


A Modeling Strategy for the Verification of Context-Oriented Chatbot Conversational Flows via Model Checking


Geovana Ramos Sousa Silva

(Department of Computer Science, University of Brasilia (UnB), Brasilia-DF, Brazil
 <https://orcid.org/0000-0002-0304-0804>, geovanna.1998@gmail.com)

Genáina Nunes Rodrigues

(Department of Computer Science, University of Brasilia (UnB), Brasilia-DF, Brazil
 <https://orcid.org/0000-0003-1661-8131>, genaina@unb.br)

Edna Dias Canedo

(Department of Computer Science, University of Brasilia (UnB), Brasilia-DF, Brazil
 <https://orcid.org/0000-0002-2159-339X>, ednacanedo@unb.br)

Abstract: Verification of chatbot conversational flows is paramount to capturing and understanding chatbot behavior and predicting problems that would cause the entire flow to be restructured from scratch. The literature on chatbot testing is scarce, and the few works that approach this subject do not focus on verifying the communication sequences in tandem with the functional requirements of the conversational flow itself. However, covering all possible conversational flows of context-oriented chatbots through testing is not feasible in practice given the many ramifications that should be covered by test cases. Alternatively, model checking provides a model-based verification in a mathematically precise and unambiguous manner. Moreover, it can anticipate design flaws early in the software design phase that could lead to incompleteness, ambiguities, and inconsistencies. We postulate that finding design flaws in chatbot conversational flows via model checking early in the design phase may overcome quite a few verification gaps that are not feasible via current testing techniques for context-oriented chatbot conversational flows. Therefore, in this work, we propose a modeling strategy to design and verify chatbot conversational flows via the Uppaal model checking tool. Our strategy is materialized in the form of templates and a mapping of chatbot elements into Uppaal elements. To evaluate this strategy, we invited a few chatbot developers with different levels of expertise. The feedback from the participants revealed that the strategy is a great ally in the phases of conversational prototyping and design, as well as helping to refine requirements and revealing branching logic that can be reused in the implementation phase.

Keywords: chatbots, formal verification, conversational design, model checking

Categories: D.2.1.1, D.2.2, D.2.4, I.2.4

DOI: 10.3897/jucs.91311

1 Introduction

Service automation has increasingly leaned towards natural language interactions since it makes the process more natural and easier for users [Følstad & Brandtzæg, 2017]. In this context, chatbots are great allies of organizations that wish to provide immediate support for their customers' needs since they can incorporate a persona and act as a brand representative [Tsai et al., 2021].

Even though the research conducted in the context of chatbots is widely focused on Natural Language Understanding (NLU) algorithms, its human-interaction aspect is also vital to providing a great user experience and satisfied users [Fiore et al., 2019]. This is of great interest to organizations because, when having chatbots as brand representatives, the experience that users have with them reflects on their view of the brand [Følstad et al., 2018]. On this basis, the design of conversational flows is an essential part of creating chatbots within an expected user experience quality.

As with any software, chatbots need to have a well-structured development process that passes by prototyping, development, and testing [Santos et al., 2022]. However, chatbots are very distinct systems and with special needs, given their conversational nature. Unlike other software systems where entry is through clicks or taps on a screen, the input itself is in the form of natural language freely elaborated by users and output in natural language as well. This aspect brings challenges to developers, especially when conceiving the conversational flows of context-oriented chatbots and testing them [Silva & Canedo, 2022].

While for designers the problem of prototyping context-oriented chatbots reduces to building decision trees with dialog boxes, developers need to focus their attention on the logic that dictates what path the chatbot will take based on previous messages [Castle-Green et al., 2020]. Chatbots that simply retrieve the same answer to a question, i.e. FAQ-oriented, are easier to design and implement since their construction is a simple one-to-one mapping. On the other hand, contextual conversations depend on previous responses, and their conversation design turns out to be a decision tree with possibly many ramifications, which can become laborious to test. In current chatbot frameworks, such as ¹Rasa and ²DialogFlow, it is possible to test conversational flows by designing test cases for each possible path of a contextual conversation. Nevertheless, to cover all possible paths, developers need to replicate these paths one by one for verification, which is not scalable.

Although there are some works that approach chatbot testing during or after implementation, there are no studies approaching chatbot verification at the design phase. In contrast to the machine learning models for chatbots, for which there are several works dealing with testing strategies for them [V. et al., 2020, Liu et al., 2021, Bozic & Wotawa, 2019, Božić, 2022], for the conversational flow itself, there is an unaddressed gap. Even if one is provided a utopian and perfect model for the NLU, if the conversational flow is not optimal, the user gets dissatisfied [Jain et al., 2018, Elmasri & Maeder, 2016].

An alternative approach for such gap is model checking, which is an automated model-based verification that can anticipate design flaws early in the software design phase. It provides a mathematically precise and unambiguous manner to avoid incompleteness, ambiguities, and inconsistencies inherent to software development. The system models are accompanied by algorithms that systematically explore all states of the system model [Baier & Katoen, 2008]. To the best of our knowledge, there is no current approach that explores the benefit of using model checking to assist chatbot developers in finding design flaws in the conversational flow itself prior to implementation.

In this work, we propose a modeling strategy for the design and verification of conversational flow via model checking. In this work, we rely on Uppaal as our model checking tool since it has an extensive list of more than 17 case studies performed in the industry since 1995 [UPPAAL, nd] and has a visual representation of the automata that helps conceptualize conversations. Although Uppaal is used mainly for timed systems, it

¹ <https://rasa.com/>

² <https://dialogflow.cloud.google.com/>

does not hinder the modeling of untimed systems [Li et al., 2020, Koike & Nishizaki, 2013]. Moreover, compared to widely known model checking tool alternatives such as NuSMV [Cimatti et al., 1999] or SPIN [Holzmann, 1997], Uppaal's GUI portrayal and manipulation of the automata are more appealing for novices [Ferrari et al., 2020]. Therefore, for the representation of conversational flows, since we are proposing a modeling strategy to ease the adoption of model checking for chatbot designers, those Uppaal's GUI features together with its (timed-)temporal property specification support and tool maturity given its use both in academia and in industry render Uppaal such a suitable alternative. Furthermore, the visual portrayal of Uppaal resembles the modeling vision of conversational design tools, such as ³BotFlow, which can ease assimilation and correlation.

As such, from an experiment performed with chatbot developers, the benefits pointed of our strategy were many-fold as it: (i) fills the gap of pre-implementation conversational flow verification, which can contribute to reducing rework after implementation; (ii) guarantees more assurance to the developer that the conversational flow is correct and ready to be implemented by performing an exhaustive verification; (iii) allows designers and developers to work together at an intermediate stage of chatbot development not previously possible prior to the chatbot development process; (iv) serves as working documentation and gives an overview of the conversation, especially in frameworks where the flow is described in natural language; (v) may facilitate refactoring flows if conversational requirements change since developers can assess change impacts in the model prior to implementation.

This work is structured under the design science research method, in which professionals in the area share experiences and insights in order to propose a solution to a problem observed in a field of study [Engström et al., 2020]. In this sense, we aim to propose and validate an engineering artifact build upon prescriptive knowledge. To do so, we followed three major steps: 1) proposal and development of the modeling strategy, 2) application of the strategy as a proof of concept, and 3) real-world evaluation of the strategy with chatbot developers.

In the first step, we elicited core elements from both chatbot and Uppaal for a one-to-one mapping, followed by a base model created in Uppaal. In the second step, we used the base model to develop a conversational flow for a hotel booking chatbot as a proof of concept. The model was then mapped into a fully functioning chatbot using the Rasa framework. Finally, in the third step, we conducted an experiment in which three chatbot developers were asked to model a chatbot using our strategy's base model. Then they provided their perceptions of the experience and the strategy through a structured interview. As a result, participants reported their feedback about the experience of following our modeling strategy in Uppaal. Our results show that the learning curve is acceptable and that developers do recognize not only the usefulness but also the gap our strategy fills in the development of chatbots.

The remaining sections are organized as follows: Section 2 presents core background knowledge underlying our strategy, Section 3 details the proposed strategy, Section 4 presents an application of the strategy, Section 5 presents the evaluation and its outcomes, Section 6 discusses the advantages, challenges, and limitations of the strategy, 7 presents other works approaching chatbot testing, and finally, Section 8 concludes this work.

³ <https://botflowapp.com/>

2 Background

In this section, we approach essential concepts to understand the strategy, as well as establish principles and relationships between the concepts of chatbots and model checking.

2.1 Chatbot Architecture

Even though chatbots can be built with different strategies in terms of the Artificial Intelligence (AI) techniques chosen, current AI-chatbots are developed according to a basic generic architecture [Adamopoulou & Moussiades, 2020, Galitsky, 2019], which is shown in Figure 1. There are two modules that are essential for parsing messages and retrieving responses [Adamopoulou & Moussiades, 2020, Galitsky, 2019]: *Natural Language Understanding (NLU)* and *Dialogue Manager (DM)*.

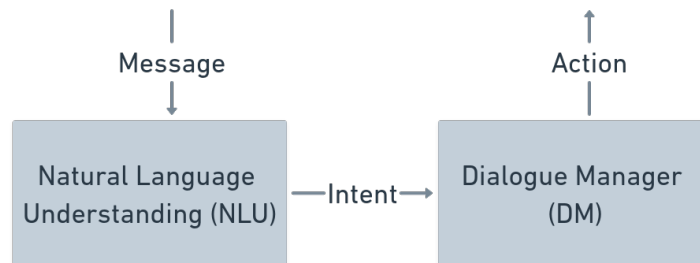


Figure 1: Generic architecture of current chatbots.

The NLU is a module that runs algorithms that aim to classify and understand incoming messages whereas the DM module receives that classification and finds a suitable action for it. This classification is commonly known as an *intent*, which is “a given name, often a verb or a noun, that best describes the user’s intention” [Sant’Anna et al., 2020] that points to a set of sentences in natural language as they would be elaborated by users [Michiels, 2017].

For example, if a given chatbot needs to understand when the user is asking its name, there could be an intent called “asks_name” which is composed of the following training examples: “who is this”, “tell me who you are”, “what is your name”, “what I can call you”, “tell me your name” and many others as needed. The group of intents composes the knowledge base the NLU will work upon to classify incoming messages [Santos et al., 2022].

The DM receives the identified *intent* from the NLU to ascertain the next action, which can be either an answer to the user with a natural language generator, a call to a method, or a call to external services for consultations or accomplishing transactions. Moreover, it “keeps track of information relevant to the dialogue in order to support the process of dialogue management” [McTear, 2020, p. 49].

Making use of the previous example, the DM can impose that the chatbot will answer “My name is Jarvis” whenever it receives the *intent* “ask_name” or, in a more elaborated

chatbot, it could answer “I already told you my name. I’m Jarvis” if it identified that the “ask_name” intent was already triggered earlier in the conversation.

On top of intent classification, the NLU can also identify *entities*, which are important pieces of information inside an intent. For example, if users say “I like music” the NLU could return the “likes_music” intent, but, if they say “I like country music”, the NLU could return an entity “genre” with the value “country” in addition to the “likes_music” intent. Entities are usually stored in *slots*, which are variables that hold information that comes from slots or custom methods. They can be used later in the conversation for branching logic.

Naturally, Figure 1 is a chatbot’s high-level view that does not get into the details about other commonly used components such as databases and input/output channels. Moreover, the NLU and the DM can be accomplished using different techniques and algorithms, even though the end result and its functioning are the same as shown in Figure 1.

2.2 Conversational Flows as State Machines

The method based on finite state machines is a method of rule-based response generation that requires defining all possible conversation states in advance that represent the conversation process [Fan et al., 2020]. Building an entire chatbot as a finite state machine is commonly associated with non-AI chatbots [Teixeira & Dragoni, 2022]. In fact, with the use of *NLU* and *intent* recognition, it is not required to list every possible input sentence and map it to an action. It is only necessary to map the intent to an action since the *NLU* is capable of generalizing new sentences and classifying them as intents.

Similarly, for the DM, most frameworks only require defining pieces of the conversation, leaving the job to the machine learning to dynamically build the conversational flow [Griol & Callejas, 2016]. As a matter of fact, building chatbots only by defining state machines is not scalable [Teixeira & Dragoni, 2022]. However, even with the use of AI, conversational flows still can be seen (albeit not practically built) as state machines, at least in isolated parts of the conversation.

Visualizing conversational flows as state machines is especially useful for conceiving contextual conversations in contrast to FAQ-oriented conversations. In FAQ-oriented chatbots, every intent is mapped to one and only action, that is, for a given intent the response is always the same [Noé-Bienvenu et al., 2020]. On the other hand, context-oriented chatbots can select a different action for the same intent based on the actions and intents that happened previously, that is, the context of the conversation [Galitsky, 2019]. Figure 2 illustrates the difference between FAQ-oriented and contextual conversations.

Currently, chatbots can be built using both strategies, when necessary. For example, a chatbot can use context to deal with “yes” and “no” messages but rely on the rule-based approach to answer to chitchat (i.e. users asking for the chatbot’s name, its age, for a joke). In this sense, it is possible to represent these contextual pieces of conversation as state machines, as seen in [Zamanirad et al., 2020, Fadhil & Gabrielli, 2017].

When first deploying a chatbot, it is essential to have the “happy path” well structured, whereas the “sad paths” can be gradually implemented according to what real conversations show [Santos et al., 2022]. The happy path refers to the ideal conversation where the user gets to their final goal through the shortest path [Flohr et al., 2021]. In this sense, it is viable to model at least contextual happy paths as state machines for verification.

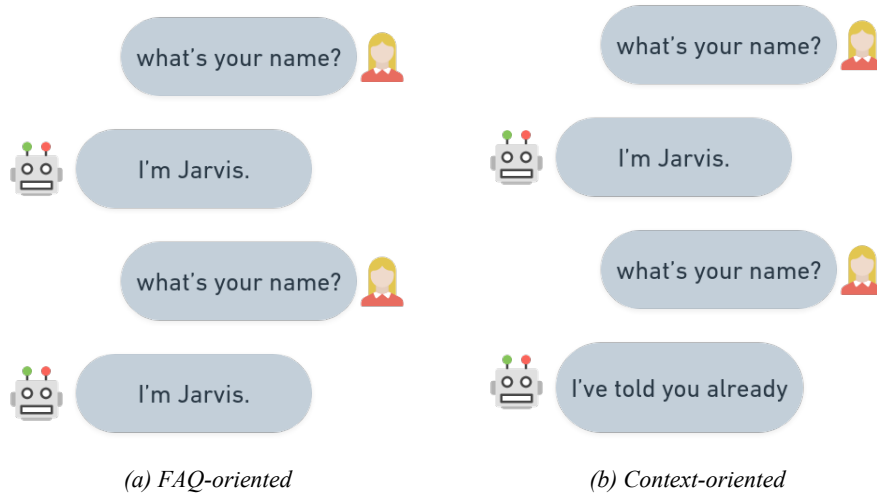


Figure 2: Different dialogue management strategies for handling the same intent.

2.3 Model Checking with Uppaal

Model checking is a formal verification method that “explores all possible system states in a brute-force manner” [Baier & Katoen, 2008, p. 8] and can help verify systems at an early stage of design. The process requires the design of a model as a state machine that dictates how the system behaves and the specification of properties that determine what the system should do. These properties are verified by going through all possible paths of the state machine.

The main advantages of model checking are [Clarke, 2008]: it does not need to construct a correctness proof; the process is automatic, it is faster than other rigorous methods; it produces a counterexample execution trace that shows why the specification does not hold; it is unnecessary to completely specify the program before beginning to check properties. Therefore, it can anticipate the user for design flaws before the design phase is complete.

There are many tools available for conducting model checking, Uppaal being one of them. It has a graphical interface divided into three main parts: the editor, the simulator, and the verifier [Behrmann et al., 2004]. In the editor, systems are modeled as networks of automata inside template files. These networks are composed of locations connected by edges that can execute functions, hold logical conditions, and synchronize with other automata in the system through channels [Behrmann et al., 2011]. The system defined in the editor can be executed in the simulator, which displays the state of the automaton at every step.

Properties in Uppaal are specified in Timed Computational Tree Logic (TCTL) language [Rodrigues et al., 2018], which has its syntax shown in Table 1. As TCTL implies, Uppaal supports verification of timed automata, such as real-time systems. Nevertheless, it can be used for verifying untimed software [Li et al., 2020, Koike & Nishizaki, 2013].

Expression	Semantics
$E\langle\rangle\phi$	there exists a path where ϕ eventually holds
$A[\square]\phi$	for all paths ϕ always holds
$E[\square]\phi$	there exists a path where ϕ always holds
$A\langle\rangle\phi$	for all paths ϕ will eventually hold
$\phi \text{ -- } > \psi$	whenever ϕ holds ψ will eventually hold
$\phi \text{ imply } c \leq T$	ϕ holds if and only if the clock c is within T
$A[\square] \text{ not deadlock}$	checks for deadlocks

Table 1: Basic TCTL expressions in Uppaal.

3 Proposed Strategy

The fact that both chatbot conversational flows and model checking rely on the notion of multiple paths allows the merging of these two fields with the aim of helping conceive and verify chatbot conversations. For this purpose, it is necessary to formalize first how to convert chatbot conversational flows into a Uppaal system.

Table 2 presents essential chatbot conversational elements, as seen in Section 2.1, and how they can be represented in Uppaal. Templates are the automata itself representing an entity, module, or micro-service that is part of a compound system. Locations are graphically presented as circles, connected by edges, representing a given state of the automata. Channels are variables that enable communication among templates and make the automata move from one location to another through the edge when the corresponding variable is triggered. Expressions are logical operations that help define which edge should be taken from multiple possibilities.

The two modules shown in Figure 1 can be represented as templates in addition to the representation of the user since they are entities or services of a chatbot conversation. User messages and intents can be channels that synchronize the templates since they are the elements responsible for changing the state of a conversation.

Actions can be modeled in Uppaal as: a location when it is just an answer to the user, merely representing the new state of the conversation; an update expression with an optional call to a function in analogy to interactions in each the chatbot does some background operation in a method; a combination of committed locations and channels when the action is to communicate with external services to get a response.

Chatbot Element	Uppaal Element
User	Template
NLU	Template
DM	Template
External Services	Template
User Message	Channel
Intent	Channel
Action (Answer)	Location
Action (Method)	Expressions
Action (External Service)	Committed Location and Channel

Table 2: Representing chatbot elements as Uppaal elements.

3.1 Modeling Strategy Templates

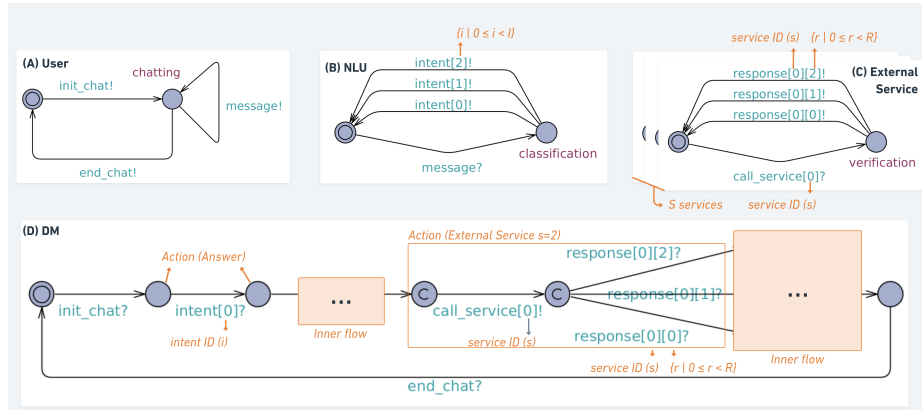


Figure 3: Modeling strategy for verifying a chatbot modeled in Uppaal.

Figure 3 shows a joint vision of the proposed templates for modeling chatbots in Uppaal. Templates (A) *User*, (B) *NLU*, and (D) *DM* are instantiated only once and are mandatory since all of them are indispensable for the model to work, as they are for real chatbots. The (A) *User* initiates the automata, the (B) *NLU* parses intents, and the (D) *DM* represents the conversational flow to be followed according to intents sent by the (B) *NLU*. Template (C) *External Service* can be absent or have more than one instantiation because it represents services for consultation, which are optional elements for maintaining the conversation.

The constants that must be defined according to the domain are the number of intents (I), the number of services (S), and the maximum number of responses from a service (R). Edges for the NLU intents, edges for services responses, and services templates must be replicated according to the constant variables. A functioning version of this system with empty inner flows is available as base model at our GitHub repository in folder *uppaal_base* [Silva, 2022].

The (A) *User* template is responsible for initiating the simulation by triggering the `init_chat` channel and holding the `chatting` state, where the user keeps triggering the `message` channel. Similarly, the user template triggers the `end_chat` to reset the dialogue manager to the initial state.

In real situations, the chatbot could be initiated by sending a message, then the user automaton could start with a `message!` right away. Nevertheless, a separate channel for initiating the chat is more adequate considering the cases where the chatbot is initiated by the act of clicking on the chat and not with a direct message. Therefore, the channel `init_chat` covers both situations.

The `message!` call is received by the NLU template, and then intent `classification` happens. This process receives a natural language input as written by the user and runs machine learning algorithms over the training examples of each intent. Since the knowledge base is composed of intents defined by the developer, it makes sense to include them in the NLU automaton. Therefore, the `classification` location returns to the initial state by calling the `intent` channel with the ID of the chosen intent.

Once the NLU automaton triggers a selected intent, the DM proceeds to the next stage of the conversational flow. The DM automaton always starts with an action to deal with the `init_chat` call. In Figure 3 (D), after the user initiates the chat, a welcoming and informative message is retrieved in the first location. Later in the conversation, if the chatbot needs to access an external service to run calculations, perform a task, or any other elaborated processing, the DM receives the triggering intent and proceeds to a block of locations and transitions representing a call to an external service.

The call starts by holding a committed location (committed locations are identified by the letter “c” inside them), which guarantees that the chatbot will proceed to the next location, according to the response callback, before receiving a new intent call. The transition to the second committed location sends a call through the `call_service` channel with the service ID to start the respective service automaton. There can be two or more paths out of this location, depending on the possible responses sent by the service. These paths lead to another location that represents an answer to the user referring to the results coming from the service.

The DM template must model the dialogue flow of the chatbot being designed, therefore, inner flows may differ, and a call to a service may be absent. Nevertheless, inner flows will replicate the elements shown in the DM, that is, either an action answer or a call to an external service. Moreover, an intent call can pass through intermediary committed locations to process more complex logic to decide the next action location. This logic is evaluated by expressions, which are later explained in Section 3.2.

External services are represented by individual templates that process the call and return a result, as seen in Figure 3(C). Similar to the NLU, templates for external services receive a call and proceed to verification or transaction that can have several response possibilities as well. The service must return to the initial location with a transition triggering a response to the DM.

3.2 Branching Logic with Expressions

Template locations are connected through edges that can be labeled with expressions. There are four types of expressions for edges in Uppaal [Behrmann et al., 2006]: selects, guards, synchronizations, and updates. In our strategy, they are especially useful for branching logic in the DM and external services automata, since sometimes channels are not enough.

Selects help to assign values to local variables, which are declared and assigned directly in the select expression. Therefore, these variables can only be used in the guards or update expressions in the same edge of the select expression in which they were declared. For chatbot automata, they can be used for simulating a value that can be extracted from user intents. For example, if the NLU detects an intent about a user requesting concert seats, the select expression can be used to assign a random value inside a range to a variable holding the number of seats, as in `s: int[0,4]`.

Guards receive variables, logical expressions with variables, or function calls that evaluate to booleans. They are useful for branching paths according to values collected previously in the path. In the chatbot context, guards can be used to change the chatbot response according to values extracted from user intents by evaluating them as logical expressions, such as `var1 > var2 && var3 > var4` or simply checking if a boolean is true or not as in `!bool1`.

Updates are used for changing or assigning variables available globally or attached to the template where the update happens. The operation can happen directly on the update

label or by calling a function. Updates are useful for persisting select variables (e.g. `v: int[0,2]` and `global_var:=v`), incrementing counters (e.g. `counter:=counter+1`) or computing more complex calculations (e.g. `var:=aFunction()`). It is important to notice that sometimes variables are not relevant for the branching logic itself, but are necessary for the verification of a property.

3.3 Verification Properties

Although verification properties are very tied to the model requirements, it is possible to define some basic properties to ensure that the developer is indeed mastering the strategy. In this sense, we attached three basic properties to our modeling strategy that can be used regardless of the chatbot context.

The first property verifies if the system is free of deadlocks ($A[] \text{ not deadlock}$) which fits any Uppaal model. Although very simple, this property alone can detect design flaws that prevent the chatbot from finishing the conversation.

The second property is $A<> \phi$ where ϕ represents the last state before the DM goes back to the initial state again by listening to the `end_chat` channel. This verifies if the chatbot is able to reach the end of the conversation. However, one should be aware that if the model has a loop that may hold infinitely (e.g. user email validation), the property will not be satisfied even though the model is correct.

The third property is $A[] \text{ es.verification imply dm.waiting}$ where *es* is an external service and *waiting* is the second committed location of a call to the *es* in the DM. This property should be replicated for each external service. It verifies if the conversation is always on hold when the chatbot is making a consultation.

3.4 Model Assumptions

First of all, since our automata are untimed, it is necessary to declare channels as urgent to avoid paths that infinitely wait on a location. By default, in untimed models, Uppaal's verifier considers paths that wait infinitely in each location, which prevents the automata from running to the next location. In fact, since our model only progresses if the user sends a message, it is possible that the user leaves the conversation. However, this case is not in our interest for verification, and for checking that the flow's actions are reachable in all paths, we must force the triggering of channels by making them urgent.

Furthermore, real-world chatbots are composed of several intents. In spite of this, for the purpose of testing, it is not necessary to define all the intents in the NLU, but only the ones that are relevant for the DM to work upon. In this sense, the list of intents can be reduced or simplified. For example, if a chatbot has the intents "good_morning" and "good_afternoon", one channel (e.g. "greeting") could be used to represent them if their classification does not result in different paths in the DM automaton.

The DM also requires abstraction. As discussed in Section 2.2, building an entire chatbot as a state machine with all possible paths is not feasible. Therefore, the DM should contain the locations relevant to the contextual conversation and necessary for its verification, leaving FAQ interactions out. Moreover, a chatbot has a lot of entities and slots, but the model should only contain variables indispensable for verification. Intents that influence conversation based on an entity can be transformed into multiple intents that represent the intent alongside an entity value, if necessary. Variables and expressions can help with storing and filling slots in case they influence the conversational flow.

Lastly, it is important to notice that we are treating external services as microservices, that is, each template representing an external service in our Uppaal system should have

a specific purpose. In this sense, if the modeled chatbot has to run more than one external consultation or transaction, this modeling strategy supports as many services as needed, with one template for each.

3.5 Workflow

Chatbots can follow the traditional software development flow, although the resulting artifacts can be very distinctive due to its data-driven and conversational nature. In this sense, the intended use of the modeling strategy is for an intermediary stage between design and implementation. The envisioned workflow is presented below:

1. Model external services: the developer should consult requirements to check if external services are already implemented or are part of the chatbot development. In either case, the modeling of the external service consists of gathering the responses it will return to the chatbot and that will be consumed by the DM in its branches. Given that the base model already provides the necessary locations, it is only necessary to add or remove edges for responses needed.
2. Envision the conversational flow: the developer should start the process by modeling the DM template by imagining step by step how the user starts the conversation, how it ends, the alternative paths, and the external services that should be called. At a first iteration, this flow can be a minimum viable one since transforming the requirements into a conversation can require a bit of trial and error when considering user experience aspects. The base model already provides a generic flow with one example of each action type, which can be reused, refactored, and expanded.
3. Fill and declare dependencies: when the flow and the external services are ready, the NLU should be filled with the intents used in the DM and the variables should be declared according to the declaration file. With everything ready, the developer can check the syntax for missing details until the flow is functional for verification.
4. Develop and verify properties: once the flow is ready, it is time to verify if it attends the requirements. The base model already comes with basic and generic properties that can be verified with no further action. However, developers can further extend this list of properties to verify specific requirements from the chatbot they are building.
5. Refine and refactor: requirements for a chatbot most likely can be met by a number of different conversational flows, meaning that there can be multiple automata that can be considered correct. The developer can build the correct conversational flow, with all properties being satisfied, but maybe the UX analyst considers it disastrous. Therefore, with the minimum viable flow it is necessary to have a multidisciplinary work of refining the flow to meet non-functional requirements as well. Here, the strategy presents its best contribution: the team can work *and* verify the multiple possibilities *before* implementation. Although model checking is complex, the whole team can benefit from the flow vision that Uppaal provides, with only one person being responsible for manipulation and verification.
6. Implementation: with the model done and verified, it is time to transform it into a real chatbot. Since changes in requirements are a reality for the development of any software, the model will still serve as documentation and for helping with the design and simulation of any changes that may come.

4 The Hotel Booking Chatbot

Prior to inviting and conducting the interview with chatbot developers, we modeled, verified, and implemented a fictional chatbot that acts as a hotel representative that books rooms. It served as a proof of concept as well as a supporting artifact for the experiment with chatbot developers, which is presented in Section 5. The representation of the system in Uppaal followed the modeling strategy of Figure 3. The complete chatbot Uppaal system can be seen at our GitHub repository in folder *uppaal_model* [Silva, 2022].

4.1 Requirements

The requirements for this chatbot are presented in Table 3. We limited requirements to those important for creating a minimal conversation context, although a real-world chatbot may have additional requirements. Additionally, consider that the hotel has only one room type, that is, there is no difference among rooms. Consequently, all rooms have a (fictional) standard price per night.

ID	Requirement
R01	The reservation must have the date of check-in.
R02	The reservation must have the date of check-out.
R03	The reservation must have at least one adult guest and four at most.
R04	The reservation can have up to six children.
R05	The reservation must have at least one room to be booked and four at most.
R06	The chatbot must call an external service to check for room availability according to the number of rooms requested.
R07	The chatbot must ask the user if it can proceed in case there is at least the number of rooms requested, otherwise, it should request the user to adjust registration data.
R08	The reservation must have the names and identity numbers of all guests.
R09	The reservation must have the age of all children guests.

Table 3: Requirements for the hotel booking chatbot.

4.2 Model Templates

The representation of the system in Uppaal is composed of all of the mandatory templates of our proposed strategy (i.e., User, NLU, DM) and an additional template *RoomsManager* which represents an external service that checks for room availability. All templates were adapted except the *User* model, which was implemented as presented in Figure 3.

Figure 4 presents the *RoomsManager* template for this hotel booking chatbot. As discussed in Section 3, the service starts a verification and returns a response from two possible. Again, selections were used to assign a random number of available rooms. If `n_available < n_rooms` is true, it means that there are fewer rooms than the number requested by the user, which triggers the `no_vacancy!` call. Otherwise, it triggers the `rooms_ok!` call that makes the DM confirm availability to the user. The `vacancy:=true` statement updates a variable that will be used during the verification process.

The DM automaton is shown in Figure 5. It starts with a linear flow collecting the mandatory information in the following order: check-in date, check-out date, number of adult guests, number of children guests, and number of rooms. These variables were declared and initialized as zero in the global declaration file. Uppaal selections, such

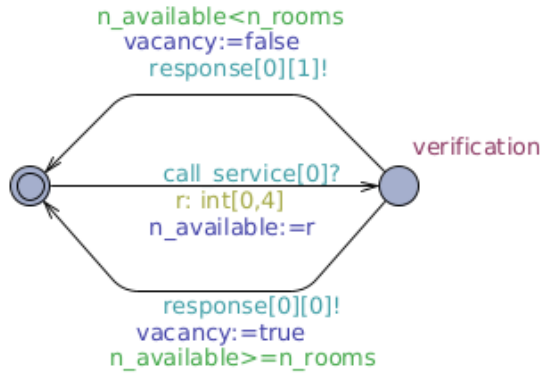


Figure 4: External service example template for a chatbot modeled in Uppaal.

as $a: \text{int}[1, 4]$, were used to assign random numbers to variables that are important in determining future paths.

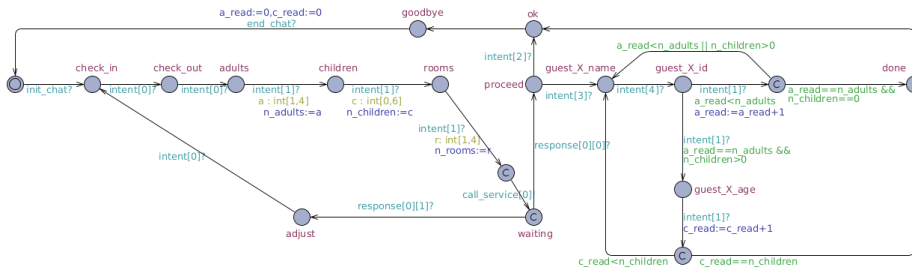


Figure 5: Dialogue Manager for the hotel booking chatbot model in Uppaal.

Variable attribution was used to extract the correct number for adults, children, and rooms (Figure 5). Variables could also be assigned for dates, but since we are not interested in validating dates in our model, there is no point in declaring and filling in these variables. Once these variables are filled, the chatbot calls the external service to check if rooms are available. The `call_service[0]!` call starts the *RoomsManager* automaton. If no rooms are available, the chatbot asks users for new dates. Otherwise, it confirms that rooms are available and asks the user if it can proceed.

Users can opt to give up the reservation, which takes the chatbot to the initial state. If users wish to proceed, the chatbot collects information about all guests. The automaton stays in a loop passing through locations `guest_X_name` and `guest_X_id` to collect all the adults' names and identification numbers.

If there are no children, when all adults were read, the guard `a_read==n_adults && n_children==0` assures that the automaton proceeds to finish the reservation. Conversely, if there are children guests, the guard `a_read<n_adults || n_children>0` takes the automaton back to reading the name again, and the guard `a_read==n_adults && n_children>0` guarantees that the age of each children guest will be collected as well.

4.3 Verification Properties

It is possible to verify if the proposed automata meet requirements in Table 3 by creating properties in the Uppaal's verifier. Table 4 presents the properties created for these automata, which were all satisfied, as seen in Figure 6. Property P01 is vital for any system modeled in Uppaal because a deadlock means that the system arrived at an impasse that prevents the continuity of its execution. For example, in the case of a chatbot, it would mean that the conversation would not be able to follow its predicted flow, and new messages from the user could receive an unwanted or out-of-context response.

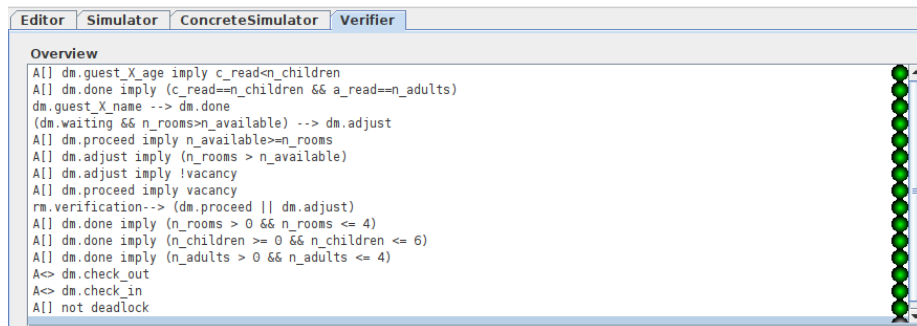


Figure 6: View of Uppaal interface showing Table 4 properties being satisfied.

The other properties are specific to this model since they aim to fulfill the requirements in Table 3. Since some requirements are not very broad in terms of system design details, for some of them, multiple properties verify all the variables related to its design, such as R07. In essence, the more complex the requirement design, the greater the number of properties to verify.

Moreover, verifying properties makes it possible to certify the importance of declaring channels as urgent. This affects properties that start with $A<>$ (P02 and P03) and in the form of $p \text{ --- } > q$ (P07, P12, and P13)⁴. As discussed in Section 3.4, if channels are not urgent, the verification process considers paths that wait infinitely on a location as valid since our model is untimed. Therefore, these properties are not satisfied because, in this situation, there are paths that do not reach the desired locations.

⁴ We should note that the *leads-to operator* $p \text{ --- } > q$ is equivalent to $A[] (p \text{ imply } A<> q)$ [Behrmann et al., 2004]

ID	Uppaal Property	Informal Description	Fulfills
P01	$A[] \text{ not deadlock}$	Checks if the automata is free of deadlocks.	-
P02	$A<> \text{ dm.check_in}$	Eventually, the chatbot will ask for a check-in date.	R01
P03	$A<> \text{ dm.check_out}$	Eventually, the chatbot will ask for a check-out date.	R02
P04	$A[] \text{ dm.done imply (n_adults}>0 \ \&\& \ \text{n_adults}<=4)}$	Always the fact that the chatbot finished the reservation implies that the user informed between 1 and 4 adult guests.	R03
P05	$A[] \text{ dm.done imply (n_children}>=0 \ \&\& \ \text{n_children}<=6)}$	Always the fact that the chatbot finished the reservation implies that the user informed between 0 and 6 children guests.	R04
P06	$A[] \text{ dm.done imply (n_rooms}>0 \ \&\& \ \text{n_rooms}<=4)}$	Always the fact that the chatbot finished the reservation implies that the user informed between 1 and 4 rooms.	R05
P07	$\text{rm.verifying} \ \text{--} \ > \ (\text{dm.proceed} \ \ \text{dm.adjust})$	Whether the rooms manager is verifying for room availability, eventually the chatbot will ask if it can proceed with the reservation or ask the user to adjust the reservation data.	R06, R07
P08	$A[] \text{ dm.proceed imply vacancy}$	Always the fact that the chatbot asked to proceed with the reservation implies that there are rooms available.	R07
P09	$A[] \text{ dm.adjust imply !vacancy}$	Always the fact that the chatbot asked for adjusting reservation data implies that there are no rooms available.	R07
P10	$A[] \text{ dm.adjust imply (n_rooms}>\text{n_available)}$	Always the fact that the chatbot asked for adjusting reservation data implies that the number of requested rooms is less than the number of available rooms.	R07
P11	$A[] \text{ dm.proceed imply n_available}>=n_rooms$	Always the fact that the chatbot asked to proceed with the reservation implies that the number of rooms available is greater or equal than the number of rooms requested.	R07
P12	$(\text{dm.waiting} \ \&\& \ \text{n_rooms}>\text{n_available}) \ \text{--} \ > \ \text{dm.adjust}$	Whether the chatbot is waiting for the rooms manager response and the number of rooms requested is greater than the number of rooms available, eventually the chatbot will ask the user to adjust the reservation data.	R07
P13	$\text{dm.guest_X_name} \ \text{--} \ > \ \text{dm.done}$	Whether the chatbot starts collecting guest names, eventually it will finish the reservation.	R08
P14	$A[] \text{ dm.done imply (c_read==n_children \ \&\& \ \text{a_read==n_adults})}$	Always the fact that the chatbot finished the reservation implies that the number of children and adults read is equal to the number of children and adults requested, respectively.	R08, R09
P15	$A[] \text{ dm.guest_X_age imply c_read}<n_children$	Always the fact that the chatbot is asking for the guest age implies that the number of children read is less than the number of children requested.	R09

Table 4: Properties for model checking the hotel booking chatbot in Uppaal

4.4 Rasa Chatbot

From the model made in Uppaal, we implemented the corresponding chatbot using the Rasa framework. The source code is available on GitHub in folder *rasa_chatbot* [Silva, 2022]. Rasa is an interesting option because it is a mature open-source chatbot framework with a large developer community. At the time of writing this work, its GitHub repository⁵ had more than four thousand forks.

Since our model was abstracted and focused on the happy path, as suggested by Section 3.4, we first implemented the paths foreseen by the model, and from there, we evolved the chatbot to deal with other situations. This helps keep the properties satisfied

⁵ <https://github.com/RasaHQ/rasa>

after translating the model to code and focusing on the primary purpose of the chatbot.

In Rasa, conversational flows can be built with *rules*, *forms*, and *stories*. A rule guarantees that an intent will always be followed by the indicated action in the rule. Forms are as rigid as rules in terms of the sequence of actions and are used for collecting a series of information from the user to fulfill a request. On the other hand, stories are more flexible because they imply machine learning which can handle situations where the user follows a path not foreseen by a story.

Since stories can handle unforeseen behavior, they should be the developer's first option in creating conversational flows. However, sometimes the business logic is complex, then forms or rules are necessary. Having that in mind, considering the path in our DM automaton (Figure 1) in which there are no rooms available and the chatbot asks the user to adjust the information, we derived the story shown in Listing 1. It starts by collecting the check-in date because there is a separate story that deals with *init_chat* and asks the user for the check-in.

```
- story: Ask to adjust reservation
  steps:
  - intent: inform_date
  - action: action_set_check_in      # helper action to save slot
  - action: utter_check_out
  - intent: inform_date
  - action: action_set_check_out    # helper action to save slot
  - action: utter_adults
  - intent: inform_number
  - action: action_set_adults      # helper action to save slot
  - action: utter_children
  - intent: inform_number
  - action: action_set_children    # helper action to save slot
  - action: utter_rooms
  - intent: inform_number
  - action: action_set_rooms
  - action: utter_calling_manager
  - action: action_call_manager
  - slot_was_set:
    - vacancy: false              # the action returned no vacancies
  - action: utter_adjust           # the chatbot asks for new dates
# here we expect that the machine learning will understand that
# the story loops back when inform_date is received again
```

Listing 1: Story representing the path where no rooms are available for booking.

The rationale behind the translation presented in Listing 1 followed Table 2. As we can see in Figure 5, all locations were translated to a Rasa *utter* action, which is a simple text response. In addition, all channels were related to an intent, such as `intent[0]` being - `intent: inform_date` and `intent[1]` being - `intent: inform_date`. Furthermore, the set of committed locations and channels that call the external service were translated into an action that calls python code through the declaration - `action: action_call_manager`. Responses were dealt with by verifying if the corresponding slot was filled, therefore, the two responses modeled in Uppaal correspond to the python code setting the slot (true) or not (false). With the transition from the model done, considering the inner constraints of Rasa, we had to use helper actions to extract exact information from intents, such as dates and numbers.

Even though a form could collect the information requested in Listing 1, the story

can deal better with deviations and with the user informing dates again. Despite this, considering that business logic for collecting guest data is complex, we opted for collecting it with forms and handling the loop with rules. Listing 2 presents these rules. The form must follow actions defined in the domain file, which asks for guests' names and ID numbers. A form customization in Python makes the chatbot ask the age and verifies if all children were read, just as the guards in the automata.

```
- rule: Deactivate guests form
  condition:
  - active_loop: guests_form
  steps:
  - action: guests_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  - action: action_check_guests
  wait_for_user_input: false

- rule: Loop back guests form if there are guests to be read still
  steps:
  - action: action_check_guests
  - slot_was_set:
    - continue_guests: true
  - action: guests_form
  - active_loop: guests_form

- rule: Finish reservation when all guests were read
  steps:
  - action: action_check_guests
  - slot_was_set:
    - continue_guests: false
  - action: utter_done
  - action: action_utter_reservation_info
  - action: action_restart
```

Listing 2: Rules that handle the form that collects guest data.

The - `active_loop: guests_form` is started by a story that starts just like the one in Listing 1, but the action returns - `vacancy: true`, which triggers the activation of the form at the end of the story. The slot `continue_guests` is set by - `action: action_` `check_guests`, which implies all the guards that are seen in the final part of the automata in Figure 5 to determine if there are still guests to be read or the chatbot can finish the reservation by presenting the reservation data with `action_utter_reservation_info` and restarting the conversation by clearing slots with `action_restart`.

Although the transition of the right side of the automata in Figure 5 is not clearly seen in Listing 2, if we dive deeper into Rasa specification files we can see the association that is presented in Table 2. For example, the automata establish the data collection as `guest-X_name`, `guest-X_id`, `guest-X_age`. This subflow can be seen in the form specification, seen in Listing 3. The form's python code (available in our GitHub[Silva, 2022]) is responsible for counting iterations using the same logic the guards apply in the Uppaal model, and the loopback is caught by the story seen in Listing 2.

Until then, we only implemented the paths foreseen by the automata. However, chatbots must be able to deal with chitchat and other questions the user may have about our fictional hotel. For that, we used rules with the ResponseSelector policy, which

```
forms:
  guests_form:
    required_slots:
      - guest_name
      - guest_id
      - guest_age
```

Listing 3: Form specification for collecting the guests' information.

can deviate from stories to respond to FAQ questions and go back to the story after. Similarly, in the form settings, it is configured to answer FAQ questions then return to form questions. These cases can be seen in our test stories, which simulate user interactions and detect failures in the conversational flow.

Since some intents are used in multiple parts of the flow (same intent for check-in and check-out, and same intent for the number of adults, children, and rooms), it is impossible to recover correctly from all deviations, which is expected. However, apart from that, everything works as expected, showing that we could translate the model into a functioning real chatbot and expand the main flow modeled in Uppaal to deal with FAQ questions.

5 Evaluation

The proposed strategy was evaluated in a multistage experiment. Section 5.1 presents the experiment phases and interview settings and Section 5.3 summarizes participants' perceptions about the strategy based on their responses.

5.1 Experiment Settings

The purpose of the experiment is to assess if chatbot developers find model checking with Uppaal helpful in designing conversational flows, if our modeling strategy helped them represent these flows as Uppaal models and if our Hotel Booking chatbot implementation followed its corresponding model. The experiment artifacts can be also found in our GitHub repository in the folder *experiment* [Silva, 2022].

For this evaluation, we invited three chatbot developers with different levels of experience in software and chatbot development, but none of them had previous knowledge of model checking in general. This lack of experience with model checking is essential for evaluating whether the strategy is too complex or the learning curve is acceptable for including the proposed strategy in the chatbot development process. Participants' profiles are summarized in Table 5.

After selecting the potential candidates, invitations were sent via social networks with the purpose and a description of the experiment. Upon acceptance in participating, they received the instructions via email. The experiment had four stages, the first three were asynchronous and the last one was synchronous and accompanied by one of the authors of this work. However, participants could ask for synchronous or asynchronous help at any stage of the process. These stages are detailed below.

1. *Preparation*: participants had to download the UPPAAL tool version 4.0.15 and run it before starting the experiment to check if they had any running problems on their machines.

ID	Gender	Software development experience	Chatbot development knowledge	Rasa framework knowledge
P1	Female	5 years	Basic	Basic
P2	Male	6 years	Intermediate	Intermediate
P3	Male	10 years	Advanced	Advanced

Table 5: Profile of the chatbot developers invited to the experiment.

2. *1st stage (15 minutes)*: participants had to watch a video prepared by the authors that presented basic concepts of model checking, the Uppaal tool, and the proposed strategy as presented in Section 3.
3. *2nd stage (at least 30 minutes)*: participants had to model a chatbot in Uppaal following the modeling strategy of Figure 3. They were given a list of minimum requirements for a chatbot that should book coworking environments. The mandatory instructions were: meet the given requirements, use the modeling strategy, and try to accomplish the stage for at least 30 minutes before giving up in case of extreme difficulty. They were encouraged to refine and go beyond the minimum requirements for a better conversational experience. Moreover, they were given the models of the hotel booking chatbot as a reference for optional consultation.
4. *3rd stage (at least 5 minutes)*: participants had to access and try to understand the Hotel Booking chatbot Uppaal model. Then, they had to inspect the Rasa chatbot source code to check if it matched the Uppaal model. Participants were only asked to assess the Hotel Booking since it captures more complex interactions and would potentially require from the model checking novices more than the given 30 minutes to model in Uppaal.
5. *4th stage (approx. 10 minutes)*: Participants had to participate in a virtual interview conducted by one of the authors in which they were asked about the entire process of the experiment.

The main point of the interview was to ascertain, after the experiment, if developers think that model checking with Uppaal could be inserted in real chatbot development situations and if the modeling strategy helped them to represent the flows since they had no previous knowledge of model checking and Uppaal. We prepared some questions to conduct this semi-structured interview, which started with questions about their profile and then proceeded to question participants about several aspects of the experiment.

5.2 Experiment Conduction

A critical concern of our experiment with the participants was whether the modeling strategy sufficed to abstract away the complexity behind model checking background so that participants would complete the experiment in a reasonable time. As such, they were given the following options: give up before modeling, deliver the incomplete model, and deliver the complete model. In the end, all three were able to follow the experiment until the end. Crucially, they had no contact with each other during the experiment, and the interview was conducted individually and separately with each participant.

Participants willingly asked the accompanying researcher to check their model before the interview. All three were able to finish the model but needed help to use Uppaal's verifier. They were given directions to the verifier's screen to load the properties that come along with the base model, which were presented in Section 3.3.

After running the verification, some properties were not satisfied, but they managed to come back to the model and find their mistake. In the end, all properties were satisfied, except property $A \langle \rangle \phi$ where ϕ since they deliberately added loop verifications in their chatbots, which makes the property not satisfied, as explained in Section 3.3.

In the end, all 3 participants delivered correctly running models. Although the creation of additional properties was optional, P3 was able to create and verify his own property ($A[] \text{dm.confirm_reservation imply end_time} > \text{start_time}$), which guaranteed that if the reservation was being confirmed, the reservation end time was after the start time, a typical verification of reservation systems. Participants also had to confirm that, in their opinion, the Rasa chatbot corresponded to the hotel booking chatbot by looking at the source code. All three agreed that the source code was implementing the proposed model of the hotel booking chatbot.

5.3 Interview Results

Transcript 1 presents the main excerpts from the interview in which the participants report what the experience of modeling the coworking chatbot was like. All participants reported that the experience was generally positive and that the strategy fills gaps in chatbot development that are not currently filled. Among the positive points, participants reported the vision of the flow that the tool provides even before implementation (the automata as seen in Figures 4 and 5) and the automatic verification performed by Uppaal.

Given the positive feedback given by participants regarding the experiment, they also affirmed in different ways that the strategy is useful for chatbot development, especially at the first stages of development, as seen in Transcript 2. Later, when questioned directly about their intention to use the strategy, they also confirmed that they would consider using it in the future. This unanimous confirmation is an indication that the strategy is useful for developers of any level, given that our participants have different experience levels ranging from basic to advanced.

We approached participants' perception of the strategy's complexity by questioning them if they found it difficult to understand the model at the 3rd stage, if they found the Rasa chatbot compatible with our with the hotel booking model, and if they thought it was worth learning this strategy to include it in the development process. As seen in Transcript 3, they all reported some initial difficulty in understanding Uppaal features but that was nothing out of the ordinary of learning something new. For them, the benefits outweigh the difficulties. Moreover, as seen in Transcript 4, they were all capable of considering the model and the chatbot for hotel booking compatible on some level, which shows that this correspondence is minimally noticeable and traceable.

Even though the interview questions were towards the strategy as a whole since participants did not have previous experience with model checking and Uppaal, there was a dedicated question to evaluate if the modeling strategy was adequate and helpful. As seen in Transcript 5, all participants agreed that the model was indeed able to represent a chatbot, although the participant with more experience considered it more of a baseline that needs customization and the participant with less experience thought of it as more of a guide. Again, this shows the usefulness of the strategy for developers with different levels of experience, albeit with slightly different purposes.

P1: “I really like tools that give me a visual aspect of what I am doing. So, in addition to just sticking to the code or even writing it down on paper, a visual representation helps me a lot. So, getting to the end of the process and seeing the DM there, basically, with what I thought, it helped me a lot to see if ... Sometimes, even in the middle, when I was doing it, in the middle of my modeling there, I saw that an interaction between the chatbot and the user was missing, and then I went there and added it. For me it was very beneficial, especially this visualization in the form of an image.”

P2: “[...] for those who already have some knowledge [about chatbot development], I think that, like me, intermediary, some terms in the lexicon here are already known as intent and responses. So for a person who already has some certain experience, it is ok.”

P3: “I think it has a lot of potential because while I was developing it, I found mistakes that I would make and would probably repeat later. [...] And I think you can think and have a preview of what this chatbot will be when complete. [...] It is something that has always been lacking. The design tools that we have to, for example, assemble texts by hand, on paper, or in a tool. They are tools, they are things that are somewhat incomplete because they depend on the subjectivity of someone standing there looking and thinking. There is no way a computer can check everything for me.”

Transcript 1: Participants reporting on the experience of making the Uppaal model for the coworking chatbot.

Notwithstanding the lack of direct questions about Uppaal itself, participants took advantage of the interview space to comment on some negative points that prevented a better experience in the experiment. As seen in Transcript 6, they felt that Uppaal could provide a better user experience in the process. While accompanying some of them that requested some follow-up at the end of the 2nd stage, we noticed their difficulty and slowness in modeling templates due to the lack of shortcuts and correlated features.

6 Discussion

Why model chatbots in Uppaal at the design phase? By implementing the hotel booking chatbot, we saw that the Rasa framework provides multiple features for building conversational flows. If the developer does not choose the right feature for parts of the flow, a lot of time will be spent translating stories into forms or rules, or vice versa. Granted that machine learning helps in reducing the paths that must be declared in development, it is still a laborious work to implement the necessary paths.

As observed by P3, shown in Transcript 1, the first benefit of modeling the chatbot as Uppaal model is to have a full view of how the flow should work, as well as defining the initial branching logic than can be fully utilized at the implementation phase. This broad view of the flow before development saves the developer’s time because modeling

P1: “[...] I think its benefit is great and I even go back to the point I touched before, which is what helped me. This visualization in image helps me a lot. I think it is beneficial, yes, and I think I would use it [...]”

P2: “In the video, the deadlock concept was presented, and there are other more complete concepts that were not presented, but these edge cases are really difficult to be tested in a chatbot. So I believe it is a relatively easy query to do [in Uppaal], which would already eliminate a very complex edge case. So, I believe that, yes, it would be useful to use this tool for these cases.”

P3: “I believe it has a great position at the beginning of the project. So when you are sitting there and seeing what you are going to do, I have a general idea, but I do not know the steps and I do not know how to implement them, how to get started, what would be most useful [...]”

Transcript 2: Participants reporting on whether they think this strategy can be included in chatbot development.

P1: “I think the learning difficulty pays off, even because it is not high. At least I have not felt it in my experience. It did not take me long to learn how to use it. Of course, there is always one detail or another that we can only get through with experience. But even so, once you know that, it is pretty easy.”

P2: “[...] in my opinion, it would be more targeted at developers. It would have a small learning curve, but not too long. So I believe it would help.”

P3: “There is a difficulty, in the beginning, to understand how the tool works and how it annotates things. But this, I believe it would happen in any other type of tool, or anything else, an initial difficulty. Once you get the hang of it and once you understand the symbols and how they are organized, how the state changes from one point to another, I think it is quite practical to look at, take a look, and understand. And the difficulty makes up for it because it was not so complex to start developing with it. I believe it scares you a little at first, but then you get the hang of it. I do not think it is that difficult.”

Transcript 3: Participants reporting on the cost-benefit of using the strategy.

the full flow at Uppaal is faster than implementing it and testing. Moreover, the model helps in visualizing what is the best feature to be used in the implementation phase.

Furthermore, the model checking process done by Uppaal guarantees that the model meets the requirements of the system and is free of deadlocks. As noted by P3 in Transcript 1, current conversational design tools do not provide automatic verification. In this sense, the modeling can aid developers in the process of identifying problematic requirements and reviewing these requirements with stakeholders prior to development. Therefore, our main contribution relies upon the chatbot community, which lacks tools at the design phase for verifying conversational flows in terms of functionality, before implementation.

P1: “[...] looking at the chatbot itself in Rasa, it seemed very good to me and that it meets these requirements that I read, which needed to be met. The [Rasa] flows make a lot of sense to me too.”

P2: “[...] I believe that it is really corresponding, both Rasa and UPPAAL. So they are corresponding.”

P3: “I believe so, I was able to look at the requirements and see parts in the model that matched. Looking through the Rasa files I could see many parts that were also equivalent to the model, which gave me confidence that the resulting chatbot would meet the requirements.”

Transcript 4: Participants reporting on whether they think the Rasa chatbot is in accordance with the hotel booking model.

P1: “Yes definitely. I think it helped not only in the syntax of the thing but how it worked there. [...] So, there is the example of the flow itself, an example of an action there that would exist in the chatbot. And it is all quite understandable. So yes, for me it was. It was great.”

P2: “Yes, a lot. It brings much more basic things, and even some other more complex concepts. For example, calling an external service using another API, making the process wait, and only after this process, this response is processed, proceed. So it is good that you already have this model that brings several concepts of conditions, external requests, so I believe that, yes, It helped a lot.”

P3: “I think it helped. I do not know if in the future for another project I would use the same model. But I know it kick-started it, allowed me to understand the tool. And I think that if I were to use another model, maybe it would even be based on the one that was proposed. [...] So, I found it very useful. But it would probably need to make some evolution for application in other situations.”

Transcript 5: Participants reporting on whether the modeling strategy helped to model the coworking chatbot.

What are the challenges of the proposed strategy? One challenge is the fact that model checking is an extensive field with a strong mathematical background. Even though we presented the required features of Uppaal to model chatbots, it may be still necessary to dive deeper into model checking concepts especially for creating properties for verification.

Furthermore, although we opted for using Rasa in our hotel booking chatbot, our strategy should fit all chatbot frameworks that follow the generic architecture shown in Figure 1. Nevertheless, each framework may have different or additional features for conceiving conversational flows. Therefore, translating the model to the framework is not a task for beginners, and requires great knowledge to implement the chatbot faithfully to the model.

P1: “I did not really like the usability of UPPAAL.”

P2: “An improvement in the interface would be a good thing.”

P3: “One of the difficulties with the tool is that its usability is still not good. I think it is still a bit in a niche, but maybe with evolutions, or with another version or a new program that uses the same concepts, and this can be resolved and turned into a commercial product, for example.”

Transcript 6: Participants suggesting improvements to the tool.

Why has the proposed strategy proven helpful in chatbot development? In our evaluation, we were able to observe quantitative and qualitative metrics regarding the effectiveness of the strategy. Quantitatively speaking, although the 3 participants had no prior knowledge of model checking and were allowed to withdraw from the experiment at any time, all 3 successfully completed the experiment. Still speaking of quantitative evidence, the participants’ models were deadlock-free, indicating that they delivered functional models. Therefore, we had a 100% completion rate of the experiment in a limited time range, indicating that the proposed strategy was able to abstract complex model checking concepts.

From the qualitative view of the accompanying researcher, the models also met the requirements given to the participants. Although participants P1 and P2 kept the model simplistic and closer to the models shared with them, P3 (the most experienced developer) delivered a pretty comprehensive model that used almost all of Uppaal’s modeling features. These models can also be openly and freely consulted in our GitHub repository in the *experiment/models* folder.

Finally, still in a qualitative view, the interviews revealed that all respondents believe that: the strategy addresses a gap in the development of chatbots; the strategy’s learning curve is acceptable; the strategy helped them to foresee errors before they were implemented. Lastly, the only negative point highlighted by the participants was the visual aspect of the tool and the practicality of use.

What are the limitations of the proposed strategy? First of all, the state machine nature of Uppaal modeling limits the size of flows that can be modeled. Thus, it is not ideal for very large flows to follow this strategy because the greater the number of states, the more computational resources will be required to meet brute-force verification of the paths. Furthermore, some templates can get too crowded with edges to be visually understandable. Moreover, if the chatbot requires many actions that execute very complex calculations for branching logic, Uppaal features may be not enough to fulfill these calculations.

Lastly, we cannot guarantee that property satisfaction will hold after the model is translated to a real chatbot, even though some frameworks support testing the flows. Nevertheless, it is unfeasible to reach a 100% coverage as Uppaal by representing all paths by hand, therefore, the integrity of the model when being implemented relies entirely on the capacity of the developer to replicate the model during the chatbot development. Currently, we do not provide yet an automated translation from model to a chatbot. However, we envision this as a future work given the receptivity of developers toward the strategy. In addition, although we cannot guarantee the permanence of full coverage in implementation, our experiment has shown that the use of the strategy brings more

benefits than harm from the participants' perspective.

What are the limitations of this study? The first limitation of the study is that participants did not make the transition from their model to a real chatbot. Indeed this represents a limitation to our results since we could not collect the participants' perceptions on the model-to-chatbot reification process and the evaluation of the results. To address this limitation, an automated translation process followed by a respective evaluation is at stake as forthcoming steps of our research.

Other limitations are threats to validity related to the study being conducted with interviews: 1) sample — we had only three participants in the experiment, which may not be representative and generalizable; 2) reactive effect — the video prepared by the authors that presented basic concepts from the experiment, presented in the first stage, could have induced positive outcomes that would not be seen without the experiment setting and environment; and 3) courtesy bias — there could be lack of sincerity from participants since they could feel an unconscious need to be courteous toward the interviewer. We adopted the following process to mitigate such threats:

- Threat 1 – a conscious effort was part of our action to promote a gender balance of the participants as well as expertise, as depicted in Table 5.
- Threat 2 – given no engaging participants had a background in model checking in Uppaal, the participants needed to first become familiarized with process. Therefore, it is necessary to bear in mind that the results achieved may have been influenced by the complementary and pedagogic material;
- Threat 3 – such threat was mitigated by: (i) having cameras off during the virtual interview so that they would not feel pressured by the interviewer's behavior or direct visual contact; (ii) reinforcing that we were after honest opinions, even if they were unfavorable towards the strategy; (iii) covering both positive and negative aspects of the strategy in interview questions, besides directly addressing possible caveats, such as the learning curve.

7 Related Works

To the best of our knowledge, there are no works approaching automatic chatbot verification at the design phase. Currently, developers do not have a way of automatically testing the design of their conversational flow, as even tools focused on prototyping conversational flows are scarce. Most of the approaches do testing of chatbot conversational flows after their implementation. Related works are either related to general chatbot testing or model-driven chatbot development.

7.1 Chatbot Testing

Bravo et al. [Bravo-Santos et al., 2020] developed a tool called Charm that aims to verify the robustness and precision of the chatbot in identifying the user intents. Charm generates tests by varying chatbot training phrases through fuzzing/mutation functions. Charm was tested with three chatbots developed with DialogFlow and produced tests that revealed faults in them, although it is necessary to manually filter some of these tests. This work focuses on testing NLU performance by automatically generating input sentences and checking if the correct intent was identified and if the conversation state correctly

changed. Although the test also covers the flow of context-dependent interactions as our work, it depends on a chatbot implementation and training utterances, whereas our proposal is for a pre-implementation phase with no need for utterances since we test only the conversational state.

Bozic et al. [Bozic et al., 2019] introduced an automated approach for testing the functional requirements of chatbots. The approach is based on AI planning where each action can be assumed to be a certain question that is given to the chatbot, with the aim of generating communication sequences and carrying them out. Moreover, their work aims to check whether a chatbot will function in a correct way even when encountering unforeseen values and orders of action. Their proposal is demonstrated with a DialogFlow chatbot, and tests revealed when the chatbot would take incorrect actions.

Bozic and Wotawa [Bozic & Wotawa, 2019] introduced a metamorphic testing approach that checks the functionality of a chatbot in an automated manner, in order to guarantee the functional correctness of a chatbot in the absence of expected data. Different from their previous work in [Bozic et al., 2019], which shares the same case study, this work is done through the user perspective and the proposed technique is to change valid user inputs by paraphrasing them. The approach succeeded in triggering defects in chatbots.

Bozic [Božić, 2022] follows the work started in [Bozic et al., 2019] and [Bozic & Wotawa, 2019]. It introduces ontologies as the basis for data conceptualization and test generation, but it keeps the previous goal of functional correctness in the absence of expected data and the same case study for validation. However, the introduction of an ontology model did not detect new defects besides the ones found in their previous works.

The main differences from our work from the three ones previously mentioned [Bozic et al., 2019, Bozic & Wotawa, 2019, Božić, 2022] is that they focus on unpredictable behavior in conversation inputs, depend on a chatbot implementation, and they do not consider conversation context. Although each work varies the rationale behind generating test cases, the test is done upon intent classification, in which it verifies if the automatically generated unpredictable sentence (the input) was classified as the correct intent (the output) whereas our work does not focus on *what* is the *input* sentence, only on *how* the *state* of the conversation changes.

Padmanabhan et al. [Padmanabhan, 2020] presented a methodology to identify the test cases for virtual assistants using a natural language conversations flow diagram. The approach allows testers to generate a set of test cases covering traces derived from the dialog flow. Despite the fact that this process increased the test coverage in comparison with the baseline approach, the authors affirm that it is necessary to further evolve the algorithm for checking deadlocks and increase coverage even more. Although this work is based on querying databases to follow the conversation flow, in terms of testing purpose, it is the closest to ours since it focuses on testing the conversational flow and how its states change. However, it does not guarantee full coverage as ours does.

7.2 Model-Driven Chatbot Development

Pérez-Soler et al. [Pérez-Soler et al., 2020] proposed a model-driven engineering approach for chatbot development. It consists of a meta-model with core primitives for chatbot design, and a domain-specific language (DSL) to define chatbots independently of the implementation technology. The DLS enables forward and reverse engineering for two chatbot frameworks, Rasa and DialogFlow, meaning that it is possible to go from the model to a chatbot and from the chatbot to the model. This work shares one of our motivations, which is to model conversations independently from chatbot frameworks

since new insights can surge in the modeling process, changing what would be the best framework for implementation. Their work is not parallel to ours, but it complements our contribution since they are proposals that can function together.

Planas et al. [Planas et al., 2021] also proposed a DSL for model-driven development of several types of conversational user interfaces. However, they had an additional motivation in comparison with [Pérez-Soler et al., 2020], which was to raise the abstraction level of conversational specification to further encourage the participation of multidisciplinary teams in chatbot teams. The assembling of supporting tools for the inclusion of other experts in chatbot development is also seen in other works [Santos et al., 2022, Khalil et al., 2021]. Khalil et al. [Khalil et al., 2021] developed a tool for flow visualization and manipulation of flows from IBM Watson chatbots, aiming at the inclusion of non-technical domain experts in chatbot flow development. In the same line, one of our motivations for using Uppaal is to provide an intermediary stage in which developers and requirements/UI/UX analysts can work together in the flow before implementation, given its GUI features.

8 Conclusion

In this work, we proposed a strategy for verifying chatbot conversational flows at the design phase through a formal method called model checking with the Uppaal tool. We demonstrated the use of this strategy by designing a hotel booking chatbot with it and later transforming the model into a fully functioning chatbot. Then, we invited three chatbot developers with different levels of experience to experience this approach and provide feedback about it.

Even though model checking is a method with extensive and non-trivial mathematical bases, developers agreed that the method has an acceptable learning curve considering the starting point that the strategy provides and that it has the potential to contribute greatly to the chatbot development process. Moreover, the proposed strategy is compatible and can be included in model driven chatbot development, a process that makes the development platform-independent and contributes to the portability and flexibility of conversational agents [Pérez-Soler et al., 2020, Planas et al., 2021]. Still, developers can use this approach with any chatbot framework as the transition is still up to them regardless of the situation.

According to the observations of developers, the implications of the proposed strategy for chatbot development are multiple. First of all, the development of conversational flows requires that developers have an ability of mentally visualizing the flow while building it since contextual conversations depend on previous interactions. Therefore, the proposed strategy fills that gap of abstracting and specifying the flow before implementation since developers reported a lack of tools for this purpose. Moreover, testing would be only possible after implementation on the framework, which would result in rework over chatbot code if requirements were not well-thought. Considering that developers reported finding design flaws in our experiment with few requirements, it is expected that the proposed strategy can aid the identification of design flaws in projects at an early design stage avoiding rework over code.

The results indicate that the proposed strategy can help in the specification of chatbot conversational flows, refining requirements, and identifying flaws at an early stage preventing rework over chatbot code. However, the manual transition to chatbot code is still a limiting factor, that should be addressed in future work. A thorough study is necessary due to the the multiple frameworks available and the possibility of integrating

it to model-driven tools [Pérez-Soler et al., 2020]. Therefore, since there are multiple possibilities for automatic translation target, the best one should be chosen based on systematical evaluation in addition to developing the adequate translation tool in accordance with the technological apparatus behind Uppaal. Lastly, future works must further evaluate the proposed strategy by conducting experiments in real contexts in addition to requiring participants to transition from the model to chatbot code.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

- [Adamopoulou & Moussiades, 2020] Adamopoulou, E. & Moussiades, L. (2020). An overview of chatbot technology. In I. Maglogiannis, L. Iliadis, & E. Pimenidis (Eds.), *Artificial Intelligence Applications and Innovations* (pp. 373–383). Cham: Springer International Publishing.
- [Baier & Katoen, 2008] Baier, C. & Katoen, J.-P. (2008). *Principles of Model Checking*. Cambridge, Massachusetts: The MIT Press.
- [Behrmann et al., 2004] Behrmann, G., David, A., & Larsen, K. G. (2004). *A Tutorial on Uppaal*, (pp. 200–236). Springer Berlin Heidelberg: Berlin, Heidelberg.
- [Behrmann et al., 2006] Behrmann, G., David, A., & Larsen, K. G. (2006). A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*, 1(1), 1–48.
- [Behrmann et al., 2011] Behrmann, G., David, A., Larsen, K. G., Pettersson, P., & Yi, W. (2011). Developing uppaal over 15 years. *Software: Practice and Experience*, 41(2), 133–142.
- [Bozic et al., 2019] Bozic, J., Tazl, O. A., & Wotawa, F. (2019). Chatbot testing using ai planning. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)* (pp. 37–44). Newark, CA, USA: IEEE.
- [Bozic & Wotawa, 2019] Bozic, J. & Wotawa, F. (2019). Testing chatbots using metamorphic relations. In C. Gaston, N. Kosmatov, & P. Le Gall (Eds.), *Testing Software and Systems* (pp. 41–55). Cham: Springer International Publishing.
- [Božić, 2022] Božić, J. (2022). Ontology-based metamorphic testing for chatbots. *Software Quality Journal*, 30, 227–251.
- [Bravo-Santos et al., 2020] Bravo-Santos, S., Guerra, E., & de Lara, J. (2020). Testing chatbots with charm. In M. J. Shepperd, F. B. e Abreu, A. R. da Silva, & R. Pérez-Castillo (Eds.), *Quality of Information and Communications Technology - 13th International Conference, QUATIC 2020*, volume 1266 of *Communications in Computer and Information Science* (pp. 426–438). Faro, Portugal: Springer.
- [Castle-Green et al., 2020] Castle-Green, T., Reeves, S., Fischer, J. E., & Koleva, B. (2020). Decision trees as sociotechnical objects in chatbot design. In *Proceedings of the 2nd Conference on Conversational User Interfaces, CUI '20* New York, NY, USA: Association for Computing Machinery.
- [Cimatti et al., 1999] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999). Nusmv: A new symbolic model verifier. In *International conference on computer aided verification* (pp. 495–499).: Springer.
- [Clarke, 2008] Clarke, E. M. (2008). *The Birth of Model Checking*, (pp. 1–26). Springer Berlin Heidelberg: Berlin, Heidelberg.

- [Elmasri & Maeder, 2016] Elmasri, D. & Maeder, A. (2016). A conversational agent for an online mental health intervention. In G. A. Ascoli, M. Hawrylycz, H. Ali, D. Khazanchi, & Y. Shi (Eds.), *Brain Informatics and Health* (pp. 243–251). Cham: Springer International Publishing.
- [Engström et al., 2020] Engström, E., Storey, M.-A., Runeson, P., Höst, M., & Baldassarre, M. T. (2020). How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25, 2630–2660.
- [Fadhil & Gabrielli, 2017] Fadhil, A. & Gabrielli, S. (2017). Addressing challenges in promoting healthy lifestyles: The al-chatbot approach. In *Proceedings of the 11th EAI International Conference on Pervasive Computing Technologies for Healthcare, PervasiveHealth '17* (pp. 261–265). New York, NY, USA: Association for Computing Machinery.
- [Fan et al., 2020] Fan, Y., Luo, X., & Lin, P. (2020). A survey of response generation of dialogue systems. *International Journal of Computer and Information Engineering*, 14(12), 461–472.
- [Ferrari et al., 2020] Ferrari, A., Mazzanti, F., Basile, D., Beek, M. H. t., & Fantechi, A. (2020). Comparing formal tools for system design: a judgment study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 62–74).
- [Fiore et al., 2019] Fiore, D., Baldauf, M., & Thiel, C. (2019). "forgot your password again?": acceptance and user experience of a chatbot for in-company IT support. In F. Paternò, G. Jacucci, M. Rohs, & C. Santoro (Eds.), *Proceedings of the 18th International Conference on Mobile and Ubiquitous Multimedia, MUM 2019* (pp. 37:1–37:11). Pisa, Italy: ACM.
- [Flohr et al., 2021] Flohr, L. A., Kalinke, S., Krüger, A., & Wallach, D. P. (2021). Chat or tap? - comparing chatbots with 'classic' graphical user interfaces for mobile interaction with autonomous mobility-on-demand systems. In J. R. Cauchard & M. Serrano (Eds.), *MobileHCI '21: 23rd International Conference on Mobile Human-Computer Interaction* (pp. 21:1–21:13). Toulouse & Virtual Event, France: ACM.
- [Følstad & Brandtzæg, 2017] Følstad, A. & Brandtzæg, P. B. (2017). Chatbots and the new world of hci. *interactions*, 24(4), 38–42.
- [Følstad et al., 2018] Følstad, A., Nordheim, C. B., & Bjørkli, C. A. (2018). What makes users trust a chatbot for customer service? an exploratory interview study. In S. S. Bodrunova (Ed.), *Internet Science - 5th International Conference, INSCI 2018*, volume 11193 of *Lecture Notes in Computer Science* (pp. 194–208). St. Petersburg, Russia: Springer.
- [Galitsky, 2019] Galitsky, B. (2019). *Chatbot Components and Architectures*, (pp. 13–51). Springer International Publishing: Cham.
- [Griol & Callejas, 2016] Griol, D. & Callejas, Z. (2016). A neural network approach to intention modeling for user-adapted conversational agents. *Intell. Neuroscience*, 2016.
- [Holzmann, 1997] Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on software engineering*, 23(5), 279–295.
- [Jain et al., 2018] Jain, M., Kumar, P., Kota, R., & Patel, S. N. (2018). Evaluating and informing the design of chatbots. In *Proceedings of the 2018 Designing Interactive Systems Conference, DIS '18* (pp. 895–906). New York, NY, USA: Association for Computing Machinery.
- [Khalil et al., 2021] Khalil, A., Leiva, F. H., Shonibare, A., Arsenault, E. M., Turner, L., khalifa, S., Tam-Seto, L., Linden, B., Wood, V., Stuart, H., Nolan, J., & McDowell, C. (2021). Model-driven chats: Enabling chatbot development for non-technical domain experts through chat flow visualization and auto-generation. In K. Arai (Ed.), *Advances in Information and Communication* (pp. 1036–1050). Cham: Springer International Publishing.
- [Koike & Nishizaki, 2013] Koike, E. & Nishizaki, S.-Y. (2013). Software analysis of internet bots using a model checker. In *2013 International Conference on Information Science and Cloud Computing Companion (ISCC-C)* (pp. 242–245). Guangzhou, China: IEEE.

- [Li et al., 2020] Li, R., Yin, J., & Zhu, H. (2020). Modeling and analysis of rabbitmq using UPPAAL. In G. Wang, R. K. L. Ko, M. Z. A. Bhuiyan, & Y. Pan (Eds.), *19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2020* (pp. 79–86). Guangzhou, China: IEEE.
- [Liu et al., 2021] Liu, Z., Feng, Y., & Chen, Z. (2021). *DialTest: Automated Testing for Recurrent-Neural-Network-Driven Dialogue Systems*, (pp. 115–126). Association for Computing Machinery: New York, NY, USA.
- [McTear, 2020] McTear, M. (2020). Conversational ai: Dialogue systems, conversational agents, and chatbots. *Synthesis Lectures on Human Language Technologies*, 13(3), 1–251.
- [Michiels, 2017] Michiels, E. (2017). Modelling chatbots with a cognitive system allows for a differentiating user experience. In J. Ralyté, B. Roelens, & S. Demeyer (Eds.), *Proceedings of the Doctoral Consortium and Industry Track Papers presented at the 10th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modelling (PoEM 2017)*, volume 2027 of *CEUR Workshop Proceedings* (pp. 70–78). Leuven, Belgium: CEUR-WS.org.
- [Noé-Bienvenu et al., 2020] Noé-Bienvenu, G. L., Nouvel, D., & Mostefa, D. (2020). Measuring the polarity of conversations between chatbots and humans: A use case in the banking sector. In M. Ganzha, L. A. Maciaszek, & M. Paprzycki (Eds.), *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems, FedCSIS 2020*, volume 21 of *Annals of Computer Science and Information Systems* (pp. 193–198). Sofia, Bulgaria: Polish Information Processing Society.
- [Padmanabhan, 2020] Padmanabhan, M. (2020). Test path identification for virtual assistants based on a chatbot flow specifications. In K. N. Das, J. C. Bansal, K. Deep, A. K. Nagar, P. Pathipooranam, & R. C. Naidu (Eds.), *Soft Computing for Problem Solving* (pp. 913–925). Singapore: Springer Singapore.
- [Pérez-Soler et al., 2020] Pérez-Soler, S., Guerra, E., & de Lara, J. (2020). Model-driven chatbot development. In G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, & H. C. Mayr (Eds.), *Conceptual Modeling* (pp. 207–222). Cham: Springer International Publishing.
- [Planas et al., 2021] Planas, E., Daniel, G., Brambilla, M., & Cabot, J. (2021). Towards a model-driven approach for multiexperience ai-based user interfaces. *Software and Systems Modeling*, 20(4), 997–1009.
- [Rodrigues et al., 2018] Rodrigues, A., Caldas, R. D., Rodrigues, G. N., Vogel, T., & Pelliccione, P. (2018). A learning approach to enhance assurances for real-time self-adaptive systems. In J. Andersson & D. Weyns (Eds.), *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2018* (pp. 206–216). Gothenburg, Sweden: ACM.
- [Sant’Anna et al., 2020] Sant’Anna, D. T., Caus, R. O., dos Santos Ramos, L., Hochgreb, V., & dos Reis, J. C. (2020). Generating knowledge graphs from unstructured texts: Experiences in the e-commerce field for question answering. In M. Koubarakis, H. Alani, G. Antoniou, K. Bontcheva, J. G. Breslin, D. Collarana, E. Demidova, S. Dietze, S. Gottschalk, G. Governatori, A. Hogan, F. Lécué, E. Montiel-Ponsoda, A. N. Ngomo, S. Pinto, M. Saleem, R. Troncy, E. Tsalapati, R. Usbeck, & R. Verborgh (Eds.), *Joint Proceedings of Workshops AI4LEGAL2020, NLIWOD, PROFILES 2020, QuWeDa 2020 and SEMIFORM2020 Colocated with the 19th International Semantic Web Conference (ISWC 2020), Virtual Conference, November, 2020*, volume 2722 of *CEUR Workshop Proceedings* (pp. 56–71).: CEUR-WS.org.
- [Santos et al., 2022] Santos, G. A., de Andrade, G. G., Silva, G. R. S., Duarte, F. C. M., Costa, J. P. J. D., & de Sousa, R. T. (2022). A conversation-driven approach for chatbot management. *IEEE Access*, 10, 8474–8486.
- [Silva, 2022] Silva, G. R. S. (2022). Supplementary material. <https://github.com/GeovanaRamos/hotel-booking-chatbot>.

- [Silva & Canedo, 2022] Silva, G. R. S. & Canedo, E. D. (2022). Requirements engineering challenges and techniques in building chatbots. In A. P. Rocha, L. Steels, & H. J. van den Herik (Eds.), *Proceedings of the 14th International Conference on Agents and Artificial Intelligence, ICAART 2022, Volume 1, Online Streaming, February 3-5, 2022* (pp. 180–187). Virtual Conference: SCITEPRESS.
- [Teixeira & Dragoni, 2022] Teixeira, M. & Dragoni, M. (2022). A review of plan-based approaches for dialogue management. *Cognitive Computation*.
- [Tsai et al., 2021] Tsai, W.-H. S., Lun, D., Carcioppolo, N., & Chuan, C.-H. (2021). Human versus chatbot: Understanding the role of emotion in health marketing communication for vaccines. *Psychology & marketing*, 38(12), 2377–2392.
- [UPPAAL, nd] UPPAAL (n.d.). Case studies and examples. <https://uppaal.org/casestudies/>. Retrieved May 18, 2022.
- [V. et al., 2020] V., V., Cooper, J. B., & J., R. L. (2020). Algorithm inspection for chatbot performance evaluation. *Procedia Computer Science*, 171, 2267–2274. Third International Conference on Computing and Network Communications (CoCoNet'19).
- [Zamanirad et al., 2020] Zamanirad, S., Benatallah, B., Rodriguez, C., Yaghoubzadehfard, M., Bouguelia, S., & Brabra, H. (2020). State machine based human-bot conversation model and services. In S. Dustdar, E. Yu, C. Salinesi, D. Rieu, & V. Pant (Eds.), *Advanced Information Systems Engineering* (pp. 199–214). Cham: Springer International Publishing.