Each entry point to a possible conversation is called a conversation *flow*. Our pizzeria agent has 3 flows, which correspond to the conversations starting in intents *start order*, *ask available toppings*, and *cancel order*. In addition, some intents may be matched only after another one (depicted as arrows between intents in the figure). This enables designing conversations that may bifurcate in alternative *paths*. For example, upon matching the intent *start order*, the user is prompted for toppings, and depending on the user answer, either *add topping* or *end order* will follow. Hence, the flow starting in *start order* can follow 2 possible paths. Conversations can also include *loops*, as the one starting in the intent *add topping*, used to add several toppings iteratively.

## 2.2 Testing with Botium

Botium is a quality assurance platform to test conversational agents. It supports automated language understanding testing, conversation flow testing, security testing, performance testing and monitoring. For this paper, we are interested in its support for conversation flow testing via the definition and execution of *test scenarios*.

In Botium, a test scenario defines a conversation path that the agent is supposed to follow. Listing 1 shows a simple test scenario for the pizzeria agent, with the expected response of the agent (lines 3–4) upon a particular user utterance (lines 1–2). The test execution will send to the agent the specified user utterance, and the agent's actual response will be compared with the one indicated in the test scenario. If the agent does not respond as expected, the test case (i.e., the scenario plus the particular test utterances used) is considered as failed.

```
1 #me
2 I'd like to order a medium pizza
3 #bot
4 Would you like any toppings?
```

**Listing 1: Basic test scenario in Botium.**

Testing an intent typically requires many test utterances. For compactness of specification, Botium permits separating the structure of the conversation from the testing phrases issued by the users or the agent in different files, called *convo* and *utterance* files, respectively. This facilitates the specification of multiple test cases on the basis of the same scenario. As an example, Listing 2 shows a convo with the conversation in Listing 1, but replacing the phrase in line 2 by a reference to the utterance file *ORDER_PIZZA* in Listing 3. Referencing an utterance file in a #me-section sends every phrase in the file to the agent and checks the response. Referencing an utterance file in a #bot-section admits any phrase in the file as a valid agent response. In this example, the convo will be executed thrice, once for each phrase in Listing 3 (i.e., three test cases).

```
1 #me
2 ORDER_PIZZA
3 #bot
4 Would you like any toppings?
```

**Listing 2: Convo file.**

```
1 ORDER_PIZZA
2 I'd like to order a medium pizza
3 I'd like a small pizza
4 A large pizza, please
```

**Listing 3: Utterance file.**

In addition to assert the expected agent text answers (e.g., line 4 in Listing 2), Botium supports other types of assertions, like the expected agent intent, or the inclusion of links or images in responses.

Botium allows injecting dynamicity in test cases by means of its *scripting memory*. This enables using prebuilt system functions (e.g., to obtain the current date or the value of a system variable), as well as pushing variables to memory so that their value can be used in convos. The name of the variables must be preceded by '$', and their value can be either extracted from the agent responses, defined statically within the convo, or read from files. Listing 5 shows an example of the latter files. It defines three variables in line 1 (*$size*, *$kind*, *$price*) and three value assignments in lines 2–4. The convo in Listing 4 uses those variables. This way, the convo will be executed thrice, each time replacing the variables by one of the defined assignments.

```
1 #me
2 How much is a $size $kind pizza?
3 #bot
4 It is $price euros.
```

**Listing 4: Convo file.**

```
1       |$size  |$kind      |$price
2 Case1 |small  |Margherita | 15
3 Case2 |large  |Hawaiian   | 30
4 Case3 |large  |Pepperoni  | 32
```

**Listing 5: Variables file.**

Finally, the Botium Crawler[8] permits generating test cases from the agents in an automated way. For this purpose, it explores the quick responses and the lists of options offered by the agent, following all paths down until reaching either the end of the conversation or a user-defined maximum conversation depth. The identified conversation paths are saved as test cases and utterance lists. In Section 6.2, we will empirically study the suitability and coverage of the agent definition by this test case generation facility, comparing the quality of the test suites it generates with the coverage-based strategies we propose in Section 4.2.

## 3 STATE OF THE ART

As Cabot at el. discuss [9], there are many open challenges for testing task-oriented conversational agents, since current approaches only cover a small set of chatbot testing aspects. Next, we discuss testing proposals for conversational agents, focussing on intents (Section 3.1), conversations (Section 3.2) and usability (Section 3.3). Then, we review works on code coverage (Section 3.4).

## 3.1 Intent-level testing

Similar to unit testing, a first level of testing for conversational agents is to ensure that individual intents are properly defined.

Most commercial platforms, like Dialogflow, offer a console on which individual intents can be manually tested in isolation. However, the challenge is generating and selecting sensible test data. To check the accuracy of the intent recognition, some approaches generate input test utterances that simulate possible user phrases [4, 6, 8, 19, 34]. For example, Charm [8] provides mutation operators for input test phrases emulating common spelling mistakes; rephrasing utterances using synonyms; generating new phrases by translating existing phrases back and forth into an intermediate language; or changing numbers to words and vice versa. These operators can be applied to the training phrases of the intents to generate test utterances. In a similar vein, Božić and Wotawa [4, 6] rely on an ontology to formalise the mutation of input phrases using metamorphic relations, and generate follow-up test cases (i.e., additional input test utterances) from an initial test suite. The system by Yalla and Sunill [34] creates new test phrases by replacing nouns with synonyms. Similarly, DialTest [19] applies synonym replacement, back translation, and word insertion. Moreover, it calculates a coefficient to select the test cases most likely to find defects in an agent.

Unfortunately, some of these utterance transformers may generate unnatural test phrases (i.e., unlikely in real conversations) or change the semantics of the source phrase. For this reason, methods like AEON [17] have been proposed to evaluate the semantic similarity and language naturalness of the input test cases.

[8] https://botium-docs.readthedocs.io/en/latest/04_usage/index.html#botium-crawler

## 3.2 Conversation testing

Intent-level testing alone is not enough to ensure a proper agent behaviour, but other aspects such as the conversation flow need to be tested. As Section 2.2 explained, Botium supports these tests. Botium is independent on the underlying agent technology, providing connectors to several platforms[9]. Instead, BotTester[10] is a JavaScript framework built atop Mocha[11] and Chai[12] to test agents created with Bot Builder[13] (a *Node.js* framework for building bots).

To automate the generation of test scenarios, Božić et al. [5] use an AI planning approach. Specifically, they convert the agents into a first-order logic problem by using the Planning Domain Definition Language (PDDL) [15], where the goals are to fill in the parameters of the intents. Then, specific plans are generated that fulfil the goals, which correspond to different conversation paths. The method is specific to Dialogflow agents, and requires their manual translation into PDDL, which is costly and error prone. Moreover, agents can exhibit a vast number of conversation paths, each corresponding to a specific plan, which can lead to an explosion of test cases. To overcome this problem, we propose coverage criteria and several strategies to reduce the number of test cases.

Instead of testing, Silva et al. [30] use model checking on the conversation design, to ensure properties of interest in the conversation model. This has the advantage that the verification can be done in the design phase, but still requires from testing to ensure a proper handling of the user utterances in natural language.

## 3.3 Usability testing

The success of an agent depends on its usability [28]. Focused on this aspect, *chatbottest*[14] defines guidelines for identifying chatbot design issues in categories like answering, error management, intelligence, navigation, personality and understanding. *Chateval* [29], evaluates the quality of the agent responses using human judgement (via Amazon's Mechanical Turk[15]) against a baseline. Instead, Han et al. [16] propose an automated framework that identifies inappropriate agent responses and explain the causes. Følstad and Taylor [13] propose a system that analyses the agent-user interactions to understand response relevance and dialogue helpfulness, identify interaction patterns, and improve the agent design. Also to evaluate usability, BotTester [32] simulates users interacting with the agent, and collects interaction metrics like answer frequency, response time, or precision of intent recognition.

## 3.4 Code coverage

Code coverage measures the degree to which a test suite executes the source code of a program. It is valuable in software engineering and testing for maximising the analysed code, and consequently reducing the number of potential errors in a system. In industry, several safety standards apply code coverage to ensure the correctness and reliability of the software under study. They require to achieve a 100% structural coverage on key aspects such as entry points, statements and branches. Some of the most important ones include DO-178/ED-12 for safety-critical avionic software [18], EN 50128 for railways systems [14] and ISO 26262 for road vehicles [25].

[9] https://botium-docs.readthedocs.io/en/latest/06_connectors/01_index.html
[10] https://github.com/microsoftly/BotTester [11] https://mochajs.org/
[12] https://www.chaijs.com/ [13] https://www.npmjs.com/package/botbuilder
[14] https://chatbottest.com/ [15] https://www.mturk.com/

However, building test cases covering all possible execution paths requires a huge effort. To illustrate the magnitude of this challenge, consider the Facebook and Google testing environments. The former was reinforced on Sapienz [21], a tool for generating and executing test cases in Facebook One World Platform, where 100000 commits per week are made. The latter executes over 150 million of tests cases using the Test Automation Platform [22]. The high computational cost of these testing processes makes it necessary to maximise the quality of the selected test cases, discarding those that obtain duplicate results, and prioritising fault detection ability.

Altogether, we find different testing approaches for agents to assess their intents, conversation, and usability. Some of them automate the generation of new test utterances, but the test scenarios typically need to be defined manually, which is costly and error prone. Furthermore, it is not easy to know which parts of an agent have been tested, or if testing is sufficient. Hence, based on the wide acceptance that code coverage metrics have for general software, we propose their reformulation for conversational agents. Our aim is to help control and monitor the agent testing process, reducing the number of errors. To our knowledge, this is the first proposal of its kind. Moreover, to reduce the effort of creating test scenarios, we propose their automated synthesis with several strategies ensuring varying levels of coverage exhaustivity, including full coverage.

## 4 SYNTHESIS OF TEST SCENARIOS BASED ON COVERAGE CRITERIA

In the following, Section 4.1 presents a set of coverage criteria on agent designs to evaluate the strength and quality of a test suite. Then, Section 4.2 proposes several strategies for the synthesis of test scenarios based on the defined coverage metrics.

## 4.1 Coverage criteria for agent designs

Coverage is a widely accepted notion for code testing, with several levels like function, statement, edge, branch or condition. Our goal is to adapt this concept to the domain of conversational agents, so that it is possible to synthesise test scenarios focused on some criteria, and evaluate existing test suites against them.

Fig. 2 summarises our coverage criteria, which target the conversation and vocabulary of agents as follows:

- **Flow coverage:** This is the percentage of conversation entry points (*flows*) that have been exercised.
- **Path coverage:** This is the percentage of conversation exit points (*paths*) that have been tested.
- **Loop coverage:** This is the percentage of conversation loops exercised 0 times, exactly once, and more than once.
- **Intent coverage:** This is the percentage of intents that have been tested in at least one scenario.
- **Parameter coverage:** This is the percentage of parameters of all intents that have been tested with utterances that assign them a value, and utterances that do not assign them a value. Moreover, if the parameter is required, the prompt asking the user to give a value to the parameter must also have been tested.
- **Entity coverage:** This is the percentage of entities that have been used in at least one test.
- **Literal coverage:** This is the percentage of literals of each entity that have been used in at least one test.

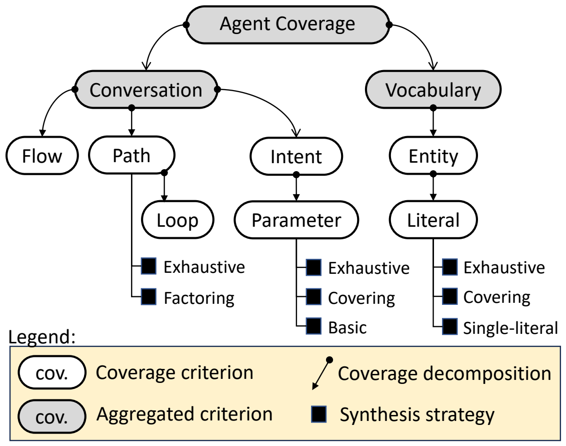


**Figure 2: Coverage criteria for intent-based agents.**

## 4.2 Coverage-based synthesis of test scenarios

To reduce the manual effort to specify test cases, we propose strategies to generate test scenarios guided by our coverage criteria. The strategies assume the existence of user utterance test sets $UTS_n$ for each intent $i_n$ of the agent. Given an agent, the strategies create a test scenario for each possible conversation path, and attempt to reduce the number of user utterances tested at each step of the conversation path, while ensuring a certain level of coverage of the paths, parameters or literals of the agent design.

*4.2.1 Path coverage strategies.* In the *exhaustive* strategy, full path coverage is obtained by using all test user utterances in every step of the conversation paths. Alternatively, the *factoring* strategy decreases the testing load by using only one test utterance in those conversation steps tested in a previous scenario execution.

*Example.* Fig. 3 illustrates both strategies for an agent with two conversation paths, given by intents *i1–i2* and *i1–i3*. Both strategies create a test scenario for each path. However, the *exhaustive* strategy (a) uses the complete user utterance test sets of all intents ($UTS_1$, $UTS_2$, $UTS_3$) in both scenarios. This accounts to $3^2+3^2=18$ test case executions (9 of each scenario, assuming 3 test utterances in each set). Instead, the *factoring* strategy (b) does not use the whole utterance test set $UTS_1$ in scenario 2, since this set was already used when testing intent $i1$ in scenario 1. Thus, it is enough to take one utterance $utter_{1k} \in UTS_1$ to place the agent execution focus on intent $i1$, and then use the utterance test set $UTS_3$. This reduces the number of test case executions to $3^2+3=12$.

*4.2.2 Parameter coverage strategies.* In the *exhaustive* strategy, full parameter coverage is obtained by using as many test user utterances as necessary to consider all combinations of existing/non-existing values for the parameters of each intent in each conversation step. Inspired by combinatorial testing [23], we propose two other strategies. The *basic* strategy only uses utterances that give value to all the parameters of each intent. This is a quick testing strategy that however does not provide full coverage, as it does not test for the absence of parameter values, but only for their presence. As a middle ground, the *covering* strategy iteratively tests the

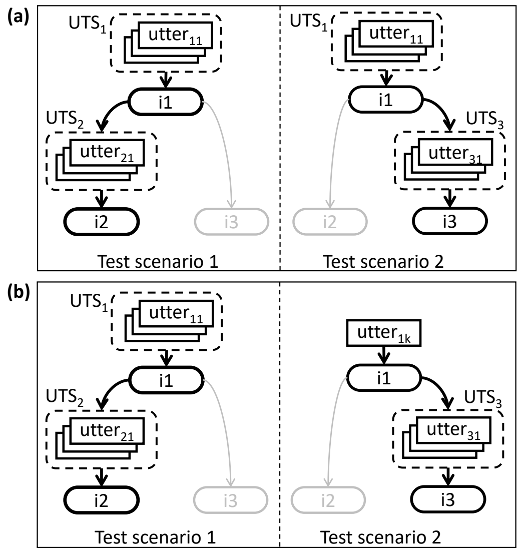


**Figure 3: Test utterance selection strategies for path coverage. (a) Exhaustive. (b) Factoring.**

conversations by providing test utterances that exercise all combinations of existing/non-existing parameter values for one intent, and using utterances giving values to all parameters for the other intents. This strategy yields full parameter coverage as well.

*Example.* Fig. 4 illustrates the strategies for testing a conversation path given by intents *i1–i2*, where each intent has two parameters. The *exhaustive* strategy (a) partitions the utterance test set $UTS_1$ in four disjoint subsets, each containing the utterances of one of the combinations of assigning or not a value to each parameter of intent *i1*. For instance, the utterances in subset $UTS_{1,10}$ give value to parameter *p1* but not to *p2*, the utterances in subset $UTS_{1,11}$ give value to both parameters, and so on. A similar strategy is used for intent *i2*. Then, the scenario is executed for each combination of existing/non-existing values for parameters *p1*, *p2*, *p3* and *p4*. If each subset had exactly one utterance, this would imply $4^2=16$ test case executions. If the parameters were mandatory, then specific scenarios would be needed to test that the agent asks for their value.

Fig. 4(b) shows the *basic* strategy, which only uses utterances that give a value to each parameter. In this example, the strategy would perform 1 test case execution.

Fig. 4(c) illustrates the *covering* strategy, which needs to employ two test scenarios. In the first one, intent *i1* is tested with the four partitions of $UTS_1$ used in strategy (a), and *i2* is tested with the utterance set $UTS_{2,11}$ of strategy (b) that gives value to all its parameters. Conversely, the second scenario uses $UTS_{1,11}$ of strategy (b) to test *i1*, and the four partitions of $UTS_2$ used in strategy (a) to test *i2*. In total, the *covering* strategy would perform 4+4=8 test case executions (4 of each scenario), resulting in half the executions of the *exhaustive* strategy, even if both provide full parameter coverage.

*4.2.3 Literal coverage strategies.* In the *exhaustive* strategy, full literal coverage can be obtained by using, for each intent with parameters, test user utterances that jointly employ every possible literal of the entities typing those parameters. Instead, the *single-literal* strategy reduces the number of test cases by selecting a

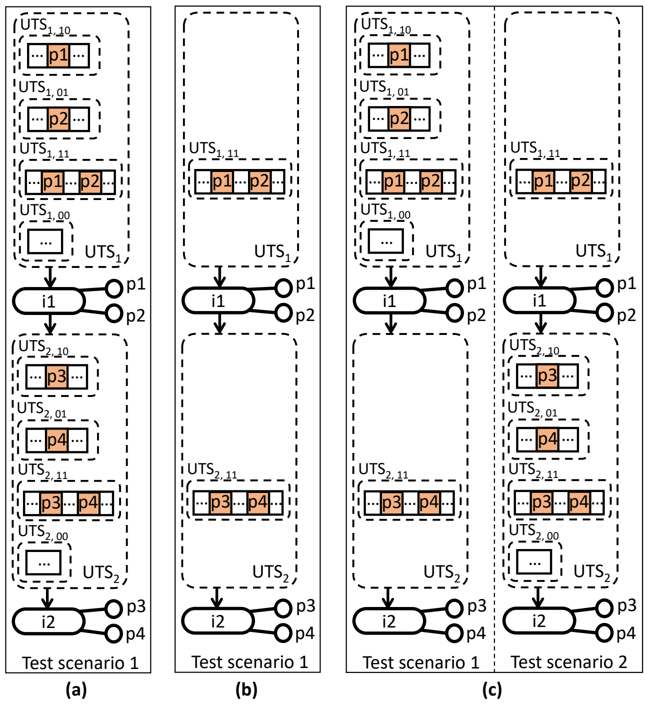


Figure 4: Test utterance partition strategies for parameter coverage. (a) Exhaustive. (b) Basic. (c) Covering.

single test utterance for each entity (i.e., with one literal). If several intents have parameters typed by the same entity, then the utterances to test each intent would employ different literals, to increase coverage. In any case, this strategy does not usually yield a high literal coverage. In between these two strategies, the *covering* one uses test utterances for every possible literal of each entity, but divided equally among the intents using the entity, thus achieving 100% literal coverage collectively. If an entity is used from just one intent, then the *exhaustive* and *covering* strategies are equivalent. Interestingly, the three strategies achieve 100% entity coverage.

*Example.* Fig. 5 exemplifies the strategies for testing a conversation path given by intents *i1*–*i2*, both having a parameter typed by the same entity *e1*, which in turn declares three literals (*l1*, *l2*, and *l3*). The *exhaustive* strategy (a) uses, in each intent, test user utterances for the three literals, achieving a 100% literal coverage. The *single-literal* strategy (b) selects test utterances for one of the literals (e.g., *l1*) in intent *i1*, and utterances for another literal (e.g., *l2*) in intent *i2*, achieving a coverage of 66.67%. Finally, the *covering* strategy divides the utterance partitions between the two intents, so that *i1* is tested with the utterances using literal *l1*, and *i2* is tested with those using *l2* or *l3*, achieving 100% literal coverage. As for the number of test case executions, if there were just one test utterance for each literal, then the *exhaustive* strategy would require 9 test case runs, the *single-literal* 1, and the *covering* 2.

## 5 TOOL SUPPORT

We have implemented the proposed coverage criteria and test generation strategies presented in the previous section. The implementation builds on an existing neutral language for agent designs, and parsers from different agent development technologies into this neutral language, which makes it possible to apply the strategies to agents created with diverse technologies (currently, Dialogflow and Rasa agents). Next, we provide details of the tool architecture (Section 5.1), the neutral agent design language (Section 5.2), and the test generation capabilities (Section 5.3).

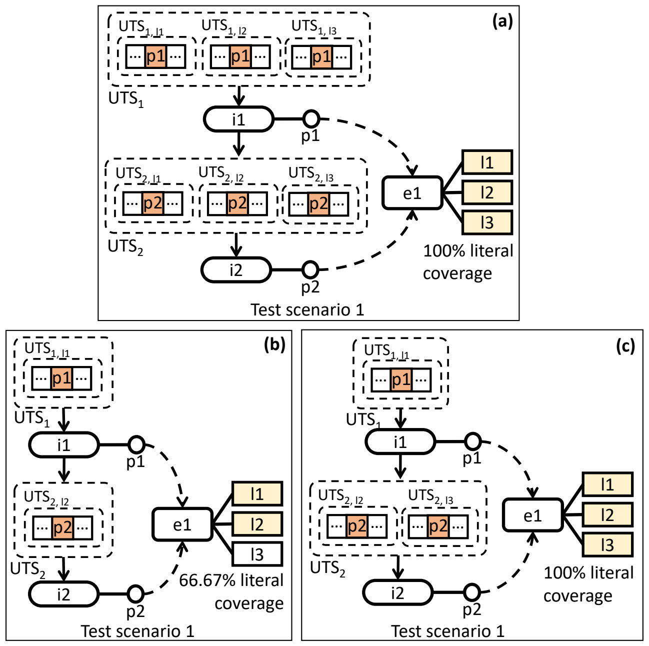


Figure 5: Test utterance partition strategies for literal coverage. (a) Exhaustive. (b) Single-literal. (c) Covering.

### 5.1 Architecture

Fig. 6 shows the architecture of our tooling. It has been developed as part of Asymob [10, 20], which is a platform to compute static metrics on conversational agents, and cluster sets of agents by their conversation topic. Asymob is available on GitHub as open source[16].

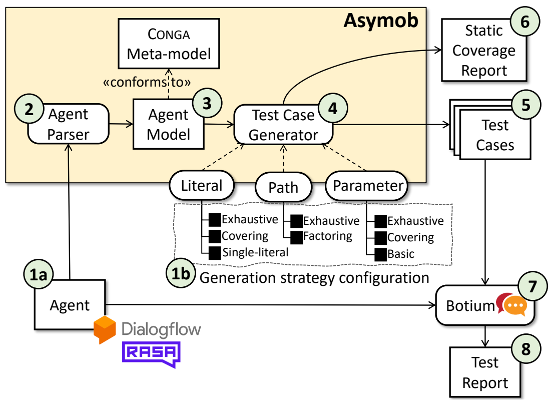


Figure 6: Architecture of our tool.

[16] https://github.com/PabloCCanizares/asymob

In the figure, we have numbered the steps required to generate a test suite based on our coverage-based strategies. First, the user must provide an agent (label 1a) and select the generation strategies among those described in Section 4.2 for paths, parameters and literals (label 1b). While our implementation is agnostic of the agent technology, currently the tool admits Dialogflow and Rasa agents. Then, the platform parses the received agent (label 2) and converts it into a Conga model (label 3). Conga is a neutral language for designing agents, which we will introduce in Section 5.2. Next, the test case generator (label 4) receives the agent model and the selected strategy as input, and generates a set of Botium test cases (convo, utterance and scripting memory files, label 5). The generator also performs a static analysis of the agent model, and produces a coverage report that includes the percentage of flows, paths, intents, parameters, entities and literals exercised in the tests (label 6). Finally, the generated test cases are executed with Botium (label 7), which provides a report of the test results (label 8).

## 5.2 A neutral language for agent designs

To make our test generation strategies independent from the technology and conversation design of the particular agent, our proposal is built on the basis of a neutral agent design language called Conga [26, 27]. This language encompasses the common features of the most widely used conversational platforms. This way, once our tool has converted the given agent into Conga (cf. labels 1–3 in Fig. 6), it is possible to handle the agent as a white box, facilitating the exploration of its internal structure and its static analysis.

Following model-driven engineering principles [7], the abstract syntax of Conga is defined by a meta-model. Fig. 7 shows a simplified excerpt, by which an agent design model (class Chatbot) contains intents (class Intent), which may be fallback. Intents have training phrases (class TrainingPhrase) and parameters (class Parameter), and these latter can be typed by predefined or user-defined entities (class Entity). The conversation flows (class Flow) are made of user-bot interactions (classes UserInteraction and BotInteraction), where bot interactions may define actions, like sending a text message (class Text). Conga supports multi-language agents (e.g., English and Spanish), for which the enumerate Language and the intermediate classes LanguageIntent, EntityLanguage or TextLanguage are needed. More details about Conga can be found at [26, 27].

## 5.3 Synthesis of test scenarios

From the agent design captured as a Conga model, our tool synthesises test scenarios for Botium according to the selected coverage-based strategy (cf. labels 4–5 in Fig. 6).

To achieve this, the tool first performs a static analysis of the agent design to extract the information needed to synthesise the test scenarios, such as the conversation paths, the intents, their training phrases and parameters, the agent responses, and the entities with their literals. Subsequently, the test case generation proceeds as follows. Each conversation path is converted into a *convo* file, which specifies a sequence of user utterances and agent responses (see Listing 1 in Section 2.2). Depending on the generation strategy, the set of training phrases of each intent is converted into one or several *utterance* files, which are used from the *convo* files when necessary (see Listings 2 and 3 in Section 2.2). In particular, selecting

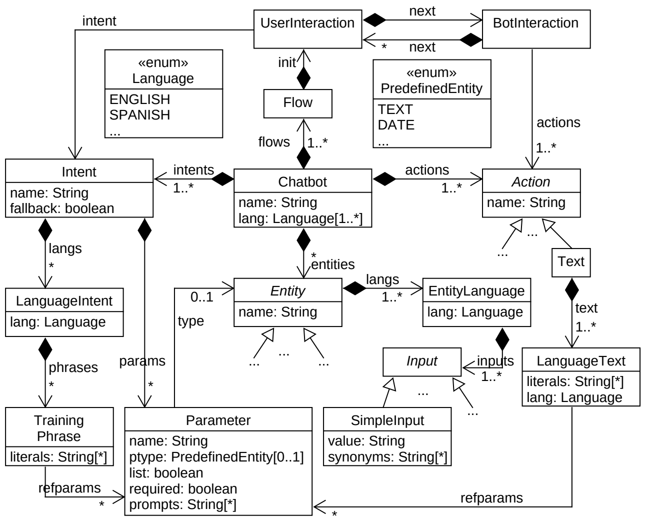


**Figure 7: Simplified meta-model excerpt of Conga.**

a parameter coverage strategy will partition the training phrases into several *utterance* files based on the parameters included in the phrases, while selecting a literal coverage strategy will partition them based on the literals they use (cf. Section 4.2). The responses provided by the agent are converted into *action* files, used from the *convo* files as oracles. In some particular cases (e.g., Rasa agents that control the actions using Python instead of declaratively) the oracle is set to the expected intent instead of a specific action. Finally, each entity is converted into a *scripting memory* file that contains all the entity literals (see Listings 4 and 5 in Section 2.2).

## 6 EVALUATION

In order to evaluate our proposal, we have performed an experiment involving agents of different technologies. The evaluation aims at answering the following research questions (RQs):

**RQ1** How do Asymob's test generation strategies compare with industrial ones in terms of correctness rate?
**RQ2** How complete are the test suites produced by our coverage-based strategies?
**RQ3** Can the reduction techniques decrease the computational testing cost?

Next, Section 6.1 describes the experimental setup, Sections 6.2–6.4 address the RQs, and Section 6.5 provides a discussion on the experiment results and the potential threats to validity. The experiment dataset and its results are available at https://github.com/PabloCCanizares/asymob/tree/master/experiments/coverage.

## 6.1 Experimental setting

For the experiment, we selected 6 agents from GitHub, built with Dialogflow and Rasa, and created by third-parties.

Table 1 shows some design metrics for the selected agents, calculated with Asymob. The first column contains the agent name, and the subsequent columns show the number of conversation entry points (FLOW), conversation paths (PATH), total paths (TPATH),

| Name | FLOW | PATH | TPATH | INT | ENT | PAR | LIT |
|---|---|---|---|---|---|---|---|
| **Dialogflow** | | | | | | | |
| BikeShopAgent | 4 | 4 | 6 | 5 | 1 | 3 | 15 |
| beta | 15 | 15 | 16 | 17 | 5 | 11 | 77 |
| Insurance_Bot | 6 | 12 | 17 | 19 | 0 | 6 | 0 |
| **Rasa** | | | | | | | |
| 256644 | 10 | 10 | 10 | 11 | 0 | 0 | 0 |
| Email-WhatsApp | 3 | 6 | 6 | 8 | 0 | 0 | 0 |
| Episode8 | 4 | 5 | 7 | 10 | 1 | 4 | 5 |

**Table 1: Features of the selected agents.**

intents (INT), user-defined entities (ENT), parameters (PAR), and literals (LIT). The number of total paths is the number of conversation paths, plus the additional paths that arise when a user does not provide a value for a mandatory parameter, so the agent prompts for it. For example, an agent that defines a single intent with two mandatory parameters (*p1* and *p2*) has one conversation path (PATH=1) and 4 total paths (TPATH=4). The latter comprise the 4 possible conversation paths depending on whether a value is provided or not for *p1* and *p2* (i.e., 4 combinations).

As the selection criterion, we tried to maximise the heterogeneity of the agents. That is, we selected agents with linear conversations (PATH=FLOW), conversations that bifurcate (PATH>FLOW), conversation paths that depend or not on the provision of parameter values (TPATH>PATH and TPATH=PATH), with and without parameters, and with different design (INT) and vocabulary size (ENT, LIT).

Then, we generated test scenarios for the 6 agents using the Botium Crawler introduced in Section 2.2, and Asymob with all strategies. As test utterances, we took the training phrases of each intent. The following sections answer the RQs by analysing the generated tests and their execution on the 6 agents.

## 6.2 RQ1: Suitability of test suites

To answer this RQ, we compare the correctness of the test scenarios generated with Botium, and those generated with Asymob.

Table 2 summarises the scenarios generated by Botium. The first two columns show the platform and the agent name. The next four columns provide information related to the quality of the test suite: its size in terms of the number of test cases (TCs), the number of successful (Pass) and failed (Fail) test cases, and the execution time of the whole suite (Time), in seconds. The next two columns show the number of generated *convo* and *scripting memory* files. Finally, the last six columns capture the coverage of flows, (total) paths, intents, entities, parameters and literals by the generated test suite.

Table 3 shows the same information for the test suites generated using the *exhaustive* strategy of Asymob.

*Answering RQ1.* In Table 2, only agents 256644 and Email-WhatsApp pass all the tests generated by Botium. However, all agents should satisfy these tests, since the test utterances are the training phrases of the agents' intents. That is, all the test cases must pass, so a failed test case reveals a problem in the test case design. Instead, as Table 3 shows, all the agents pass the test suites generated by Asymob using the *exhaustive* strategy. They also pass the test suites synthesised for our other strategies (omitted due to space limitations). Hence, we conclude that our test generation strategies achieve a higher correctness rate than Botium's for the synthesis of test scenarios.

## 6.3 RQ2: Completeness of test suites

To address RQ2, we have measured the coverage on conversation and vocabulary aspects by the different strategies, as well as the testing artefacts produced by the strategies.

First, we take Botium as a baseline (Table 2). We observe that the size of the test suites ranges from 18 to 423 test cases, encoded in 3 to 164 convos, and no scripting memory file. In terms of coverage, the test suites generated by Botium cover 100% of some elements like INT and ENT. However, the achieved coverage is not complete for other aspects such as PATH in each of the considered agents, LIT in the beta agent, and FLOW in the Rasa agents.

With respect to our *exhaustive* strategy (Table 3), the generated test suites are generally bigger[17], ranging from 107 to 16847 test cases. This increase in size is due to an in-depth exploration of all conversation paths, achieving a 100% coverage in all FLOW, PATH, INT, PAR, ENT and LIT in both Dialogflow and Rasa agents. The test suites were encoded in 6 to 17 convos, and 0 to 5 scripting memory files. Interestingly, our strategy generated fewer convos than Botium. This is so as our strategy produces convos with longer conversations that test many intents, while Botium-generated convos typically consist of a single user-bot interaction. In addition, our technique is able to generate scripting memory files with suitable values for parameters.

Table 4 shows the effect of our test case generation strategies relative to the *exhaustive* one. In the *basic* strategy, the test suites are generally smaller than those created by the *exhaustive* strategy, with reductions from 0 to 3150 test cases. Moreover, this strategy achieved a 100% coverage in FLOW, INT, ENT, PAR and LIT. However, this strategy compromised the full coverage of PATH, reducing it in a range from 12.5% to 50% in beta, Episode8, Insurance_Bot and BikeShopAgent.

The *covering* strategy produces the same test suites as the *exhaustive* strategy in all cases. Moreover, the literal *covering* strategy was not applicable to any of the agents.

The *factoring* strategy generates test suites very similar to those of the *exhaustive* strategy when it is applied over linear agents (e.g., BikeShopAgent, beta). However, when applied over agents with complex conversations, it reduces the size of the test suite (41 test cases less in Insurance_Bot, 707 test cases less in Email-WhatsApp, and 6065 test cases less in Episode8) while maintaining the coverage levels of the *exhaustive* strategy. Hence, this strategy achieves a good compromise between coverage and reduction, being suitable for the synthesis of test cases for agents where PATH > FLOW.

Finally, in the *single-literal* strategy, the size of the test suites of the agents that contain literals in their design (LIT > 0) becomes significantly reduced. This is reflected in the agents BikeShopAgent, Episode8 and beta, whose test suites are reduced in 602, 5096 and 16485 test cases, respectively. The size of the test suites of the other agents remains equal to the *exhaustive* strategy. The coverage of the agents' elements is maintained except in the case of LIT, which decreases significantly.

*Answering RQ2.* The *exhaustive* strategy achieves 100% coverage in all conversation and vocabulary criteria. Anyhow, reaching 100% coverage on entities and literals depends on the availability of suitable test utterances. Since we used the training phrases as test

[17] The number of test cases is equal for the case of Insurance_Bot.

| Platform | Name | TCs | Pass | Fail | Time (s) | #Convos | #Scripts | FLOW | PATH | INT | ENT | PAR | LIT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dialogflow | BikeShopAgent | 18 | 10 | 8 | 17.3 | 3 | 0 | 100 | 50 | 100 | 100 | 100 | 100 |
| | beta | 362 | 352 | 10 | 382.6 | 14 | 0 | 100 | 87.5 | 100 | 100 | 100 | 45.16 |
| | Insurance_Bot | 107 | 34 | 73 | 110.9 | 11 | 0 | 100 | 64.7 | 100 | - | 100 | - |
| Rasa | 256644 | 73 | 73 | 0 | 3.1 | 21 | 0 | 90 | 90 | 100 | - | - | - |
| | Email-WhatsApp | 48 | 48 | 0 | 3 | 8 | 0 | 33.3 | 33.3 | 100 | - | - | - |
| | Episode8 | 423 | 235 | 188 | 10529.2 | 164 | 0 | 40 | 28.5 | 100 | 100 | 100 | 100 |

**Table 2: Summary of the test suites generated with Botium.**

| Platform | Name | TCs | Pass | Fail | Time (s) | #Convos | #Scripts | FLOW | PATH | INT | ENT | PAR | LIT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dialogflow | BikeShopAgent | 620 | 620 | 0 | 1318 | 6 | 3 | 100 | 100 | 100 | 100 | 100 | 100 |
| | beta | 16847 | 16847 | 0 | 19323 | 16 | 5 | 100 | 100 | 100 | 100 | 100 | 100 |
| | Insurance_Bot | 107 | 107 | 0 | 345 | 17 | 0 | 100 | 100 | 100 | - | 100 | - |
| Rasa | 256644 | 107 | 107 | 0 | 5.3 | 10 | 0 | 100 | 100 | 100 | - | - | - |
| | Email-WhatsApp | 836 | 836 | 0 | 168 | 6 | 0 | 100 | 100 | 100 | - | - | - |
| | Episode8 | 6400 | 6400 | 0 | 2770.9 | 7 | 2 | 100 | 100 | 100 | 100 | 100 | 100 |

**Table 3: Summary of the test suites generated with the *exhaustive* strategy.**

| | Basic | | Covering | | Factoring | | Single-literal | |
|---|---|---|---|---|---|---|---|---|
| Name | TCs | Coverage | TCs | Coverage | TCs | Coverage | TCs | Coverage |
| BikeShopAgent | -440 | -50% PATH | 0 | – | 0 | – | -602 | -80% LIT |
| beta | -12 | -12.5% PATH | 0 | – | 0 | – | -16485 | -94%LIT |
| Insurance_Bot | -45 | -48% PATH | 0 | – | -41 | – | 0 | – |
| 256644 | 0 | – | 0 | – | 0 | – | 0 | – |
| Email-WhatsApp | 0 | – | 0 | – | -707 | – | 0 | – |
| Episode8 | -3150 | -29% PATH | 0 | – | -6065 | – | -5096 | -75% LIT |

**Table 4: Effect of the reduction strategies on the generated test suites w.r.t. the *exhaustive* strategy.**

utterances, 100% was achieved. Our reduction strategies decrease the amount of test cases generated, which may have a toll on coverage. In comparison with our baseline Botium, we obtain higher coverage, and produce scripting memory files, which helps to better structure the test specifications. Remarkably, the PATH coverage of our *basic* strategy is similar to the one of Botium for Dialogflow.

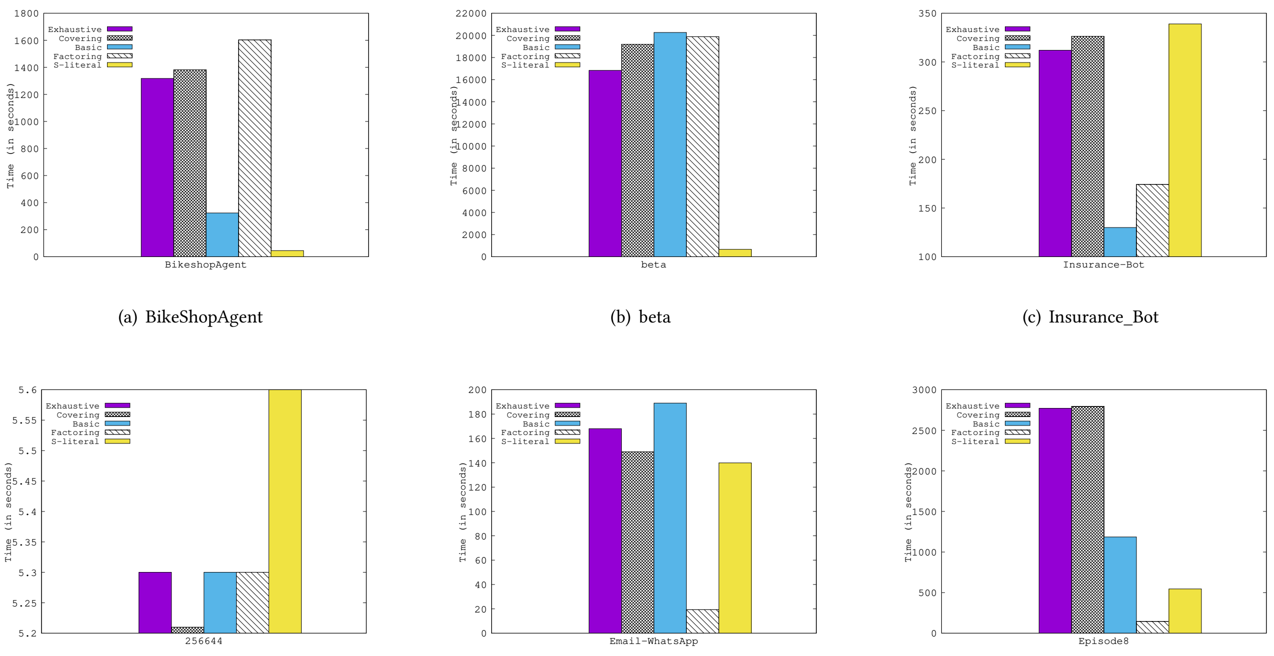
## 6.4 RQ3: Computational cost reduction

Achieving full coverage may have a high computational cost, hence we analyse the test execution time of each proposed reduction strategy. Fig. 8 displays the execution times of the test suites generated by the *exhaustive*, *basic*, *covering*, *factoring* and *single-literal* strategies. The $x$-axis represents the names of the agents under study while the $y$-axis shows the total time in seconds.

To measure the effectiveness of the reduction strategies, we use the execution times of the *exhaustive* approach as a baseline. The strategy has required 1318 seconds for BikeShopAgent (Fig. 8(a)), 16835 seconds for beta (Fig. 8(b)), 312 seconds for Insurance_Bot (Fig. 8(c)), 5.3 seconds for 256644 (Fig. 8(d)), 168 seconds for Email-WhatsApp (Fig. 8(e)), and 2770.9 seconds for Episode8 (Fig. 8(f)).

In general, we observe that, in all cases, at least one reduction technique performs better than the *exhaustive* strategy. The *basic* strategy outperforms the *exhaustive* strategy in three agents (BikeShopAgent, Insurance_Bot and Episode8) requiring 323, 133.3 and 1185.7 seconds, respectively. This entails a reduction of 75.4%, 57.37%, and 57.2% in execution time. The *covering* strategy beats the *exhaustive* one in case of the 256644 and Email-WhatsApp agents, completing the test in 5.2 and 149 seconds, resulting in an 2% and 11.3% of reduction. The *factoring* approach is faster than the *exhaustive* strategy in the Insurance_Bot, Email-WhatsApp and Episode8 agents, completing in 174.2, 19.4 and 144.2 seconds and reducing the execution time by 44.1%, 88.4% and 94.8%. Finally, the *single-literal* strategy achieves faster executions compared to the *exhaustive* strategy in four agents (BikeShopAgent, beta, Email-WhatsApp, Episode8) requiring 45.3, 665.5, 139.9 and 545.7 seconds, respectively. This corresponds to reductions of 96.5%, 96.1%, 16.7% and 80.2% in total time. Please note that, in some cases, the reduction strategies show slightly higher execution times than the *exhaustive* one. We argue that this is a spurious effect of the execution, since the reduction strategies never produce more test cases than the *exhaustive* one.

*Answering RQ3.* Comparing all strategies, the *single-literal* has obtained the best results in two agents (BikeShopAgent and beta), the *factoring* in two agents (Email-WhatsApp and Episode8), the *basic* in one agent (Insurance_Bot) and the *covering* in one agent (256644). In conclusion, reduction strategies effectively decrease computational costs compared to the *exhaustive* strategy, allowing us to respond RQ3 affirmatively.



(a) BikeShopAgent (b) beta (c) Insurance_Bot

(d) 256644 (e) Email-WhatsApp (f) Episode8

**Figure 8: Computational time required to execute the proposed reduction techniques.**

## 6.5 Discussion and threats to validity

The experiment has shown three main points. First, that the use of an explicit agent design (as extracted by Asymob from Dialogflow and Rasa agents, and represented in CONGA) is beneficial when generating test suites, in comparison to a black-box test generation technique as realised by Botium. This is so as the availability of the agent design enables a white-box analysis that can generate more complete test suites, using more sophisticated elements (e.g., scripting memory files). Second, our coverage criteria can be used to quantify the completeness of the test cases. We have seen that our *exhaustive* strategy achieves higher completeness than that of Botium Crawler. Finally, we have seen that by using reduction techniques, we can decrease the test execution times, controlling the level of completeness of the test suite.

Regarding threats to the validity of the experiment, the reduced size of our dataset (6 agents) may affect the generalisability of the results (external validity). This way, stronger results may be achieved with a larger sample size, which we plan to conduct in future work. Regarding construct validity, we used coverage as a measure of the completeness of the test suite. This could be used as a proxy for its quality, but its ability to detect errors, e.g., as given by mutation analysis [11], can be used here. We plan to perform such a study in future work. Finally, we found some spurious effects when measuring execution times of the reduction strategies against the *exhaustive* one. Even if all chatbots were deterministic w.r.t. intent recognition, these could have been mitigated by repeating the testing more times and calculating the average. The long execution times in some cases (>6 hours) prevented us from performing this procedure, which we will conduct in the future.

## 7 CONCLUSIONS AND FUTURE WORK

In this paper, we have characterised coverage criteria for testing task-oriented conversational agents, along with different reduction strategies for test scenario generation. We have implemented a test generation facility atop the Asymob tool, which is able to synthesise Botium test scenarios for agents created in Dialogflow and Rasa. Our evaluation shows that the generated test suites are suitable (i.e., correct), more complete than industrial test generation facilities, and reduction techniques can help decrease testing time.

In future work, we plan to use mutation analysis to evaluate the effectiveness of the different strategies, and we aim to explore new prioritisation and minimisation strategies based on artificial intelligence (AI). We plan to combine the presented techniques with test case augmentation techniques for analysing the understandability of the agents. For this, we aim at generating test utterances by incrementing the training phrases of the system, including variations that emulate syntactical errors committed by the users. Finally, we plan to adapt the proposed strategies for their application to conversational agents based on generative AI (e.g., GPT4, LlaMA 2), like those built with Langchain (https://www.langchain.com/).

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] Eleni Adamopoulou and Lefteris Moussiades. 2020. Chatbots: History, Technology, and Applications. *Machine Learning with Applications* 2 (2020), 100006.

[2] Hussam Alkaissi and Samy I McFarlane. 2023. Artificial Hallucinations in ChatGPT: Implications in Scientific Writing. *Curēus* 15, 2 (2023), 4 pages.

[3] Antonia Bertolino, Emilio Cruciani, Breno Miranda, and Roberto Verdecchia. 2022. Testing Non-Testable Programs Using Association Rules. In *3rd ACM/IEEE International Conference on Automation of Software Test.* ACM/IEEE, 87–91.

[4] Josip Bozic. 2022. Ontology-Based Metamorphic Testing for Chatbots. *Softw. Qual. J.* 30, 1 (2022), 227–251.

[5] Josip Bozic, Oliver A. Tazl, and Franz Wotawa. 2019. Chatbot Testing Using AI Planning. In *2019 IEEE International Conference on Artificial Intelligence Testing (AITest).* IEEE, 37–44.

[6] Josip Bozic and Franz Wotawa. 2019. Testing Chatbots Using Metamorphic Relations. In *31st IFIP WG 6.1 Int. Conf. on Testing Softw. and Syst. (LNCS, Vol. 11812).* Springer, 41–55.

[7] M. Brambilla, J. Cabot, and M. Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers.

[8] Sergio Bravo-Santos, Esther Guerra, and Juan de Lara. 2020. Testing Chatbots with Charm. In *13th Int. Conf. on Quality of Information and Communications Technology (CCIS, Vol. 1266).* Springer, 426–438.

[9] Jordi Cabot, Loli Burgueño, Robert Clarisó, Gwendal Daniel, Jorge Perianez-Pascual, and Roberto Rodríguez-Echeverría. 2021. Testing Challenges for NLP-intensive Bots. In *3rd IEEE/ACM International Workshop on Bots in Software Engineering (BotSE@ICSE).* IEEE, 31–34.

[10] P. C. Cañizares, J. M. López-Morales, S. Pérez-Soler, E. Guerra, and J. de Lara. 2023. Measuring and clustering heterogeneous chatbot designs. *ACM Trans. Softw. Eng. Methodol.* (2023), 42 pages. https://doi.org/10.1145/3637228

[11] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41.

[12] Dana Doherty and Kevin Curran. 2019. Chatbots for Online Banking Services. In *Web Intelligence*, Vol. 17. IOS Press, 327–342.

[13] Asbjorn Folstad and Cameron Taylor. 2021. Investigating the User Experience of Customer Service Chatbot Interaction: A Framework for Qualitative Analysis of Chatbot Dialogues. *Quality and User Experience* 6 (2021), 1–17.

[14] European Committee for Electrotechnical Standardization. 2001. *EN 50128: Railway Applications-Communication, Signalling and Processing Systems-Software for Railway Control and Protection Systems.* Standard.

[15] Malik Ghallab, Adele Howe, Craig A. Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL – The Planning Domain Definition Language.* Technical Report CVC TR-98-003/DCS TR-1165. Yale Center for Computational Vision and Control.

[16] Xu Han, Michelle Zhou, Yichen Wang, Wenxi Chen, and Tom Yeh. 2023. Democratizing Chatbot Debugging: A Computational Framework for Evaluating and Explaining Inappropriate Chatbot Responses. In *5th International Conference on Conversational User Interfaces.* ACM, New York, NY, USA, Article 39, 7 pages.

[17] Jen-tse Huang, Jianping Zhang, Wenxuan Wang, Pinjia He, Yuxin Su, and Michael R. Lyu. 2022. AEON: A Method for Automatic Evaluation of NLP Test Cases. In *31th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, New York, NY, USA, 202–214.

[18] Leslie A. Johnson. 1998. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. *CrossTalk* 199 (1998), 11–20.

[19] Zixi Liu, Yang Feng, and Zhenyu Chen. 2021. DialTest: Automated Testing for Recurrent-Neural-Network-Driven Dialogue Systems. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, 115–126.

[20] Jose María López-Morales, Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2022. Asymob: A Platform for Measuring and Clustering Chatbots. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE Companion).* ACM/IEEE, 16–20.

[21] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *25th International Symposium on Software Testing and Analysis.* ACM, 94–105. See also: https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/.

[22] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* IEEE, 233–242.

[23] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2, Article 11 (feb 2011), 29 pages.

[24] OpenAI. (last accessed in 2023). https://openai.com/research/gpt-4#limitations.

[25] Rob Palin, David Ward, Ibrahim Habli, and Roger Rivett. 2011. ISO 26262 Safety Cases: Compliance and Assurance. In *6th IET International Conference on System Safety 2011.* IET, 1–6.

[26] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2020. Model-Driven Chatbot Development. In *39th International Conference on Conceptual Modeling (ER) (LNCS, Vol. 12400).* Springer, 207–222.

[27] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2021. Creating and Migrating Chatbots with Conga. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE Companion).* IEEE, 37–40.

[28] Ranci Ren, John W. Castro, Silvia Teresita Acuña, and Juan de Lara. 2019. Evaluation Techniques for Chatbot Usability: A Systematic Mapping Study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.

[29] João Sedoc, Daphne Ippolito, Arun Kirubarajan, Jai Thirani, Lyle H. Ungar, and Chris Callison-Burch. 2019. Chateval: A Tool for Chatbot Evaluation. In *2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations).* Association for Computational Linguistics, 60–65.

[30] Geovana Ramos Sousa Silva, Genaína Nunes Rodrigues, and Edna Dias Canedo. 2023. A Modeling Strategy for the Verification of Context-Oriented Chatbot Conversational Flows via Model Checking. *Journal of Universal Computer Science* 29, 7 (2023), 805–835.

[31] Richard J. Somers, James A. Douthwaite, David J. Wagg, Neil Walkinshaw, and Robert M. Hierons. 2023. Digital-Twin-Based Testing for Cyber-Physical Systems: A Systematic Literature Review. *Inf. Softw. Technol.* 156 (2023), 107145.

[32] Marisa Vasconcelos, Heloisa Candello, Claudio Pinhanez, and Thiago dos Santos. 2017. Bottester: Testing Conversational Systems with Simulated Users. In *XVI Brazilian Symposium on Human Factors in Computing Systems* (Joinville, Brazil). ACM, New York, NY, USA, Article 73, 4 pages.

[33] Franz Wotawa, Lorenz Klampfl, and Ledio Jahaj. 2021. A Framework for the Automation of Testing Computer Vision Systems. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST).* IEEE, 121–124.

[34] Muralidhar Yalla and Asha Sunil. 2020. AI-Driven Conversational Bot Test Automation Using Industry Specific Data Cartridges. In *IEEE/ACM 1st International Conference on Automation of Software Test (AST).* ACM, New York, NY, USA, 105–107.

[35] J. D. Zamfirescu-Pereira, Heather Wei, Amy Xiao, Kitty Gu, Grace Jung, Matthew G. Lee, Bjoern Hartmann, and Qian Yang. 2023. Herding AI Cats: Lessons from Designing a Chatbot by Prompting GPT-3. In *2023 ACM Designing Interactive Systems Conference (DIS).* ACM, 2206–2220.

Received 24 January 2024