2019 IEEE International Conference on Artificial Intelligence Testing (AITest)

# Chatbot Testing Using AI Planning

Josip Bozic, Oliver A. Tazl and Franz Wotawa
*Institute of Software Technology*
*Graz University of Technology*
A-8010 Graz, Austria
{jbozic,oliver.tazl,wotawa}@ist.tugraz.at

***Abstract*—Chatbots, i.e., systems that can interact with humans in a more appropriate way using natural language, have been of increasing importance. This is due the fact of the availability of computational means for natural language interaction between computers and humans that are becoming closer to the interaction between humans alone. Consequently, there are more and more chatbots available that are intended to support humans organizing tasks or making decisions. In this paper, we focus on how to verify the communication capabilities provided by chatbots. In particular, we introduce an automated approach for generating communication sequences and carrying them out. The approach is based on AI planning where each action can be assumed to be a certain question that is given to the chatbot. The answer of the chatbot should make the action post-condition true, in order to proceed with the plan. In cases of deviations between the actual chatbot behavior and the expected one, re-planning is required. Besides the approach, we discuss its application to the domain of tourism and outline a case study.**

***Index Terms*—Planning, automated testing, chatbot testing**

## I. Introduction

The use of virtual assistants, or chatbots [1], as bases for system interactions with humans is becoming of growing interests. This is due to the fact that human languages offer a natural way of communication and the availability of methods and tools allowing computers to analyze and process such languages. There are studies, e.g. [2], [3], predicting a rise of the chatbot market in the future. Natural language interfaces (NLI) offer a lot of new possibilities for humans to interact and collaborate with chatbots [4]. Chatbots rely often on a set of rules or trained neural networks for deciding answers to be given to users or questions to be stated in order to handle requests. In principle chatbots are not restricted to an application domain. However, it is often the cases that chatbots rely on pre-specified patterns that trigger the chatbot's behavior (e.g. [5], [6]), which restricts the possibilities of interaction patterns with human users. This is certainly not a drawback for applications like chatbot-based systems especially designed for a particular purpose, e.g., for providing tourism information, planning trips, or booking hotels.

With increasing complexity, chatbots can be expected to support humans in decision making. Ensuring correct functionality becomes critical for such systems. The failure to fulfil this requirement can lead to negative consequences for individuals and companies alike. Therefore, testing the chatbot's functionality will become an important task in the future.

In this paper, we do not focus on the underlying techniques and methods behind successful chatbot implementation. Instead, we discuss the validation and verification part behind chatbots. In particular, we are interested in providing methods and tools for automating the test of chatbots in order to increase quality of the chatbot implementation. Testing chatbots requires to provide test cases and a test execution framework for checking functional and non-functional requirements. The former is for confirming that the chatbot's behavior follows given expectations. For example, when we want to book a hotel in Prague, a hotel-booking chatbot should be able to guide the user until a hotel in Prague can be booked. The latter deals with issues like security, which is important especially in the case of textual interfaces provided over the Internet. There chatbots might be vulnerable against classical attack patterns like cross-side scripting (XSS) or SQL injection.

Until now many approaches have been introduced for testing of specific types of systems. Automated methods offer the advantage of generating and executing numerous tests in a short amount of time, thereby using test oracles. For example, model-based testing relies on models of the system under test (SUT) [7] or test inputs (e.g. [8]).

Another technique, namely automated planning and scheduling, or simply planning, is taken from artificial intelligence (AI). Here testing is depicted as a planning problem that uses first-order logic and planning algorithms (e.g. [9], [10]).

In this paper, we discuss how to automatize testing the functional requirements of chatbots. Here we build in part upon our previous works [11], [12] and introduce a planning-based automated testing framework for chatbots. The primary motivation is to ensure functionality of a chatbot with regard to correctness of data. For illustrating the approach, we make use of a tourism chatbot. We depict a use case of such a chatbot comprising booking a hotel room. Furthermore, we claim that a planning-based approach can be adapted for testing of non-functional properties as well.

The paper is organized as follows. First, Section II discusses the tested chatbot. Then, Section III gives an overview about planning and its adaptation to chatbot testing. The implemented testing approach is discussed and demonstrated on a use case in Section IV and Section V, respectively. Afterwards, Section VI gives an overview about related literature and the paper is concluded in Section VII.

## II. Hotel Booking Chatbot

The tested chatbot is meant for tourism purposes. As such, it offers the possibility to book a room by interacting with a user. The chatbot was developed in the known chatbot development tool, Dialogflow [13]. The typical program flow of Dialogflow, as shown in Figure 1, consists of several steps.

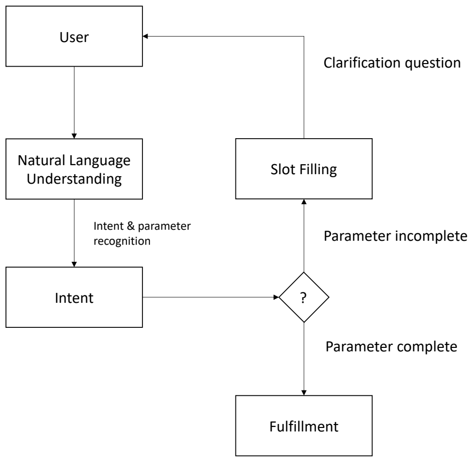

Fig. 1: Communication flow chart in Chatbots

First, the user provides an input to the system, which is then parsed by the agent, thereby responding accordingly. To specify the conversation flow, intents are used in order to classify requests. Each intent is composed of possible user inputs that can trigger this intent. It also defines what data to extract and in what manner to respond. Generally speaking, an intent represents a specific dialogue part within a conversation. In our case, the chatbot is able to recognize a booking request and respond appropriately.

Training phrases represent an important part of the intent matching of user inputs. These comprehend libraries of possible utterances, which the user could say to match the intent. Also, phrases can be annotated to match entities. These represent categories of things relevant for the intent, like city names or numbers. A typical chatbot consists of several intents. Each of them representing a variety of user intentions.

Every user request triggers the chatbot to find a corresponding match for a particular intent. Afterwards, the agent returns a response according to that intent. On the other hand, a fallback to match unrecognized intents can occur as well. In case that an intent is matched, the chatbot tries to complete the intent by filling the parameters needed for a reservation. To do so, the chatbot further asks clarification questions, which is called slot filling. The chatbot remains in this mode even if the user responds in an unexpected way. A detailed overview about this functionality is given in Section V.

In general, after the parameters of the intent are matched altogether, the chatbot executes the fulfillment. Eventually, this leads to a final response or an API call to finish the intended action.

From the technical point of view, the communication between a user and the chatbot usually proceeds textually over HTTP(S). HTTP messages, i.e. requests and responses, are exchanged between both peers. Here, an important element in the human-to-chatbot communication are keywords. A specific context in a communication is built from multiple connected keywords. A typical user inquiry can have the following form:

*`I want to book a room in Prague.`*

The keyword *Prague* represents a value for the parameter `$location` in Dialogflow. However, `$location` is a symbolic name that encompasses a wide range of values of the intern entity `@sys.location`. This means that a user can specify any value for a location in proper English language and the chatbot will understand it. In our example, every user inquiry is followed by a response from the server.

A hotel reservation consists of multiple information like name of the hotel, accommodation type, number of stars etc. The user can provide one or two information per inquiry, e.g.:

*`I want to book an apartment in Prague.`*

Here, the information for the parameter `$venue-type`, namely *apartment*, is given in addition to `$location`. In the tourism chatbot, eight parameter values are mandatory before a reservation can be completed:

```
$location, $venue-type, $venue-title,
$checkin-date, $adult, $child, $night,
$star
```

The submission of these parameter values represents a requirement for the chatbot. The only exception is the `abort` message, which terminates the communication. This functionality is implemented by intents in Dialogflow. The booking chatbot uses natural language to demand clarification for specific parameters during communication. On the other hand, the user communicates by combining natural language with keyword values or submits only the latter. A reservation is complete once all mandatory information is provided by the user.

In the next section, we discuss the testing representation for the implemented chatbot.

## III. Planning for Modeling and Testing

Planning has been initially used in robotics and intelligent agents [14], [15]. A planning specification provides an agent the possibility to react to specific conditions. This is defined as a planning problem and its solution presents a sequence of actions that leads from an initial to a goal state. The program that processes the specification, i.e. the planner, generates a plan according to a planning algorithm. For this purpose, we rely on an enhanced version of Fast Downward [16] that was introduced in [17].

The initial definition of planning, as given in [14], is adapted for this paper as follows:

**Definition 1.** *The quadruple $(P, I, G, A)$ defines the planning problem. $P$ denotes the set of predicates, $I$ is the initial and $G$ the goal state. $A$ represents the set of actions, where an* *action $a \in A$ uses a predicate $p \in P$ as a first-order logic formula. A state is given by one or more predicates that are valid in this state. An action $a \in A$ can be accessed by the functions pre(a) and post(a), respectively.*

An action is added to the plan only if its $pre(a)$ is true in a state $S$. The consequences of $post(a)$ will result in the transition to a new state $S'$, thus $S \xrightarrow{a} S'$. All predicates that are true in $post(a)$ will be valid in $S'$ as well.

**Definition 2.** *The solution for the planning problem $(P, I, G, A)$ represents a sequence of actions so that $I \xrightarrow{a_1} S_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} G$.*

The motivation behind using planning is the fact that every action or event in a program can be represented with certain pre- and postconditions. Here the initial state represents the starting conditions. From this idle state, the action's conditions change dynamically during planning. The goal state defines the termination point, i.e., the final condition where planning finishes. In our example, this represents a situation where an expected response from the chatbot is encountered. The plan itself is a set of actions that guide the execution from an initial to the goal state.

Once generated, the execution of the plan is deterministic. Since the goal of the proposed approach is to test the functionality of a chatbot, we specify such information as a planning problem. In this manner, the test generation itself becomes a planning problem.

## A. Planning Model

Model representations play an important role in testing. As already mentioned, state machines are often used in model-based testing in order to depict the SUT or an attack. In our example, however, we make use of planning for modeling purposes. The resulting planning model serves as the foundation for test generation. The specification is implemented in the Planning Domain Definition Language (PDDL) [18], which represents a planning language standard. The data definitions in the model are adapted to chatbot testing and are separated into two distinct files:

- Domain: Encompasses definitions that are used in every planning problem.
- Problem: The definitions that are used for one specific problem.

We will depict an excerpt from the planning specification for our booking chatbot. Here, the domain is defined in the following way:

```
(define (domain tdomain)
(:requirements :strips :typing :negative-
   preconditions)
(:types town accom date name stars adults
    children nights)
(:constants europa sacher imperial - name
    vienna prague ljubljana - town)
(:predicates
    (Place ?t)
    (Accommodation ?a)
    (Hotel ?h)
    (CheckIn ?w)
    (Adults ?d)
    (Ninos ?c)
    (Days ?n)
    (Rating ?r)
    (Reservation ?h ?a ?t ?w ?d ?c ?n ?r)
(:functions
    (Place ?t - town)
    (Accommodation ?a - accom))
(:action askPrague
    :parameters()
    :precondition (and (not (PlaceAsked))
       (not(HotelAsked))(not(NameAsked)))
    :effect (and (Place prague)
       (PlaceAsked)))
(:action askHotel
    :parameters()
    :precondition (and )
    :effect (and (Accommodation hotel)))
(:action askSacher
    :parameters()
    :precondition (and )
    :effect (and (Hotel sacher)))
(:action setAdultsOne
    :parameters()
    :precondition (and )
    :effect (and (Adults eins)))
(:action askRatingFive
    :parameters()
    :precondition (and (PlaceAsked)
       (HotelAsked) (NameAsked))
    :effect (and (Rating five)))
(:action combineFamily
    :parameters(?d - adults ?c - children
       )
    :precondition (or (and (Place prague)
       (Accommodation hotel)) (and
       (Ninos cero)))
    :effect (and (FamilyCombined)))
(:action response
    :parameters(?h - name ?a - accom ?t -
        town ?w - date ?d - adults ?c -
       children ?n - nights ?r - stars)
    :precondition (and (Hotel ?h)
       (Accommodation ?a) (Place ?t)
       (CheckIn ?w) (Adults ?d)(Ninos ?c)
       (Days ?n) (Rating ?r) (Reservation
        ?h ?a ?t ?w ?d ?c ?n ?r))
    :effect (and (Done))))
```

Domain description in PDDL

PDDL uses a type-object hierarchy that resembles classes from object-oriented programming. Predicates are used in further definitions and consist of zero or multiple parameters.

On the other hand, constants can be used in order to assign concrete, pre-specified values, to a parameter. Actions consist of lists of parameters, pre- and postconditions (or effects). In general, a combination of valid predicates during execution defines a specific state. It should be noted that an action does not necessarily contain preconditions, thus it can be added to a plan without restrictions. Transitions between states are realized through actions.

On the other hand, the shortened problem description for the domain looks as follows:

```
(define (problem tproblem)
(:domain tdomain)
(:objects
    t - town
    a - accom
    h - name
    w - date
    d - adults
    c - children
    n - nights
    r - stars)
(:init
    (Place nn)
    (Accommodation na)
    (Hotel none)
    (CheckIn never)
    (Adults null)
    (not (Ninos cero))
    (Rating zero)
    (Days nula)
    (not (Done))
    (Reservation sacher hotel prague
       intwodays eins cero dva five))
(:goal
    (and (Done))))
```

Problem description in PDDL

As already mentioned, the initial state defines a specific value assignment in predicates that define the starting point. Here, the planner begins the search for solutions. The initial predicate definitions resemble the initial data definition in the chatbot. The goal, on the other hand, describes a final state where planning terminates.

Once specified, both files are read by the planner and a plan is constructed. During planning, we assume that no external interruptions are encountered. A notification is given in case that the goal state cannot be reached. A few resulting plans from various configurations are given in Table I.

TABLE I: Generated plans for chatbot testing

| | |
|---|---|
| 1 | (askprague ),(askhotel ),(asksacher ), (askintwodays ),(askratingfive ), (setadultsone ),(setchildrenzero ), (setnightstwo ) |
| 2 | (askprague ),(askhotel ),(asksacher ), (askintwodays ),(askratingfive ), (asktown ljubljana),(setchildrenzero ), (combinefamily null dos),(setnightstwo), (setadultsone ) |
| 3 | (askratingfive ),(asksacher ), (setchildrenzero ),(combinefamily d c), (setadultsone ),(asktown prague), (setnightstwo ),(askintwodays ),(askhotel ) |
| 4 | (abort ), (askhotel ), (askintwodays ), (askprague ), (combinefamily d c), (askratingfive ),(asksacher ), (setadultsone ),(setchildrenzero ), (setnightstwo ),(asktown ljubljana) |
| 5 | (askprague ),(askhotel ),(abort ), (combinefamily drei c),(asksacher ), (askintwodays ),(askratingfive ), (asktown ljubljana),(setadultsone ), (setchildrenzero ),(setnightstwo ) |
| 6 | (askprague ),(askhotel ),(asksacher ), (askintwodays ),(setadultsone ), (setchildrenzero ),(asksacher ), (setnightstwo ),(askratingfive ) |

In this example, the planning specification allows to infer multiple plans in order to reach the given goal. The challenge behind our approach is to check whether a chatbot will function in a correct way even when encountered with unforeseen values and orders of actions. Therefore, the planner builds plans by adding repetitive actions, different values for the same parameter etc. Different values for an initial state will cause different plans to be generated.

However, it should be noted that a plan represents only an abstract test case. A test concretization is needed in order to carry out real tests against the SUT. The concretization mechanism will be explained in more detail in Section IV.

## *B. Planning for Chatbot Testing*

Planning offers the possibility to model different domains in PDDL. As already mentioned, we specify chatbot-related information as part of the planning model. The method of mapping of chatbot information to a planning specification is shown in Table II.

Following the data structure in Dialogflow, corresponding counterparts are specified in PDDL. For testing purposes, we depict only mandatory information for completing a hotel reservation. As already mentioned, these are expected to be submitted via user inquiries. Therefore, we define parameters in PDDL for all of Dialogflow's crucial booking parameters. In order to resemble entities, we apply concrete values to parameters by defining constants (see domain description in Section III-A). The actions represent instructions with specific pre- and postconditions, eventually relying on parameters. In planning, multiple parameters can be used in one action definition. In this way, we map all necessary user inquiries to planning actions.

A generated plan resembles a sequence of user inquiries. The question arises whether the chatbot will follow its implemented conversational pattern regardless of the submitted input. This will be addressed further in Section V.

TABLE II: Mapping of chatbot elements to PDDL

| Dialogflow | | PDDL | |
|---|---|---|---|
| **intent text** | **parameter** | **action** | **parameter** |
| I want to book a room in Prague | `$location` | `askPrague` | `?t - town` |
| book an apartment | `$venue-type` | `askApartment` | `?a - accom` |
| i prefer Sacher | `$venue-title` | `askSacher` | `?h - name` |
| I want to check-in today. | `$checkin-date` | `askToday` | `?w - date` |
| one adult | `$adult` | `setAdultsOne` | `?d - adults` |
| 2 children | `$child` | `setChildrenTwo` | `?c - children` |
| one night | `$night` | `setNightsOne` | `?n - nights` |
| i want 5 stars | `$star` | `askRatingFive` | `?r - stars` |

## IV. Approach Overview

In this section, we briefly outline our approach for functional testing of chatbots. The goal of our implementation is to emulate a tester for automated test execution. The planning adaptation results in the generation of several plans that serve as abstract test cases. The test execution framework, a Java implementation, automatically parses plans and generates concrete test cases at runtime.

The entire approach, which includes planning, is depicted in Figure 2. In this paper, our framework encompasses an interface with a Dialogflow chatbot. Therefore, we claim that our framework can be used for testing of other applications of Dialogflow as well. Also, since the communication relies on HTTP, the framework can be adapted for testing of other online chatbots. For testing of other types of chatbots, a corresponding interface needs to be programmed.

After the list of plans is generated, the framework reads the action sequences and their corresponding parameters. Inside the implementation, every action from PDDL has a corresponding Java method in the framework. During the execution, the reservation response from the chatbot, if encountered, is checked against a test oracle. This is done with the parser jsoup [19]. In this scenario, a test oracle represents a final chatbot response that confirms a hotel booking. The individual responses are checked whether they contain expected data for a specific reservation. Then, the next action is picked from the plan and a new request is generated.

The whole process continues as long as all actions from a plan are carried out or the execution gets stuck in the meantime. If the execution reaches the goal state, then we consider a test case to be successful. Otherwise, if there is a discrepancy between the plan and the real execution, then the plan is a failed test case. Although, this does not necessarily imply that the SUT's behaviour was incorrect. Afterwards, the next plan is selected and executed. When all plans have been carried out, a list of successful ones is displayed.

## V. Use Case

Since chatbots provide an interface between the user and other information systems, we included our hotel booking system as an actor into a UML use case diagram, as shown in Figure 3.

For this scenario, plans from different initial configurations are used. From the planning point of view, the use case is a sequence of actions that leads to an objective, i.e. booking a hotel room for the provided information. Usually, the communication between peers is triggered by sending an initial message, as demonstrated in Section III-B. By definition, the chatbot excepts at least one information per request. So, after every request the chatbot provides a response and demands additional information, e.g. the number of guests. It should be noted that the chatbot will insist on answering the particular question and ignore any information that it deems inappropriate. The only exception being an `abort` message that restarts the reservation process immediately. The interaction concludes when a reservation is finalized, thus sending the user a confirmation with all summarized data.

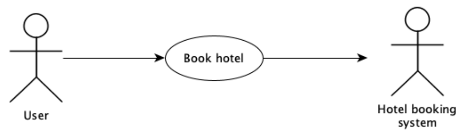

Fig. 3: Use case for a hotel booking chatbot

In this use case, our framework resembles a user who wants to make a reservation in the following manner.

```
I want to book a room in Prague.
- Do you want a hotel, hostel or
apartment?
I want a hotel.
- Which one do you prefer?
hotel sacher
- When do you want to start your stay?
tomorrow
- How many adults?
one
- How many children?
two children
- How many nights?
1 night
- How many stars do you prefer?
five
- Let me sum up! Location: Prague,
Name: Sacher Hotel, Check-In: 2018-12-21,
Accommodation: hotel, People: 1 adults,
2 children, Nights: 1, Stars: 5 stars
```

As already shown, Table II gives an overview about the parameter recognition in the chatbot. The chatbot's last response

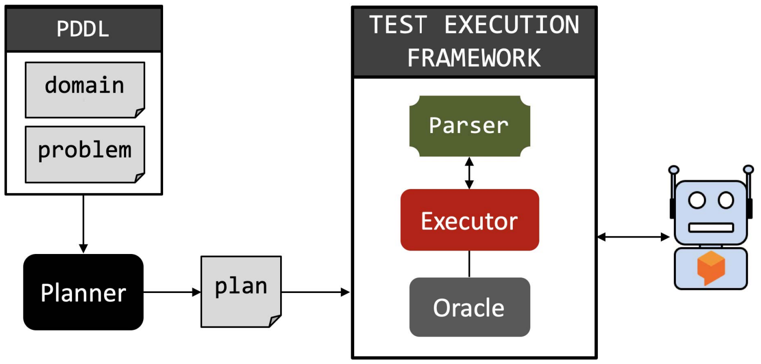

Fig. 2: Planning-based chatbot testing

also finalizes the session with the user. Afterwards, she or he can book another reservation.

There are cases where multiple information can be submitted as part of one request. For example, a user can provide the number of guests and number of children in one request:

```
- How many adults?
one adult and one child
- How many nights?
```

The chatbot should recognize the intention and not ask for this information in the future and switch to the next question. In general, the order of specific questions in the chatbot is pre-defined.

One interaction flow between user and chatbot is considered to be one test case. The user can interrupt and demand a restart of the process anytime by submitting a `abort` message. Table III depicts the concretization mechanism between abstract and concrete test cases. For example, the first plan from Table I represents an abstract test case. The execution framework parses the actions and generates a HTTP message. The corresponding parameters will be added to the HTTP request, together with some samples of pre-specified natural language. Then, this request is submitted against the SUT. The fully concretized plan represents a concrete test case. The chatbot replies after every message according to the request content (not depicted). In fact, the framework encompasses information for all possible abstract data from planning.

In this use case, we want to test whether the chatbot behaves correctly under different user behavior. For this reason, we generated plans that have diverse sequences of messages, including repetitive actions, different values for same parameters and abort requests. Every plan represents a test case after which the booking process is started from scratch, i.e. the initial state. It is also our intention to see if a sequence of events exists that can confuse the chatbot by bypassing the use of keywords so it gets a wrong impression about affiliated information. Here, the test execution does not necessarily follow the plan execution. The reason for this is the fact that many plans result in unexpected behavior for the chatbot. Basically, a plan depicts a set of challenges for the chatbot. This means that it does not necessarily imply correct behavior on side of the SUT. For example, the chatbot can keep insisting on a specific information although the user continues to submit other information (according to plan). However, the chatbot should behave correctly even under such circumstances.

Our goal state specification corresponds to the final state of the chatbot, that is, a reservation. According to that, actions and parameter values are picked so that the state's conditions can be met. In fact, the goal state resembles the postcondition of the server's final response. In PDDL, this is formalized as a conjunction of the following predicates:

```
(Hotel ?h)∧(Accommodation ?a)∧
(Place ?t)∧(CheckIn ?w)∧
(Adults ?d)∧(Ninos ?c)∧(Days ?n)∧
(Rating ?r)∧
(Reservation ?h ?a ?t ?w ?d ?c ?n ?r)
```

Here the planner searches for values that will satisfy the predicate:

```
(Reservation sacher hotel prague intwodays
eins cero dva five)
```

In order to demonstrate our approach, we generated three test sets, each containing 200 different plans, by the following means:

1) Providing only one information per request; use of repetitive actions with same values.
2) Providing multiple information per request; re-requests with different valid and unintended values.
3) Adding a `abort` message to the specification of (1) and (2).

In order to obtain distinct test sets, different domain definitions are implemented for each of the cases above. These PDDL specifications differ only to a small degree from each other. For example, minor changes are added to certain action conditions. Thus, the resulting planner output will be distinguishable as well. In every plan, the action arrangement differs as well as their number.

TABLE III: Concretization of test cases

(a) Abstract test case

| # | Action |
|---|---|
| 0: | (askprague ) |
| 1: | (askhotel ) |
| 2: | (asksacher ) |
| 3: | (askintwodays ) |
| 4: | (askratingfive ) |
| 5: | (setadultsone ) |
| 6: | (setchildrenzero ) |
| 7: | (setnightstwo ) |

→

(b) Concrete test case

| # | Message |
|---|---|
| 0: | ”I want to book a room in Prague” |
| 1: | ”hotel” |
| 2: | ”sacher” |
| 3: | ”in two days” |
| 4: | ”five stars” |
| 5: | ”one adults” |
| 6: | ”zero children” |
| 7: | ”two nights” |

In the first case, several different plans lead to the expected reservation. The result analysis indicates that the arrangement of actions does not affect the final result, besides two exceptions that are discussed below. Also, repetitive actions with same values do, as expected, not influence the final result. In fact, every update for an already set parameter will overwrite the former value with the new one. However, the reason for the high rejection was likely an implementation oversight: The information for hotel ratings does not require the specification of the *stars* keyword. During communication, the chatbot asks about the hotel rating at the end. However, since it accepts only numbers, in contrast to other information, it cannot distinguish if the user already submitted the rating before. For example, if the user submits *two children* during the last request, the chatbot will assume that the user asks for *two stars*. This is not the case for other information, e.g. number of adults, nights or children.

Also, the chatbot always excepts a statement about the town to visit to be the first request.

To summarize, the communication flow for the first case must follow the structure:

```
(askprague )
(askhotel )
(asksacher )
(askintwodays )
(setadultsone )
(setchildrenzero )
(setnightstwo )
(askratingfive )
```

As can be seen, the first and last request types must follow a specific order. All other requests can be sent at will, independently of the server's response.

Regarding the second case, submitting different data does affect the testing as well. For example, the user changing her or his mind and asking for a different town. As already mentioned, the chatbot takes the last information about one parameter as the one to consider. For this reason, if the initial version of the town corresponded to the expected one, another town request did disregard that value. Thus, it results in a failing test case. On the other hand, submitting multiple information at once did not influence the order (with the mentioned exceptions above).

Finally, the test set with aborting messages restarts the booking process. For this reason, none of these test cases were able to accomplish a reservation due to lack of information at the end. The exception is only the case when the restarting request is the first action in the plan.

In general, in many cases the obtained execution sequence differed from the plan. However, this does not downgrade the chatbot's functionality. Actually, the chatbot behaved according to its specification under all tested circumstances. Unfortunately, a false reservation can be confirmed with values that differ from the expected ones. This happened in cases where the chatbot was tricked by providing incomplete data, e.g. numerical values without further indication like children, stars etc. In these cases, the chatbot could not recognize the intent.

## VI. Related Work

Two major fields represent the bulk of this paper, namely planning and testing of natural language agents. Work that focuses on the first topic usually applies planning in robotics but, to a smaller degree, to testing of software as well. On the other hand, literature about testing of AI systems in general, and especially chatbots, is sparse.

A broad overview about planning and its implementations is given in [15] and [20]. The first works that used planning for test case generation usually by manipulating the planning specification [9], [10]. In such way, diverse sequences of actions would be generated. The authors from the latter work introduced PATHS, a planning system for GUI testing. The actions that are necessary for GUI testing, like events, are defined by the tester. In turn, the planner generates plans for test execution. In addition to that, PATHS encompasses a test oracle as well.

A planning approach that relies on a learning algorithm is elaborated in [21]. Here a model of the SUT is extracted, which in turn is used in order to infer a planning specification. The initial plan represents a weak solution that represents an actual test case. During execution, the state transitions are recorded and added to the specification. The next generation of plans will take the former execution traces and generate stronger solutions.

Other works that used planning for testing purposes in various manifestations include [22], [23] and [24].

Testing of chatbots lacks, to our knowledge, many scientific works.

The authors from [25] deal with issues during chatbot development with regard to several dimensions. They conduct analyses on several chatbots by targeting their functionality. Besides the conversational capabilities, external concerns like databases and API integration are addressed as well.

Tools that automatically test the functionality of chatbots include [26] and [27]. The latter tool, called Bottester, emulates the conversation between a user and chatbot in an automated manner. The authors test and evaluate the functionality of the system by exhaustively submitting inputs. The tool contains questions that are submitted to a specific chatbot. Also, it comprehends a list of expected outputs, i.e. a test oracle, for the individual inputs.

In [11] the authors of this paper introduced a security testing approach for chatbots. The implemented framework is meant for automated testing for vulnerabilities. Malicious inputs are submitted against a chatbot and test oracles are consulted during execution.

## VII. Conclusion and Future Work

In this paper, we addressed an upcoming issue in artificial intelligence, namely chatbots. These service-oriented implementations communicate in natural language and need to guarantee functionality. For this reason, we introduced a planning-based testing approach for functional testing of chatbots. Here planning is used to create a model, which serves for test case generation. The test purpose is to test a tourism chatbot by booking a hotel reservation. Planning generates sequences of actions that depict possible test scenarios. In order to refine the test generation, specific conditions can be set in planning so more diverse test cases are generated. Afterwards, a testing framework executes the test cases in an automated manner.

The first results have proven that the new approach is successful to test chatbots. The general conclusion is that the chatbot reacted according to its implementation. However, in some cases it could not recognize the intent of a request, thus making incorrect reservations. In order to cover this issue, the implementation leaks in the chatbot have to be addressed further.

We claim that the proposed planning-based approach can be used for other functional and non-functional testing, like security, as well. Basically, a system can be tested by identifying critical information and modeling it in terms of pre- and postconditions. The proposed model is easily extendable so new test data can be added to the specification.

In the future, we want to extend the testing approach by testing other SUTs for a more generalized approach.

## Acknowledgement

The research presented in the paper has been funded in part by the Cooperation Programme Interreg V-A Slovenia-Austria under the project AS-IT-IC (Austrian-Slovenian Intelligent Tourist Information Center).

## References

[1] M. L. Mauldin, "ChatterBots, TinyMuds and the Turing Test: Entering the Loebner Prize Competition," in *AAAI '94 Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, 1994, pp. 16–21.

[2] "Chatbot Market Size And Share Analysis, Industry Report, 2014 - 2025," https://www.grandviewresearch.com/industry-analysis/chatbot-market, accessed: 2018-05-07.

[3] "Gartner Top Strategic Predictions for 2018 and Beyond," https://www.gartner.com/smarterwithgartner/gartner-top-strategic-predictions-for-2018-and-beyond/, accessed: 2018-05-07.

[4] A. Følstad and P. B. Brandtzæg, "Chatbots and the new world of HCI," *interactions*, vol. 24, no. 4, pp. 38–42, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3085558

[5] R. S. Wallace, "The Elements of AIML Style," in *ALICE A.I. Foundation*, 2003.

[6] B. A. Shawar and E. Atwell, "Using corpora in machine-learning chatbot systems," in *International Journal of Corpus Linguistics, vol. 10*, 2005.

[7] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing," in *Department of Computer Science, The University of Waikato, New Zealand, Tech. Rep, 4*, 2006.

[8] A. Takanen, J. DeMott, and C. Miller, "Fuzzing for Software Security Testing and Quality Assurance," in *Artech House, Inc., Norwood, USA*, 2008.

[9] A. E. Howe, A. von Mayrhauser, and R. T. Mraz, "Test Case Generation as an AI Planning Problem," in *Automated Software Engineering, 4*, 1997, pp. 77–106.

[10] A. M. Memon, M. E. Pollack, and M. L. Soffa, "A Planning-based Approach to GUI Testing," in *Proceedings of the 13th International Software / Internet Quality Week (QW'00)*, 2000.

[11] J. Bozic and F. Wotawa, "Security Testing for Chatbots," in *Proceedings of the 30th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'18)*, October 2018.

[12] ——, "Planning-based Security Testing for Chatbots," in *Proceedings of the 21st International Multiconference Information Society - IS 2018, AS-IT-IC Workshop*, 2018.

[13] "Dialogflow," https://dialogflow.com/, accessed: 2018-12-11.

[14] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," in *Artificial Intelligence*, 1971, pp. 189–208.

[15] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentic Hall, 1995.

[16] M. Helmert, "The Fast Downward Planning System," in *Journal of Artificial Intelligence Research 26 (2006):191-246*, 2006.

[17] M. Katz, S. Sohrabi, O. Udrea, and D. Winterer, "A Novel Iterative Approach to Top-k Planning," in *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS'18)*, 2018.

[18] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL - The Planning Domain Definition Language," in *The AIPS-98 Planning Competition Comitee*, 1998.

[19] "jsoup: Java HTML Parser," https://jsoup.org/, accessed: 2018-02-02.

[20] M. Ghallab, D. Nau, and P. Traverso, "Automated Planning: Theory and Practice," in *Morgan Kaufmann*, 2004.

[21] A. Leitner and R. Bloem, "Automatic Testing through Planning," Technische Universität Graz, Institute for Software Technology, Tech. Rep., 2005.

[22] N. Ghosh and S. K. Ghosh, "An Intelligent Technique for Generating Minimal Attack Graph," in *Proceedings of the 1st Workshop on Intelligent Security (SecArt'09),*, 2009.

[23] J. Lucangeli, C. Sarraute, and G. Richarte, "Attack Planning in the Real World," in *Workshop on Intelligent Security (SecArt 2010)*, 2010.

[24] K. Durkota and V. Lisy, "Computing Optimal Policies for Attack Graphs with Action Failures and Costs," in *7th European Starting AI Researcher Symposium (STAIRS'14)*, 2014.

[25] J. Pereira and O. Díaz, "Chatbot Dimensions that Matter: Lessons from the Trenches," in *Proceedings of the International Conference on Web Engineering (ICWE 2018)*, 2018.

[26] "Botium - new generation testing," http://www.botium.at, accessed: 2018-05-07.

[27] M. Vasconcelos, H. Candello, C. Pinhanez, and T. dos Santos, "Bottester: Testing Conversational Systems with Simulated Users," in *Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems (IHC 2017)*, 2017.