This is the peer reviewd version of the followng article:

RV4Chatbot: Are Chatbots Allowed to Dream of Electric Sheep? / Gatti, A., Mascardi, V., Ferrando, A.. - In: ELECTRONIC PROCEEDINGS IN THEORETICAL COMPUTER SCIENCE. - ISSN 2075-2180. - 411:(2024), pp. 73-90. (6th International Workshop on Formal Methods for Autonomous Systems, FMAS 2024 Manchester, eng 11/11/2024 - 13/11/2024) [10.48550/arxiv.2411.14368].

Open Publishing Association
Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

14/06/2026 01:50

(Article begins on next page)

RV4Chatbot: Are Chatbots Allowed to Dream of Electric Sheep?

Andrea Gatti Viviana Mascardi
Department of Informatics, Bioengineering, Robotics and Systems Engineering
University of Genoa
Genoa, Italy
forename.surname@unige.it

Angelo Ferrando
Department of Physics, Informatics and Mathematics
University of Modena and Reggio Emilia
Modena, Italy
angelo.ferrando@unimore.it

Chatbots have become integral to various application domains, including those with safety-critical considerations. As a result, there is a pressing need for methods that ensure chatbots consistently adhere to expected, safe behaviours. In this paper, we introduce RV4Chatbot, a Runtime Verification framework designed to monitor deviations in chatbot behaviour. We formalise expected behaviours as interaction protocols between the user and the chatbot. We present the RV4Chatbot design and describe two implementations that instantiate it: RV4Rasa, for monitoring chatbots created with the Rasa framework, and RV4Dialogflow, for monitoring Dialogflow chatbots. Additionally, we detail experiments conducted in a factory automation scenario using both RV4Rasa and RV4Dialogflow.

1 Introduction

On November 30th, 2022, ChatGPT was unveiled [40] deeply shaking the industry and academic worlds. The impression, at that time, was that ChatGPT and chatbots based on Large Language Models (LLM) would have irreversibly changed the way chatbots were designed and built, wiping away any pre-existing technology.

After almost two years, a more balanced view on the future is emerging, with the shared feeling that there is still room for chatbots that do not rely on generative AI techniques.

There are many ways to classify chatbots based for example on the knowledge domain, the service provided, the goals, the response generation method [2], the locus of control (chatbot- or user-driven) and duration of the interaction (short or long) [23], or their affordances and disaffordances [32, 35]. For the purposes of this paper, the simplest and most suitable classification of text-based chatbots divides them into conversational AI and generative AI ones [18].

DialogFlow [28, 42], Rasa [11, 41], Wit.ai [38, 39], just to name a few, are text-based conversational AI chatbots, also referred to as intent-based chatbots. They can understand the users questions, no matter how they are phrased, thanks to Natural Language Understanding (NLU) capabilities that allow them to detect the user's intent and further contextual information. The NLU component exploits machine learning techniques for the intent classification and performs well even with a small amount of training sentences. The answer to be provided to the user is not autonomously generated by the chatbot, but is designed by the chatbot's developer. Conversational AI chatbots can remember conversations with users and incorporate contextual information into their interactions.

ChatGPT, Gemini [29], Jasper Chat [31] are examples of generative AI chatbots. They go far beyond conversational AI chatbots thanks to their capability of generating new content as their answer in form of high-quality text, images and sound based on LLMs they are trained on. This impressive power,

Matt Luckcuck and Mengwei Xu (Eds.):
Sixth International Workshop on Formal Methods for Autonomous Systems (FMAS 2024)
EPTCS 411, 2024, pp. 73–90, doi:10.4204/EPTCS.411.5

© A. Gatti, V. Mascardi & A. Ferrando
This work is licensed under the Creative Commons Attribution License.

however, does not come without pitfalls. Besides religious bias [1], gender bias and stereotypes [33], and hallucinations [48], major privacy concerns are associated with LLMs.

In March 2023, Italy's data regulator imposed a temporary ban on ChatGPT due to concerns related to data security. During its development, in November 2023, an open letter was signed by nine Italian scientific associations including the Italian Association for AI and the Italian Association for Computer Vision, Pattern Recognition and Machine Learning, and by around 500 scientists, asking the Italian government to guarantee that strict rules for the use of generative AI were included in the European AI Act [21].

Scientific studies on LLM privacy leakage are so recent to be still unpublished at the time of writing, but many pre-prints by academic scholars show that the problem is real [17,46,47]. Personal Identifiable Information (PII) protection can only be complied with by organisations able to have a private installation of a LLM within a private cloud or on premise [30]. The resources needed to implement this solution make it not affordable for most companies and universities.

The global LLM market size (that includes the generative AI chatbot market plus a wide range of other applications) is projected to reach 259,886 Million USD revenue by 2029 [26], while the conversational AI market is expected to reach 29,800 Million USD by 2028 [37]: the market forecasts and the privacy, ethical, and economical issues of LLM suggest that traditional conversational AI chatbots will still be needed and used by many players in the next few years.

Although more controllable than their generative evolution, the behaviour of conversational AI chatbots can also be unsafe. In a factory automation scenario, where an intent-based chatbot provides a natural language interface between the user and a virtual representation of a factory, a conversation becomes unsafe if the user requests to position an object where another object has already been placed, or if the distance between objects is insufficient. Similarly, a conversation is unsafe if the chatbot provides the coordinates of an object that the user never inserted.

To cope with safety issues in conversational AI chatbots, we present an approach to verify at runtime the conversation between the user and the chatbot. Runtime Verification (RV) [8] is a formal verification technique used to analyse the runtime behaviour of software and hardware systems concerning specific formal properties. A RV monitor emits boolean verdicts that state whether the property is satisfied or not by the currently observed events. The default functioning is to state that something went wrong when it just went wrong, and trigger *recovery* actions. In some cases, the monitor may intervene before the wrong event is generated or the unsafe action is done, hence allowing for *prevention*. With respect to other formal verification techniques, such as Model Checking [19] and Theorem Provers [36], RV is more dynamic and lightweight and shares some similarities with software testing, being focused on checking how the system behaves while it is running.

To perform RV of chatbots, we have designed a general and formalism-agnostic framework named RV4Chatbot. We show RV4Chatbot versatility by instantiating it for RV of chatbots created with Rasa, widely used to develop chatbots in local environments, and Dialogflow, used in cloud-based applications. We demonstrate how our engineering decisions render RV4Chatbot a highly practical methodology and how it can seamlessly integrate with existing chatbot frameworks. It is essential to note that our utilisation of Rasa and DialogFlow serves only to illustrate potential applications of our approach. Our ultimate aim is to encompass any chatbot development framework.

The paper is structured as follows. After overviewing the related work in Section 2, Section 3 introduces one example that motivates the need of RV4Chatbot. After that, Section 4 describes the architecture and the data and control flow of RV4Chatbot. Sections 5 and 6 describe, respectively, RV4Rasa and RV4Dialogflow, the two concrete instantiations of the RV4Chatbot logical architecture. Section 7 discusses the formalisation of some relevant safety properties in the motivating scenario, using a highly

expressive RV language, and the experiments we carried out to verify those properties with RV4Rasa and RV4Dialogflow. Section 8 concludes the paper and highlights the possible future directions.

2 Related Work

RV of interaction protocols attracted the attention of researchers starting from the beginning of the millennium. The first interaction protocols to be verified at runtime involved web services [34], cloud applications [44], cryptography [9]. RV of interactions among autonomous software agents followed soon [3, 7][1].

Despite the large interest in RV of interactions and the pressing need to monitor what chatbots say and do, to the best of our knowledge no studies on RV of human-chatbot interactions exist, if we exclude the very recent works where we were involved.

Apart from [22] which serves as the foundation for this paper but is tailored for a specific chatbot framework, the only other work in the literature that deals with the formal verification of chatbot systems is [20], introducing a framework known as RV4JaCa. In that work we integrated RV within the multiagent system (MAS) domain and demonstrated how to monitor agent interaction protocols within that context. The focus there was not on the chatbot itself, but on the software agents interacting with it. The main contribution was hence in the MAS domain, although applied in a scenario where messages for agents are generated by a chatbot.

Expanding the boundaries of our investigation, we can mention a recent proposal that approaches formal verification of chatbots from a static perspective, instead of at runtime as we do. In [25] the authors introduce a strategy for verifying chatbot conversational flows during the design phase using the UPPAAL tool [10], a well-known model checker. The approach is tested by designing a hotel booking chatbot and receiving feedback from developers. The strategy is found to have an acceptable learning curve and potential for improving chatbot development. In contrast to our approach, the work presented in [25] focuses on abstracting the chatbot using a model and subsequently verifying it through model checking. Due to the distinct inherent natures of these two verification approaches, we envision the possibility of integrating them to harness their respective strengths. Specifically, our technique could enhance the visibility of [25] by providing information that is only available at runtime. Conversely, the exhaustiveness of [25] could be leveraged by our approach to simplify the properties for monitoring, thanks to prior knowledge of the chatbot's behavioural model.

If we further expand our search and give up formality, hence resorting to software testing of chatbots, some works from J. Bozic's research group can be mentioned. The paper [14] introduces a planning-based testing approach for chatbots, focusing on functional testing, specifically in the context of tourism chatbots for hotel reservations. Planning is used to generate test scenarios, and a testing framework automates the execution of test cases. The results show success in testing chatbots, but some issues, such as intent recognition errors, need further attention. Metamorphic testing is illustrated in [15], where metamorphic relations are used instead of traditional test oracles due to the unpredictable nature of AI systems. On a similar line of research, the work [13] introduces an approach that leverages ontologies to generate test cases and addresses the absence of a test oracle by using a metamorphic testing approach. The method is demonstrated on a real tourism chatbot.

A methodology that automates the generation of coherence, sturdiness, and precision tests for chatbots and leverages the test results to enhance their precision is presented in [16]. The methodology is

[1] Starting from 2012, a large share of the scientific production of the authors dealt with RV of agent interaction protocols, see `https://rmlatdibris.github.io/biblio.html`. We limit ourselves to cite the most relevant works among these.

implemented in a tool called Charm, which uses Botium [12] for automated test execution. The paper also presents experiments conducted to improve third-party-built DialogFlow chatbots.

While these works share similarities with ours in that they focus on the actual, runtime execution of the chatbot rather than its abstraction, none is based on a rigorous and formal specification that guides the correctness checks.

To conclude, we emphasise that our goal is to assess the overall correctness of a conversation between a user and a chatbot as a coherent whole, rather than focusing on how individual message utterances are generated by the chatbot itself. This distinction is pivotal and sets our work apart from existing literature on the formal verification of Machine Learning models, as surveyed in [43]. In such literature, the emphasis is generally on verifying the Machine Learning model. In contrast, in RV4Chatbot, our focus is not on whether the model correctly produces or classifies individual messages, but rather on the consistency of these messages within a conversation. In this sense, RV4Chatbot delves deeper into the conversational semantics of the messages exchanged with or generated by the chatbot, rather than attempting to dissect the chatbot to understand its internal behaviour, which is often treated as a black-box.

3 Motivating Example

Chatbots can be exploited for achieving three main goals: providing specific information stored in a fixed source (information chatbots); holding a natural conversation with the user (chat-based chatbots); and understanding the tasks that the user wants to perform, hence executing functions to perform them (task-based chatbots) [2].

Usually, information chatbots provide an answer to one questions and go back to a state – the only state they can be – where they are ready to answer a new question. There is no need for them to keep memory of what the user already asked or said, and to carry out a coherent and fluent conversation. The correctness of the chatbot is related with the correctness of the search engine in its backend. Given that we aim at verifying the conversation flow rather than the quality of the retrieved information, RV of information chatbots following our approach is out of our scope.

Chat-based and task-based chatbots, on the other hand, engage into conversations that should evolve in different ways depending on what the user utters. For example, a chat-based chatbot may show different reactions to the very same request from the user, depending on how much the user insists upon it. In a task-based chatbot, the possibility for the user to ask for some task to be performed may depend on the fact that some prerequisite task had been asked, and hence performed, before.

Without loss of generality, the following motivating example focuses on a task-based chatbot. The application of RV4Chatbot to chat-based chatbots is left for future work, as it would mainly require adapting the types of properties to be verified, rather than altering the verification methodology. In essence, RV4Chatbot is not restricted to task-based chatbots; its architecture is sufficiently flexible to support the verification of any intent-based chatbot.

A Task-based Chatbot in the Factory Automation Domain. This example, presented in the VORTEX 2023 workshop [22] and briefly summarised here, is set in the field of robotics and involves the development of a task-based chatbot assisting in the creation of a simulated factory work floor. The chatbot's role is to guide users through this process, taking into account both the users' requirements and the factory regulations[2] concerning what can or cannot be added or removed from the factory work

[2]See ISO 10218-1:2011 standard on Robots and robotic devices - Safety requirements for industrial robots [45].

floor for safety reasons. The user interacts with the chatbot by requesting to add a robot to a specific position on the factory work floor, removing a robot, or relocating a previously added robot to a different position. For example, properties verified at runtime might include ensuring that objects are not added to an already occupied position on the factory floor, confirming that each removal request corresponds to an object that actually exists in the current state, and enforcing spacing rules between objects as defined by safety regulations. The validity of these actions depends on the state of the simulated work floor, which evolves as the user-chatbot conversation progresses. RV4Chatbot checks these properties dynamically to detect and prevent any violations that could compromise the safety or coherence of the conversation.

4 RV4Chatbot: The Foundation

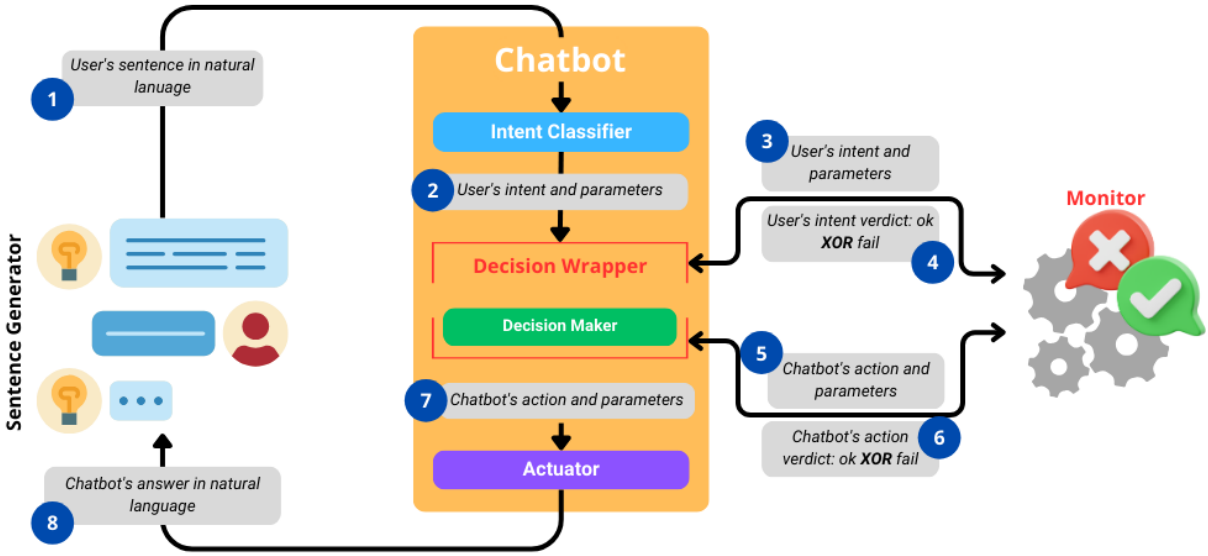

Figure 1: RV4Chatbot architecture.

Figure 1 illustrates the operation of an intent-based chatbot. RV4Chatbot specifically focuses on verifying intent-based chatbots, leaving the verification of non-intent-based chatbots for future work, as mentioned in both the introduction and conclusion sections.

A human user, or more generally, a sentence generator, produces a sentence in natural language (1). This sentence is categorised by a classifier based on its intent, and its parameters are extracted. The intent classifier is responsible for Natural Language Understanding (NLU). The recognised intent, along with its parameters (2), is then passed to a decision maker that determines the chatbot's response (7) to the input sentence. This decision-making process is typically hard-coded and integrated directly into the chatbot framework. Finally, the chatbot generates the response to be delivered to the user (8). The module responsible for this generation process is termed the 'actuator'.

RV4Chatbot introduces a decision wrapper, depicted in red with right angles inside the chatbot architecture, to manage actions numbered (3) to (6) on the right side of the figure. The decision wrapper extends the decision maker to allow the data characterising a chatbot's lifecycle—user's intents and chatbot's actions, both with optional parameters—to be sent to an external monitor where Runtime Verification (RV) occurs. Depending on the chatbot's framework and its modularity, implementing the decision

wrapper may vary in complexity and intrusiveness. The decision wrapper instruments the 'System Under Scrutiny' (the chatbot in this application), using standard RV terminology. In RV4Chatbot, instrumentation is confined solely to this module.

The decision wrapper sends the recognised intents and parameters (3) to the monitor. Regardless of its implementation and the language used for modeling properties to verify, the monitor observes one event at a time and emits a boolean verdict indicating whether the event complies with the property (4). If the verdict is true (or inconclusive[3]), control returns to the decision maker, which decides the subsequent action. Before executing the action, the decision wrapper sends it to the monitor (5), which again verifies its compliance with the property and emits a verdict (6).

A true (or inconclusive) verdict from the monitor does not alter the chatbot's standard execution flow. A false verdict—whether originating from an unexpected user intent or a disallowed action by the chatbot—returns the chatbot to a listening state, displaying a message explaining the failure to the user. In both cases, no unsafe actions are performed.

Numerous intent-based chatbot frameworks are documented in the literature. Although their implementations may vary significantly, their main components and functionalities are accurately represented in Figure 1. Similarly, many RV monitors exist. Regardless of the monitor used, it must at least be able to observe events from the System Under Scrutiny and output a verdict that is either true, false, or inconclusive. This is the only assumption we make regarding the RV monitor's function, and it is satisfied by the definition of a monitor. Thus, the RV4Chatbot logical architecture is parametric in both the chatbot framework and the monitor.

To automate the experiments presented in Section 7.3, we developed a piece of software capable of reading natural language sentences from a file and sending them to the chatbot using the APIs provided by the chatbot frameworks considered in this paper. Although our primary interest lies in RV, we soon realised that the files of simulated user sentences could be seen as test cases, and that the software component named 'sentence generator' in Figure 1 (left side) could be used to run batches of tests. We re-engineered this component and elevated it to the status of one of the RV4Chatbot components. This approach allows testing the chatbot during its development by exploiting the monitor as an offline test engine. The advantage of this method is that once the chatbot has been tested offline and then deployed, the monitor can continue to function at runtime, in line with its primary objective. No code changes are required in the monitor or the instrumented chatbot when switching from offline testing to RV; only the source of sentences changes, becoming a human user in the latter case.

Now that we have completed the introduction of RV4Chatbot, we can focus on its two instantiations for the RV of Rasa and Dialogflow chatbots.

5 RV4Rasa

5.1 Rasa

"*Rasa Open Source is an open source conversational AI platform allowing developers to understand and hold conversations, and connect to messaging channels and third party systems through a set of APIs.*"[4]

Rasa [11] is composed by two different tools: Rasa NLU and Rasa Core. When a message is received from the user, Rasa NLU extracts the intent and the entities (namely, structured pieces of information

[3] We remind the reader that in RV, it is common to have at least a third outcome indicating that the monitor does not yet have sufficient information to determine whether the property under analysis is satisfied or violated.

[4] `https://rasa.com/docs/rasa/`

inside a user message) from it. The structured information is then passed to the Tracker object. This object is used to store the dialogue state. The tracker object is then passed to the policies. Each policy has a ranking and can return a list containing one score for every possible action to perform next. Rasa Core will perform the action with the best score provided by the highest ranked policy. The action server executes the action, the tracker object is updated and then passed again to the policies. When no more actions to be performed are available, the policies return the 'listen' action, waiting for a user input.

Rasa NLU and policies use three files for training:

- `nlu.yml`, containing all the example sentences uses to train the intent classification and the entities extraction;
- `stories.yml`, containing all the paths that a conversation can follow;
- `rules.yml`, containing stricter conversation patterns and actions that must take place if triggered.

Rasa Core employs two main files for the flow control and configuration:

- `domain.yml`, containing all the information on what Rasa NLU can extract (intents and entities), definition of slots (stored values), available actions and responses;
- `config.yml`, containing the pipeline for the Rasa NLU training and the policies definition.

Rasa actions can be defined either as strings to answer or as complete Python classes to be executed when called.

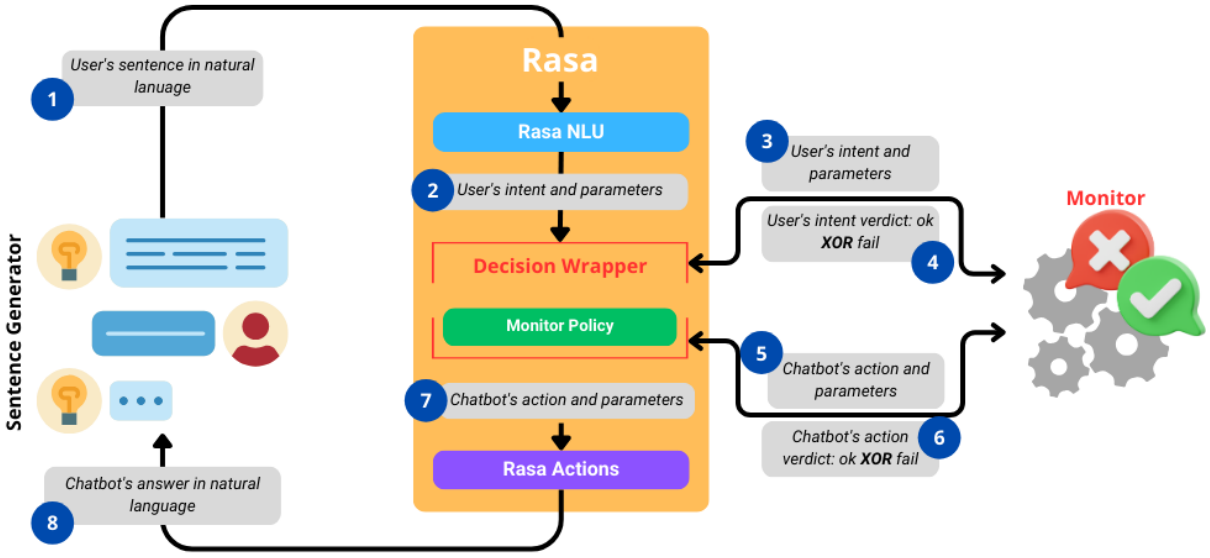

Figure 2: The RV4Rasa instantiation of RV4Chatbot.

5.2 The RV4Rasa instantiation of RV4Chatbot

The Rasa architecture aligns closely with that of RV4Chatbot, as shown in Figure 2. Each element of the RV4Chatbot architecture maps directly to a specific component in the RV4Rasa instantiation, without requiring any modifications to Rasa's original design.

The Sentence Generator can be either the human user using the shell provided by Rasa or a script that sends messages as POST requests to the Rasa server provided by Rasa. The Intent Classifier is the

Rasa NLU that extracts intent and entities. The Decision Wrapper in Rasa is managed by the policies. In particular, for this module it is necessary to add a policy (`monitorPolicy`) that sends an event to the monitor for each action executed. The Actuator overlaps with the Rasa Actions that can perform any piece of provided Python code.

Notice that the policies predict the next action based on the previous ones so the `monitorPolicy` can only stop the chatbot immediately after the wrong action has been executed. This paves the way to *recovery* from wrong actions, but not to *prevention*. However, by exploiting Rasa policies the developer only needs to add the `monitorPolicy` to them, without any other change to the chatbot; RV will be performed automatically thanks to the `monitorPolicy`. The simplicity and the minimal invasiveness of 'injecting' RV capabilities into Rasa this way, motivates our decision to give up prevention, and accept that the monitor realizes that something went wrong, after this already happened. Actually, ex-post notification is a standard operating way in RV.

5.3 Challenges in the RV4Rasa design and development

The main effort required by the RV4Rasa development was understanding how policies work, and implementing the `monitorPolicy`. In fact, whereas Rasa's documentation is extensive and well-assorted for a basic usage, it is almost completely absent when policies come into play. The policy is added in the config file as follows:

```
policies:
...
  - name: policies.monitorPolicy.MonitorPolicy
    priority: 6
    error_action: "utter_error_message"
```

In its implementation the main class, `monitorPolicy`, inherits Rasa's Policy class; in particular, it inherits and redefines:

- `__init__`, the initialisation method, here the error action provided by the user is saved or set to a default value if needed;
- `predict_action_probabilities`, called every time the policy runs and returning the list of probabilities. This method may also return no value at all, and this is exactly the way we use it, to keep the conversation flowing as if no RV were performed, if no errors occur.

Note that, the `priority` assigned to the policy is user-defined and ensures that Rasa gives precedence to the `monitorPolicy` over other custom policies. This higher priority is crucial since the `monitorPolicy` addresses safety aspects, which must take precedence in the chatbot's decision-making process.

5.4 Source Code

To instantiate RV4Rasa there are only two additions to be made in the chatbot:

1. `monitor_policy.py`: 150 lines of code;
2. `config.yml`: 3 further lines should be added to the configuration file, to turn Rasa into RV4Rasa.

The code of RV4Rasa is available at `https://github.com/driacats/RV4Chat/tree/main/Rasa`.

6 RV4Dialogflow

6.1 Dialogflow

Dialogflow [27] is a lifelike conversational AI platform developed by Google that enables users to create virtual agents equipped with intents, entities (similar to those in Rasa), and fulfillment. Fulfillment refers to the capability of these agents to interact with external systems or APIs to retrieve dynamic responses, process data, or execute specific actions based on the user's input, going beyond pre-defined static responses.

Dialogflow performs NLU using *intents*, defined via a name and a set of training example sentences. Dialogflow trains a model able to identify, for each user message sent on the chat, the *nearest* intent and the confidence score. Training sentences may also contain *entities*, namely pieces of information that may be significant for the conversation and that should be extracted from the text. For each intent, a bunch of possible answers may be displayed. However, some messages cannot be answered from inside Dialogflow, as they require to process data or execute operations. In this case, users can use the *fulfillment* for sending a message to an external server that will execute the correct actions, and provide an answer back.

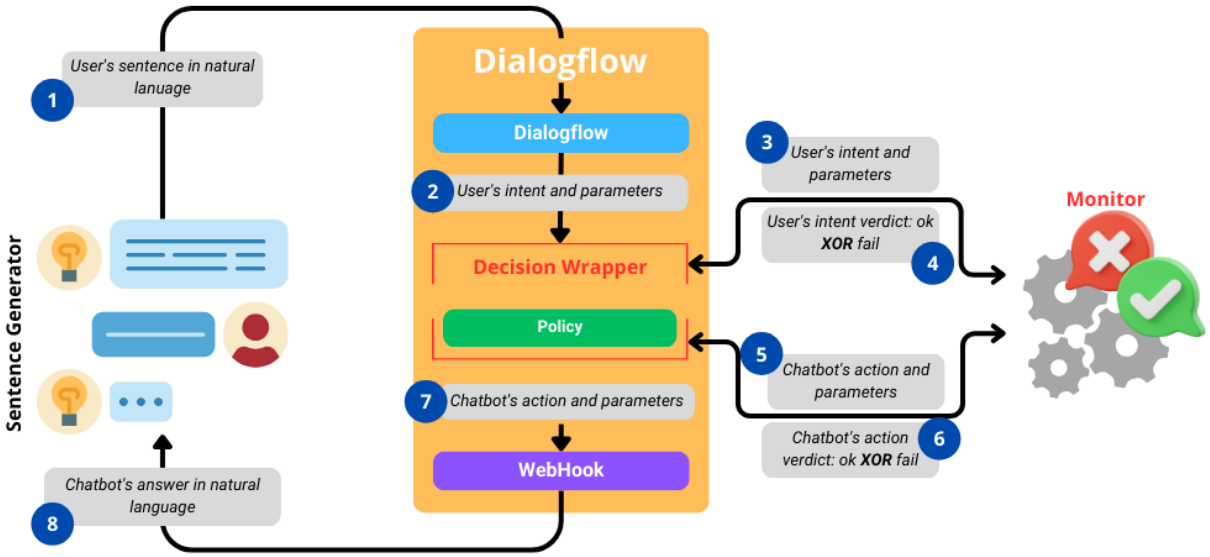

Figure 3: The RV4Dialogflow instantiation of RV4Chatbot.

6.2 The RV4Dialogflow instantiation of RV4Chatbot

RV4Dialogflow is structured as shown in Figure 3. To instantiate RV4Chatbot in Dialogflow, we had to add a brand new component to the Dialogflow architecture. This made the design and implementation of RV4Dialogflow much more complex than the RV4Rasa one.

This additional component can be generated directly from an exported Dialogflow agent using an instrumentation script that we developed. We call this brand new component *policy*, for analogy with RV4Rasa.

The instrumentation script has two outputs: a .zip file containing the new Dialogflow agent, and the policy python script. In the new Dialogflow agent, every message is forwarded to the policy that controls

the flow. The policy maintains the original agent's flow, forwarding only the necessary messages back to DialogFlow through a webhook[5]. Additionally, for each message and action performed, the policy sends a message to the monitor.

6.3 Challenges in the RV4Dialogflow design and development

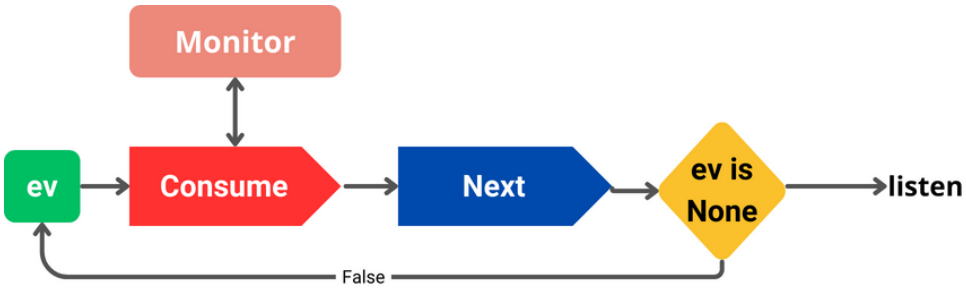

Figure 4: RV4Dialogflow policy flow.

The RV4Dialogflow policy works with *events* as shown in Figure 4 (reported as ev). There are two main types of events: user and bot events.

The policy can receive messages both from the user or the bot. When it receives a message it creates an event. An event in this domain can be of five main types: (1) a user message with all its features; (2) a bot message with all its features; (3) a plain answer to be sent as answer; (4) an action to be performed by the webhook; (5) the error action.

The event is then consumed: it is sent to the monitor and then if it is a plain answer it is sent on the chat, if it is an action it is performed. Obviously, if the monitor claims an error the action is set immediately to the error one.

When the event is consumed the policy computes the next one. For examples, if the event is a user message the next event is the answer. The policy consumes and computes next event until the next event is None, in this case it listens for new inputs.

6.4 Source Code

To instantiate RV4Dialogflow the needed files are:

1. `instrumenter.py`: 190 lines of code;
2. `policy.py`: (generated by the instrumenter) from 150 lines of code;

The code of RV4Dialogflow is available at `http://github.com/driacats/RV4Chat/tree/main/Dialogflow`.

7 Experiments

The formalism we use to model properties to be verified at runtime is named Runtime Monitoring Language (RML) and has been selected for its high expressive power that goes beyond Linear Temporal

[5]Which is the mechanism used in DialogFlow to communicate to DialogFlow from an external service, that in RV4Dialogflow is the policy.

Logic (LTL) [5] and the familiarity of the authors. As explained in Section 4, the RV4Chatbot framework is meant to be instantiated with any conversational chatbot framework and any RV language and tool. RML is one among the many existing RV languages and its adoption is only functional to run experiments with RV4Rasa and RV4Dialogflow.

In this section, we briefly introduce RML and illustrate the RML properties that capture safety requirements in the factory automation case study, providing the RML encoding of one of the properties verified in that scenario (the complete encoding can be found in [22]). All these properties are correctly verified by the monitor, so we do not allocate space to the qualitative experiments we conducted, as they can be summarised by stating that "the monitor always works as expected". Instead, we present performance experiments, demonstrating that the addition of the monitor to the chatbot introduces negligible overhead.

7.1 Runtime Monitoring Language

The Runtime Monitoring Language (RML [4, 6]) is a Domain-Specific Language (DSL) for specifying highly expressive properties in RV such as non context-free ones. We chose to use RML in this work because of its support of parametric specifications and its native use for defining interaction protocols.

Since RML is just a means for our purposes, we only provide a condensed view of its syntax and denotational semantics in terms of the represented traces of events. A detailed explanation of some of its operators is provided in Section 7.2 where RML specifications are provided. The complete presentation can be found in [6].

In RML, a property is expressed as a tuple $\langle t, ETs \rangle$, with t a term and $ETs = \{ ET_1, \ldots, ET_n \}$ a set of event types. An event type ET is represented as a set of pairs $\{ k_1 : v_1, \ldots, k_n : v_n \}$, where each pair identifies a specific piece of information (k_i) and its value (v_i). An event Ev is denoted as a set of pairs $\{ k'_1 : v'_1, \ldots, k'_m : v'_m \}$. Given an event type ET, an event Ev matches ET if $ET \subseteq Ev$, which means $\forall (k_i : v_i) \in ET \cdot \exists (k_j : v_j) \in Ev \cdot k_i = k_j \wedge v_i = v_j$. In other words, an event type ET specifies the requirements that an event Ev has to satisfy to be considered valid.

An RML term t, with t_1, t_2 and t' as other RML terms, can be:

- ET, denoting a set of singleton traces containing the events Ev s.t. $ET \subseteq Ev$;
- $t_1 \; t_2$, denoting the sequential composition of two sets of traces;
- $t_1 \mid t_2$, denoting the unordered composition of two sets of traces (also called shuffle or interleaving);
- $t_1 \wedge t_2$, denoting the intersection of two sets of traces;
- $t_1 \vee t_2$, denoting the union of two sets of traces;
- $\{ \; let \; x; \; t' \; \}$, denoting the set of traces t' where the variable x can be used (i.e., the variable x can appear in event types in t', and can be unified with values);
- $t'*$, denoting the set of chains of concatenations of traces in t'.

Event types can contain variables (we use the terms *argument* and *variable* interchangeably). In RML, recursion is modelled by syntactic equations involving RML terms, such as $t = ET_1 \; t \vee ET_2$, modeling a finite (possibly empty) sequence of events matching the event type ET_1 ended by one event matching ET_2, or the infinite trace including only events matching ET_1.

7.2 Factory Automation Domain properties

The three properties to be verified in the factory automation domain have been presented in the VORTEX 2023 paper [22]. We report one of them to better clarify the use of RML and the kind of protocols we are interested in verifying at runtime.

The first property aims at ensuring that the user does not add an object in an already taken position. The corresponding RML specification is reported in the following (as in [22]).

$$
\begin{aligned}
AddObject = \{\ & let\ x, y; \\
& (msg\_user\_to\_bot \wedge add\_object(x,y)) \\
& (msg\_bot\_to\_user \wedge object\_added) \\
& (not\_add\_object(x,y)* \wedge AddObject)\ \} \\
ETs = \{\ & msg\_user\_to\_bot, \\
& msg\_bot\_to\_user, add\_object(x,y), \\
& object\_added\ \} \\
msg\_user\_to\_bot = \{\ & sender: \text{“}user\text{”},\ receiver: \text{“}bot\text{”}\ \} \\
msg\_bot\_to\_user = \{\ & sender: \text{“}bot\text{”},\ receiver: \text{“}user\text{”}\ \} \\
add\_object(x,y) = \{\ & intent: \{\ name: \text{“}add\_object\text{”}\ \}, \\
& slots: \{\ horizontal: x, vertical: y\ \}\ \} \\
object\_added = \{\ & last\_action: \text{“}utter\_add\_object\text{”}\ \}
\end{aligned}
$$

As an example, the user's request *"Add a robot in position (3, 5)"* is safe only if the position *(3, 5)* is empty. But, the position is empty if the user did not already ask to put objects there. Hence, the history of the previous interactions must be taken into account, to verify the feasibility of a new object addition. The property is parametric w.r.t. coordinates and is defined recursively; it involves the definition of four event types and exploits the *let*, $\wedge$, and $*$ RML operators.

Figure 5 presents screenshots of the simulated environment in which the Rasa (and Dialogflow) chatbots operate. These screenshots specifically demonstrate how, by interacting with the chatbot, the user can add new objects to the simulated factory floor. This interaction occurs under the scrutiny of the RML monitor, which checks, among other things, the previously mentioned property.

The second property, whose RML encoding is more complex than the previous one, deals with the addition of an object in a position which is relative to another object in the simulation. The user's request may be *"Add a robot on the left of Robot3"*. In order to be a safe request, *Robot3* should have been previously positioned somewhere, hence previous messages involving added and removed objects, their name, and their position in the simulation must be taken into account. The property is parametric w.r.t. coordinates and objects names, and defined recursively; it involves eight event types and exploits the $|$ and $\vee$ RML operators, besides *let*, $\wedge$, and $*$. The $|$ operator is used, for example, to cope with the interleaving of future additions relative to the currently added object and additions of objects that are not relative to the newly added one. Disjunction is used to discriminate between the situation where one added object is then removed, and hence no further references to it are allowed, and the situation where no removal takes place, and references are safe.

The third property exploits the RML feature of constraining values in event types. It checks that any observed message has the value associated with its *confidence* field – as returned by the NLU component of the chatbot – greater than 60%.

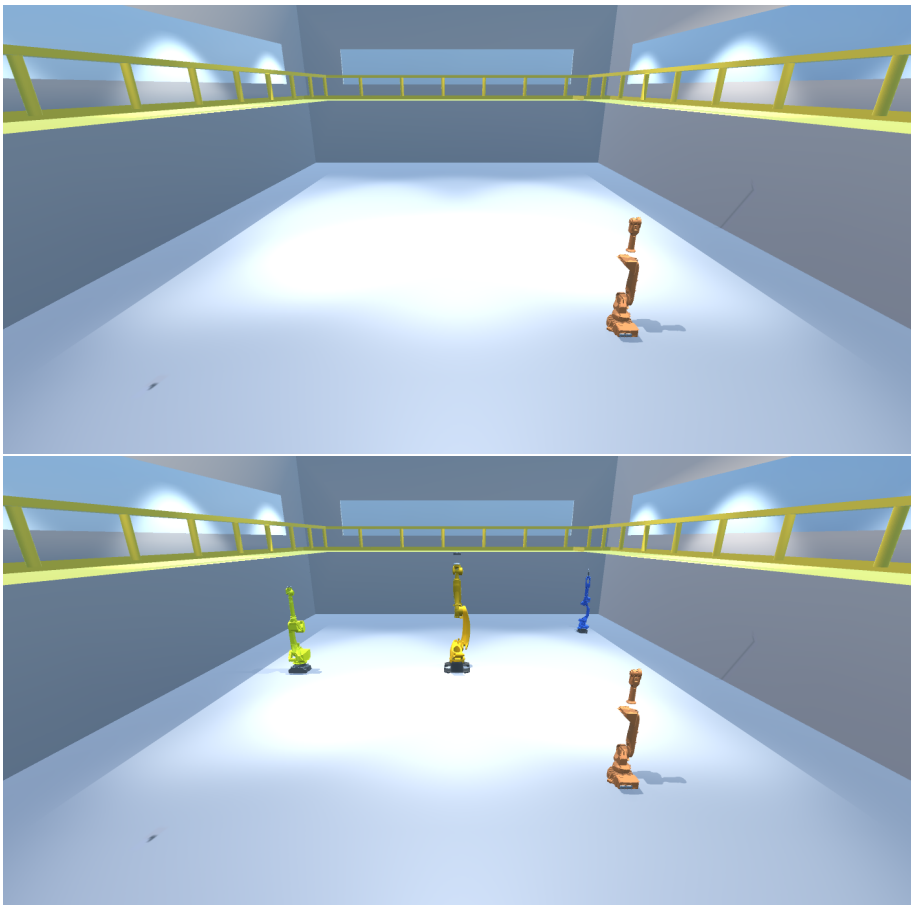

Figure 5: Initial scenario of the simulated factory floor (above) and the result after further iterations of adding new objects in the scene (below) taken from [24].

7.3 Performance Evaluation

All the experiments can be tested using the code provided here `https://github.com/driacats/RV4Chat/tree/main/Examples`. In particular there are three important scripts for each experiment:

- `start_service.py`: this script allows the user to select the platform (Rasa, Dialogflow), the monitor (no monitor, dummy monitor, real monitor), and the scenario (factory automation), and starts the service;
- `run_test.py`: this script launches the test conversation. The input messages are stored in the `test_input.txt` file, and the conversation is iterated a certain number of times for each combination of platform and monitor. For each iteration, a file is created to store the response times for each message sent;
- `chat.py`: this script provides a chat interface to test the chatbot. It takes as argument the platform and manages the connection automatically. To test the program it is sufficient to start the service and then launch the chat with the same platform.

The tests have been performed using RV4Rasa and RV4Dialogflow with three different monitoring levels:

1. without a monitor;

2. with a dummy monitor that replies always `True`;
3. with a real monitor that checks the properties discussed in the previous sections.

For this experiment the chatbot can identify three intents and five entities. The three intents are (1) add an object (2) add a object with a reference to another object (3) remove an object, while the entities are (1) object to add or remove (2) vertical position (3) horizontal position (4) relative position (5) reference object.

The Dialogflow WebHook in this case is more complex and manages the addition and the removal of objects inside a real virtual environment. The implementation of this experiment in Rasa, with a real Virtual Environment in the backend and a Multi-Agent System in the middle, has been presented in [22]. For the tests presented here, the API calls that in [22] accessed the virtual environment are instead sent to a dummy script that provides a terminal based representation of a virtual space and simulates the execution. No virtual environment implementation is involved in this experiment, which is aimed at testing the performance of the RV mechanism.

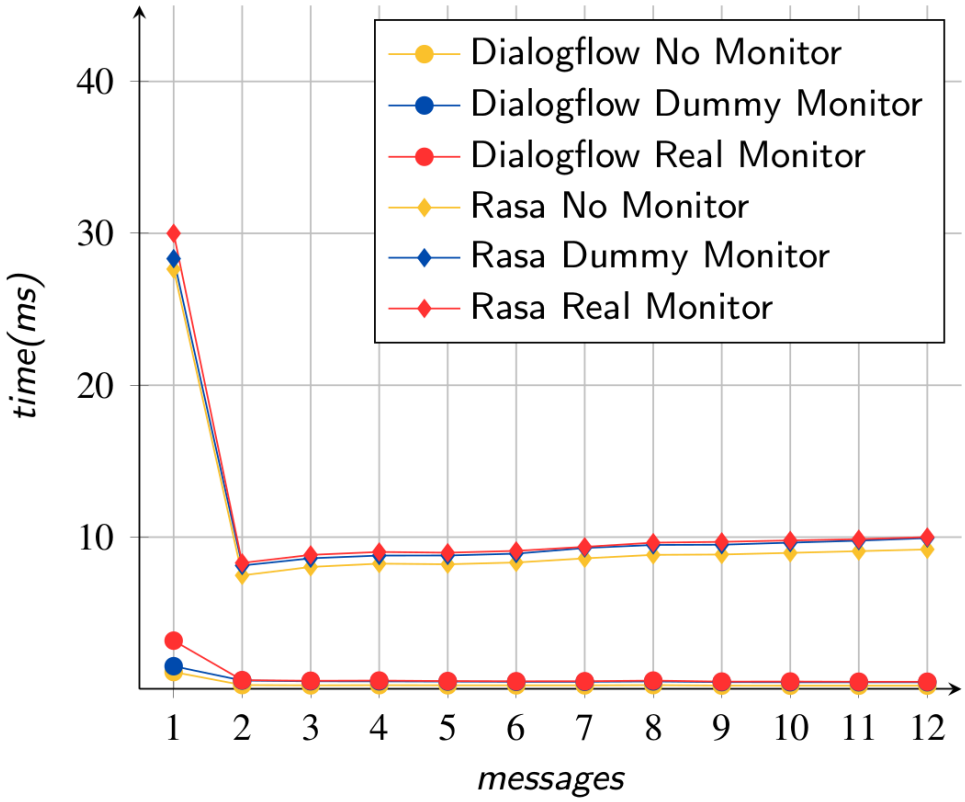

Figure 6: Factory Automation Domain times on a test conversation of 12 messages. Messages for the test are: (1) *Add a table* (2) *Add a box right of table1* (3) *Add a robot in front on the left* (4) *Add a robot in front on the right* (5) *Remove box0* (6) *Remove robot1* (7) *Add a table behind on the left* (8) *Add a robot behind on the right* (9) *Remove table1* (10) *Remove table2* (11) *Remove robot2* (12) *Remove robot3*.

As shown in Figure 6, the monitor does not affect the execution time of the chatbot. The first message exhibits a significant time delay compared to the subsequent messages when using Rasa. This behaviour is due to Rasa itself: the Rasa Tracker object and all necessary instances for the conversation are ini-

tialised with the first message rather than at the server launch, resulting in a significantly higher time required to process the first message compared to the others.

8 Conclusions and Future Work

This paper introduces RV4Chatbot, a framework for verifying the behaviour of conversational AI chatbots. RV4Chatbot achieves this in a versatile manner, imposing minimal constraints on both the chatbot creation framework and the monitors deployed at runtime for formal verification. To demonstrate its efficacy, this paper presents two implementations of RV4Chatbot: RV4Rasa and RV4Dialogflow. The engineering and experimental outcomes of these implementations are detailed, particularly when applied to safety-critical case studies in domains such as factory automation.

The experimental findings underscore RV4Chatbot's generality, efficiency, and lightweight nature in terms of the overhead introduced by its monitoring components.

Looking ahead, our plans involve further exploration and experimentation with RV4Chatbot, including its application to more complex case studies that can better challenge the framework's robustness and performance. This will allow us to assess how the performance overhead of RV4Chatbot is impacted when applied to larger, real-world conversational systems with increased message volumes, more complex dialogue flows, and higher interaction frequencies. Additionally, while our current focus is on conversational AI chatbots, we plan to evaluate the scalability of RV4Chatbot to understand how it performs as the number of monitored properties, intents, and concurrent conversations grows. Preliminary intuition suggests that the framework's modularity may support scaling to moderately large applications, but this hypothesis needs to be tested empirically.

Furthermore, the insights and experiences gained from this work may facilitate future developments for handling generative chatbots. In such scenarios, where intents may be unavailable and decision-making is based on machine learning techniques, we aim to refine and expand RV4Chatbot to integrate more dynamic monitoring approaches that can accommodate the unpredictability and complexity of generative AI.

References

[1] Abubakar Abid, Maheen Farooqi & James Zou (2021): *Persistent Anti-Muslim Bias in Large Language Models*. In: *AIES*, ACM, pp. 298–306, doi:10.1145/3461702.3462624.

[2] Eleni Adamopoulou & Lefteris Moussiades (2020): *Chatbots: History, technology, and applications*. *Machine Learning with Applications* 2, p. 100006, doi:10.1016/j.mlwa.2020.100006.

[3] Hind Alotaibi & Hussein Zedan (2010): *Runtime verification of safety properties in multi-agents systems*. In: *10th International Conference on Intelligent Systems Design and Applications, ISDA 2010, November 29 - December 1, 2010, Cairo, Egypt*, IEEE, pp. 356–362, doi:10.1109/ISDA.2010.5687238.

[4] Davide Ancona, Angelo Ferrando, Luca Franceschini & Viviana Mascardi: *RML web site*. Available at `https://rmlatdibris.github.io/`. Accessed on November 22, 2024.

[5] Davide Ancona, Angelo Ferrando & Viviana Mascardi (2016): *Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification*. In: *Theory and Practice of Formal Methods*, *LNCS* 9660, Springer, pp. 47–64, doi:10.1007/978-3-319-30734-3_6.

[6] Davide Ancona, Luca Franceschini, Angelo Ferrando & Viviana Mascardi (2021): *RML: Theory and practice of a domain specific language for runtime verification*. *Sci. Comput. Program.* 205, p. 102610, doi:10.1016/j.scico.2021.102610.

[7] Najwa Abu Bakar & Ali Selamat (2013): *Runtime Verification of Multi-agent Systems Interaction Quality*. In: *Intelligent Information and Database Systems - 5th Asian Conf., ACIIDS 2013*, *LNCS* 7802, Springer, Berlin, Heidelberg, pp. 435–444, doi:10.1007/978-3-642-36546-1_45.

[8] Ezio Bartocci, Yliès Falcone, Adrian Francalanza & Giles Reger (2018): *Introduction to Runtime Verification*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *LNCS* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[9] Andreas Bauer & Jan Jürjens (2010): *Runtime verification of cryptographic protocols*. *computers & security* 29(3), pp. 315–330, doi:10.1016/j.cose.2009.09.003.

[10] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson & Wang Yi (1995): *UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems*. In: *DIMACS/SYCON WS on Verification and Control of Hybrid Systems*, *LNCS* 1066, Springer, pp. 232–243, doi:10.1007/BFB0020949.

[11] Tom Bocklisch, Joey Faulkner, Nick Pawlowski & Alan Nichol (2017): *Rasa: Open Source Language Understanding and Dialogue Management*. *CoRR* abs/1712.05181, doi:10.48550/arXiv.1712.05181. arXiv:1712.05181.

[12] Botium: *Bots Testing Bots*. Available at `https://botium-docs.readthedocs.io/en/latest/`. Accessed on November 22, 2024.

[13] Josip Bozic (2022): *Ontology-based metamorphic testing for chatbots*. *Softw. Qual. J.* 30(1), pp. 227–251, doi:10.1007/s11219-020-09544-9.

[14] Josip Bozic, Oliver A. Tazl & Franz Wotawa (2019): *Chatbot Testing Using AI Planning*. In: *IEEE Int. Conf. On Artificial Intelligence Testing, AITest 2019*, IEEE, pp. 37–44, doi:10.1109/AITest.2019.00-10.

[15] Josip Bozic & Franz Wotawa (2019): *Testing Chatbots Using Metamorphic Relations*. In: *Testing Software and Systems - 31st IFIP WG 6.1 Int. Conf., ICTSS 2019*, *LNCS* 11812, Springer, pp. 41–55, doi:10.1007/978-3-030-31280-0_3.

[16] Sergio Bravo-Santos, Esther Guerra & Juan de Lara (2020): *Testing Chatbots with Charm*. In: *Quality of Information and Communications Technology - 13th Int. Conf., QUATIC 2020*, *CCIS* 1266, Springer, pp. 426–438, doi:10.1007/978-3-030-58793-2_34.

[17] Xiaoyi Chen, Siyuan Tang, Rui Zhu, Shijun Yan, Lei Jin, Zihao Wang, Liya Su, XiaoFeng Wang & Haixu Tang (2023): *The Janus Interface: How Fine-Tuning in Large Language Models Amplifies the Privacy Risks*. *CoRR* abs/2310.15469, doi:10.48550/ARXIV.2310.15469. arXiv:2310.15469.

[18] Bella Church (2023): *5 types of chatbot and how to choose the right one for your business*. Available at `https://www.ibm.com/blog/chatbot-types/`. Accessed on November 22, 2024.

[19] Edmund M Clarke (1997): *Model checking*. In: *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer, pp. 54–56, doi:10.1007/BFb0058022.

[20] Debora C Engelmann, Angelo Ferrando, Alison R Panisson, Davide Ancona, Rafael H Bordini & Viviana Mascardi (2023): *RV4JaCa — Towards Runtime Verification of Multi-Agent Systems and Robotic Applications*. *Robotics* 12(2), p. 49, doi:10.3390/robotics12020049.

[21] European Parliament (2023): *Artificial Intelligence Act*. Available at `https://www.europarl.europa.eu/news/en/press-room/20231206IPR15699/artificial-intelligence-act-deal-on-comprehensive-rules-for-trustworthy-ai`. Accessed on November 22, 2024.

[22] Angelo Ferrando, Andrea Gatti & Viviana Mascardi (2023): *RV4Rasa: A Formalism-Agnostic Runtime Verification Framework for Verifying ChatBots in Rasa*. In: *6th Int. WS on Verification and Monitoring at Runtime Execution, VORTEX 2023*, ACM, pp. 1–8, doi:10.1145/3605159.3605855.

[23] Asbjørn Følstad, Marita Skjuve & Petter Bae Brandtzæg (2018): *Different Chatbots for Different Purposes: Towards a Typology of Chatbots to Understand Interaction Design*. In Svetlana S. Bodrunova, Olessia Koltsova, Asbjørn Følstad, Harry Halpin, Polina Kolozaridi, Leonid Yuldashev, Anna S. Smoliarova & Heiko

Niedermayer, editors: *Internet Science - INSCI 2018 International Workshops, St. Petersburg, Russia, October 24-26, 2018, Revised Selected Papers*, *Lecture Notes in Computer Science* 11551, Springer, pp. 145–156, doi:10.1007/978-3-030-17705-8_13.

[24] Andrea Gatti & Viviana Mascardi (2023): *VEsNA, a Framework for Virtual Environments via Natural Language Agents and Its Application to Factory Automation*. *Robotics* 12(2), p. 46, doi:10.3390/ROBOTICS12020046.

[25] Sousa S. Geovana Ramos, Nunes R. Genaína & Dias C. Edna (2023): *A Modeling Strategy for the Verification of Context-Oriented Chatbot Conversational Flows via Model Checking*. *Journal of Universal Computer Science* 29(7), pp. 805–835, doi:10.3897/jucs.91311.

[26] Global Information, Inc. – GII (2024): *Global Large Language Model (LLM) Market Research Report*. Available at `https://www.giiresearch.com/report/qyr1384359-global-large-language-model-llm-market-research.html`. Accessed on November 22, 2024.

[27] Google: *DialogFlow: Online Resource, `https://cloud.google.com/dialogflow/`*. Available at `https://cloud.google.com/dialogflow/`.

[28] Google: *Dialogflow web site*. Available at `https://cloud.google.com/dialogflow`. Accessed on November 22, 2024.

[29] Google: *Gemini web site*. Available at `https://gemini.google.com/`. Accessed on November 22, 2024.

[30] Cobus Greyling (2023): *Conversational UIs & LLMs*. Available at `https://cobusgreyling.medium.com/large-language-model-llm-disruption-of-chatbots-8115fffadc22`. Accessed on November 22, 2024.

[31] Jasper AI: *Jasper web site*. Available at `https://www.jasper.ai/chat`. Accessed on November 22, 2024.

[32] Jaeho Jeon, Seongyong Lee & Hohsung Choe (2023): *Beyond ChatGPT: A conceptual framework and systematic review of speech-recognition chatbots for language learning*. *Comput. Educ.* 206, p. 104898, doi:10.1016/j.compedu.2023.104898.

[33] Hadas Kotek, Rikker Dockum & David Q. Sun (2023): *Gender bias and stereotypes in Large Language Models*. In: *The ACM Collective Intelligence Conf., CI 2023*, ACM, pp. 12–24, doi:10.1145/3582269.3615599.

[34] Zheng Li, Yan Jin & Jun Han (2006): *A runtime monitoring and validation framework for web service interactions*. In: *Australian Software Engineering Conf. (ASWEC'06)*, IEEE, pp. 10–pp, doi:10.1109/ASWEC.2006.6.

[35] Xiaolin Lin, Bin Shao & Xuequn Wang (2022): *Employees' perceptions of chatbots in B2B marketing: Affordances vs. disaffordances*. *Industrial Marketing Management* 101, pp. 45–56, doi:10.1016/j.indmarman.2021.11.016. Available at `https://www.sciencedirect.com/science/article/pii/S001985012100242X`.

[36] Donald W. Loveland (1978): *Automated theorem proving: a logical basis*. *Fundamental studies in computer science* 6, North-Holland.

[37] MarketsandMarkets (2023): *Conversational AI Market*. Available at `https://www.marketsandmarkets.com/Market-Reports/conversational-ai-market-49043506.html`. Accessed on November 22, 2024.

[38] Meta: *Wit.ai web site*. Available at `https://wit.ai/`. Accessed on November 22, 2024.

[39] Martin Mitrevski (2018): *Getting started with wit.ai*, doi:10.1007/978-1-4842-3396-2_5.

[40] Open AI (2022): *Introducing ChatGPT*. Available at `https://openai.com/blog/chatgpt`. Accessed on November 22, 2024.

[41] Rasa technologies: *Rasa web site*. Available at `https://rasa.com/`. Accessed on November 22, 2024.

[42] Navin Sabharwal, Amit Agrawal, Navin Sabharwal & Amit Agrawal (2020): *Introduction to Google Dialogflow*.

[43] Sanjit A. Seshia, Ankush Desai, Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte & Xiangyu Yue (2018): *Formal Specification for Deep Neural Networks*. In: *Automated Technology for Verification and Analysis - 16th Int. Symposium, ATVA 2018*, *LNCS* 11138, Springer, pp. 20–34, doi:10.1007/978-3-030-01090-4_2.

[44] Jin Shao, Hao Wei, Qianxiang Wang & Hong Mei (2010): *A Runtime Model Based Monitoring Approach for Cloud*. In: *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, IEEE Computer Society, pp. 313–320, doi:10.1109/CLOUD.2010.31.

[45] Technical Committee:ISO/TC 299 Robotics (2011): *Robots and robotic devices – Safety requirements for industrial robots*. Standard.

[46] Xianjun Yang, Xiao Wang, Qi Zhang, Linda R. Petzold, William Yang Wang, Xun Zhao & Dahua Lin (2023): *Shadow Alignment: The Ease of Subverting Safely-Aligned Language Models*. *CoRR* abs/2310.02949, doi:10.48550/ARXIV.2310.02949. arXiv:2310.02949.

[47] Zheng Xin Yong, Cristina Menghini & Stephen H. Bach (2023): *Low-Resource Languages Jailbreak GPT-4*. *CoRR* abs/2310.02446, doi:10.48550/ARXIV.2310.02446. arXiv:2310.02446.

[48] Yue Zhang, Yafu Li, Leyang Cui & et al. (2023): *Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models*. arXiv:2309.01219.