Adapting a Conversational Text Generator for Online Chatbot Messaging

Lorenz Cuno Klopfenstein(✉), Saverio Delpriori, and Alessio Ricci

DiSPeA, University of Urbino, Urbino, Italy
cuno.klopfenstein@uniurb.it

Abstract. Conversational interfaces and chatbots have a long history, but have only recently been hyped as a disruptive technology ready to replace mobile apps and Web sites. Many online messaging platforms have introduced support to third-party chatbots, which can be procedurally programmed, but usually rely on a retrieval-based specification language (such as AIML), natural language processing to detect the user's intent, or on machine learning. In this work we present a work-in-progress integration of a widely-used system for story generation, the *Tracery* grammar, a conversational agent design tool, the *Bottery* system, and online messaging platforms. The proposed system provides a complete and easy-to-use system that allows the creation of chatbots with a graph-based dialogue structure, a contextual memory, pattern-based text matching, and advanced text generation capabilities, that aims for being well-suited for experts and technically unskilled authors alike. Features of the system and future additions are discussed and compared to existing solutions.

Keywords: Conversational interfaces · Chatbots · Story generation · Generative text

1 Introduction

Software applications that engage with users through text-based conversations using natural languages, usually called "chatbots", have been making headlines recently and have captured the interest of major tech companies [17].

Chatbots have a long history, which includes attempts at emulating human language patterns as seen in the conversational mechanisms of ELIZA [25]. These early experiments arguably fueled many of the later tries at playing Turing's "Imitation Game" and to make natural language conversation with a computer possible. ALICE is one of the noteworthy systems developed with this aim: a chatbot based on the AIML language and capable of responding based on pattern matching, which allowed it to achieve the Loebner prize multiple times [24].

While exchanging messages through a teletype console could appear as an interesting contraption in 1966, almost fifty years later the majority of the human

S. S. Bodrunova et al. (Eds.): INSCI 2018 Workshops, LNCS 11551, pp. 87–99, 2019.
https://doi.org/10.1007/978-3-030-17705-8_8

population is accustomed to daily sending and receiving SMS or using online messaging. Top messaging apps nowadays reach a vast audience, rivalling that of the most popular social networks [22]. These highly popular messaging platforms have shown to be the ideal ground for chatbots: over the course of the last years, starting in 2014, many of these platforms have added support to chatbots, in the form of third-party software that can conduct text conversations with their users. Thanks to their quickly rising prominence, chatbots have been announced as the new platform aimed at replacing mobile apps and Web sites [10]. Most messaging platforms also allow chatbots to access features that go beyond simple text exchanges: bots can now exchange messages with pictures, sounds, or geographical positions. Some platforms also include advanced UI elements that can enhance the look or the functionality of chatbot messages, using buttons, quick replies, or special representations for common online transactions [12].

Users show a growing interest in chatbots, not only for small talk, but also as aids for productivity tasks, entertainment, and communication [4]. Successful chatbot applications in real-world usage include customer assistance, conversational commerce, ELIZA-like "chatterbot" conversations [14], virtual assistants, which may exploit contextual information or known personal preferences [1], multiplayer games [13], or emphatic learning companions for kids [26].

Major tech companies have also started focusing heavily on conversational interfaces. Many chatbot middleware systems open to third-party developers are now on offer, including a multi-platform connector with procedural dialogues and NLP capabilities[1], machine learning systems for virtual assistants[2], intent-detection text processors[3], hosting platforms for AIML-based chatbots[4], and solutions with graphical dialogue building and NLP support[5].

Conversational interfaces in literature often rely on pattern matching systems (such as the AIML syntax originally used by ALICE) or different natural language processing (NLP) techniques, which attempt to extract the user's intent from text. Some recent attempts make use of deep learning techniques [11]. Chatbot systems can be categorized into retrieval-based or generative-based (whether they pick predefined responses or generate new ones) and into their application to an open or closed domain [20]. Many chatbots can make use of external linguistic resources or knowledge ontologies, they can extract answers from knowledge bases using information retrieval techniques, or they can be integrated with existing systems in order to provide access to services [21].

However, conversational interfaces often fall short of their user's expectations. Current systems generally have poor social skills, with limited capabilities for the reuse of previous knowledge and context [2]. While most chatbots and virtual assistant promise a natural and human-like interaction, they often fail at clearly revealing their effective capabilities and in correctly processing the full context of a

[1] Microsoft Bot Connector, https://dev.botframework.com.
[2] Dialogflow, backed by Google, https://dialogflow.com.
[3] Wit.ai, https://wit.ai.
[4] Pandorabots, https://pandorabots.com.
[5] IBM Watson Assistant, https://www.ibm.com/watson/services/conversation/.

conversation [16]. A very careful design and user interaction (UX) effort must be made to ensure success, suggesting conversational UX design to be a distinctive and emerging discipline focused on replicating human conversation [19].

1.1 Contribution

Most common messaging platforms provide programming interfaces and software development kits for easy chatbot development. Many middleware solutions allow development of chatbots for a multitude of messaging platforms. However, development platforms either require advanced programming skills or they provide generic chatbot templates that offer limited customization—especially for unskilled users. In this work, this issue is tackled by proposing a flexible and easy-to-use system for creating modern chatbots based on *Bottery*, a prototyping system for generative contextual conversations modelled as finite state machines, which is specifically designed for non-technical users.

In the next section, an operative overview of *Bottery* is provided. A thorough presentation of its design and syntax helps in filling the lack of an adequate documentation of the system in previous literature. A novel system for creating chatbots is proposed, illustrating how it combines the flexibility of *Bottery*-based conversational agents and the UX features of modern messaging platforms. In the last section, advantages and shortcomings of the proposed system are discussed, along with possible future developments and extension to the existing *Bottery* syntax.

2 Conversational Agents with Bottery

Bottery is a conversational agent prototyping platform [7]. Originally released in 2017, it takes inspiration from *Tracery* and provides a syntax, an editor, and an integrated simulator that allows for interactive testing of an agent. To the best of our knowledge, this paper provides *Bottery*'s first documentation in literature.

In the following sections, we will introduce the two systems mentioned above and describe the process of adapting them to be used as an online messaging chatbot. The design choices in transforming conversational agent interactions into a message-based conversation are outlined, together with a proposed software architecture.

2.1 Tracery and Generative Text

Tracery is an easy-to-use and lightweight grammar system that enables users to generate any kind of texts or stories [9]. Syntax and authoring tools were published in 2014 [5]. An online editor is available and allows users to create and test *Tracery* grammars interactively [8].

The system is based on a standard context-free grammar, defined by a set of *rewrite rules* that operate on the input text. Symbols within the text, which are enclosed in '`#`' hashtags, are replaced with a string. Replacement strings can,

in turn, be terminals (strings on which no further rewrite rules can be applied) or they can be composed by more symbols. When multiple replacement strings apply to a symbol, the system picks one at random. Other rewrite rules can perform simple but desirable text transformations, such as word capitalization or switching between "a" and "an" depending on how a symbol is expanded.

Context-free grammars exhibit strong limitations and have been shown to be inadequate to generate complex stories, since they generally lack the ability to maintain contextual history and formal causality [3], even though *Tracery* includes the ability to keep a limited form of context while processing rewrite rules [5]. These limitations notwithstanding, the system has been successfully adopted in games and more elaborate story generation tools [6,15,23].

A *Tracery* grammar is expressed in JSON format as a simple list of rewrite rules that associate symbols with one or more expanded strings.

```
{
  "alienName": [ "Jaglan Beta", "Santraginus V", "Kakrafoon", "Traal" ],
  "action": [
    "wrap it around you for warmth as you bound across #alienName#",
    "lie on it on the brilliant marble-sanded beaches of #alienName#",
    "sleep under it beneath the stars which shine so redly on #alienName#",
    "use it to sail a raft down the slow heavy river #alienName#"
  ],
  "origin": [ "The towel has great practical value, you can #action#; #action
      #; and of course #action#." ]
}
```

This grammar can, for instance, generate the following text:

```
The towel has great practical value, you can sleep under it beneath the stars which shine so
 redly on Santraginus V; use it to sail a mini raft down the slow heavy river Traal; and of
           course lie on it on the brilliant marble-sanded beaches of Kakrafoon.
```

2.2 Bottery Agent Syntax

The *Bottery* syntax allows authors to create an agent by specifying 4 components:

(a) A set of **states** in which the agent can be, with information that describes the conditions in which the agent moves from one state to another;
(b) A **memory** that can be read and manipulated freely, giving a "blackboard"-style representation of the agent's state for dialogue and text generation [18];
(c) An optional set of global **transitions**, that allow the agent to switch to a given state in response to certain conditions;
(d) An optional *Tracery*-based **grammar**. Text produced by the agent is used as an input string for text generation through the grammar. The expanded text is used as the agent's final output for the user.

Agent States. States describe the agent's position within the dialogue, as defined by the script. Each state is uniquely identified by an identifier. A state contains instructions about the agent's behavior (its output) and its reactions to user input.

In particular, states of the agent are composed of the following:

(a) An alphanumeric **ID**, unique within the same chatbot script (the initial state must have the ID "`origin`" by convention);
(b) A list of **actions** that are executed when the state is entered by the bot;
(c) A list of **exits** that describe transitions to another state and the conditions they require;
(d) A set of **suggestion chips**: suggested text inputs that are provided by the agent.

Actions. On entering a state, the agent will perform a set of actions. These can be expressed in a variety of ways. For instance, "`onEnter`" actions are always performed when the state is entered. On the contrary, "`onEnterDoOne`" actions are prefixed by conditions: the agent will only perform the first action whose condition is satisfied, if any. Actions expressed as "`onEnterSay`" provide simple text strings that are always output by the agent. Additionally, agents may also play audio files through "`onEnterPlay`" actions or execute arbitrary Javascript functions through "`onEnterFxn`".

Actions executed by "`onEnter`" and "`onEnterDoOne`" are expressed using *action syntax*: a space-delimited sequence of expressions that are executed in order. Expressions can rely on a very limited set of operations using a Javascript-inspired syntax, that allows to operate on variables in the agent's *blackboard* (e.g., "`counter++`" increments a variable). String expressions or constants are evaluated using the *Tracery* grammar and output by the agent.

Conditions for "`onEnterDoOne`" actions can also be written using a limited syntax, which includes basic arithmetic and logical operators, that accesses variables in the *blackboard* (e.g., "`counter>4`"). Simple strings are directly matched against the last user input. The asterisk character '`*`' matches any user input. When no condition is given, the condition is always verified.

Exits. All states have one or more exits that describe possible transitions of the agent to other states. Exists are expressed following this syntax:

"`[condition ...] -> target [action ...]`"

Conditions and actions are expressed using the same syntax as above. The target state is referenced by its ID. The '`@`' character can be used to reference the agent's current state (i.e., the agent re-enters the same state).

Executing an Agent. An agent based on this system acts like a finite state machine (FSM). It can be displayed as a graph, with states connected by arcs, each of which marked with a condition that must be satisfied to transition the agent from one state to another.

On entering a state, the agent will: (1) execute all available *actions*; (2) display *suggestion chips* to the user, if any; (3) wait for user input; (4) evaluate *exits* and pick the first that satisfies its condition; (5) execute *actions* associated with the picked *exit*; (6) move to the target state; repeat.

2.3 Adapting Bottery to Online Messaging Platforms

The basic mechanism animating the FSM-based execution flow of a *Bottery* agent can be adapted to the reactive nature of online messaging: (1) receive new user input; (2) evaluate *exits* on the current state (including global *exits*) and pick the first whose conditions are satisfied; (3) execute all *actions* associated with the *exit*; (4) move agent to the target state; (5) execute all *actions* of the new state; (6) display *suggestions chips*, if any.

The direct interface between the target messaging platform and the conversational agent simulation is limited to sending and receiving text messages. When receiving a text message, the conversational agent will handle the input by performing a state change, executing actions, and/or outputting any number of text responses, before returning to a waiting state.

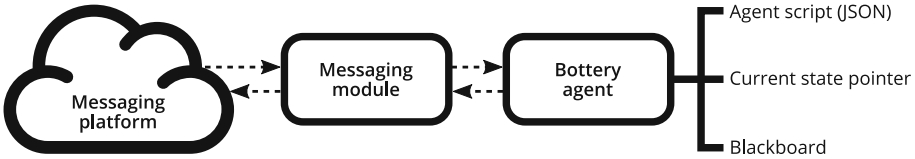

Fig. 1. Software architecture of the proposed system.

The proposed system's architecture is shown in Fig. 1: a messaging module (a specialized software module that interacts with an online messaging platform) receives messages from users and forwards them to the *Bottery* engine. Any output generated by the agent is returned back to the user in the form of text messages.

While the system in development only includes a messaging module for the Telegram platform (chosen for ease of development), any number of messaging modules can be added, making the system multi-platform.

The *Bottery* agent is composed by: an agent script (specified in JSON using the syntax described in Sect. 2.2), a pointer to the agent's current state, a *blackboard* memory (as a map of variable names and values).

In order to keep the system's state for multiple concurrent users, the system will store the current state pointer and the blackboard for each conversation (e.g., in a database system, identifying each conversation with a unique chat ID), while the agent's scripts can be stored once in static JSON files. On receiving a new message the system will load the agent's script and restore the conversation context, by moving the agent to the last state and loading its blackboard into memory.

Figure 2 describes a simple chatbot providing access to weather information. The chatbot offers a guided conversation allowing users to pick a city and a date for which the forecast must be fetched. The conversation is structured over a set of 7 states, depicted as a graph on the right. The forecast generation makes use of *Tracery* to generate random weather information and of an array to keep a

```
{
  "grammar": {
    "forecast": ["fair", "a little cloudy", "cloudy", "rain", "storm"]
  },
  "states": {
    "origin": {
      "onEnterSay": "Welcome to WeatherBot",
      "exits": "->choose_city"
    },
    "choose_city": {
      "onEnterSay": "Enter the city for which you want to know the weather",
      "exits": "'*' ->choose_day city=INPUT"
    },
    "choose_day": {
      "onEnterSay": "Choose the day for which you want to know the weather",
      "chips": ["today", "tomorrow", "day after tomorrow", "change city"],
      "exits": [
        "'today' ->search_forecast day=INPUT",
        "'tomorrow' ->search_forecast day=INPUT",
        "'day after tomorrow' ->search_forecast day=INPUT",
        "'change city' ->choose_city",
        "'*' ->error_choose"
      ]
    },
    "error_choose": {
      "onEnterSay": "Sorry, I don't understand",
      "exits": "->choose_day"
    },
    "search_forecast": {
      "exits": ["(list_city[city][day]!=undefined) ->send_forecast",
                "->ask_forecast"]
    },
    "ask_forecast": {
      "onEnter": "list_city[city][day]='#forecast#'",
      "exits": "->send_forecast"
    },
    "send_forecast": {
      "onEnter": "weather=list_city[city][day]",
      "onEnterSay": ["#/day# in #/city# the weather will be #/weather#"],
      "exits": "->choose_day"
    }
  }
}
```

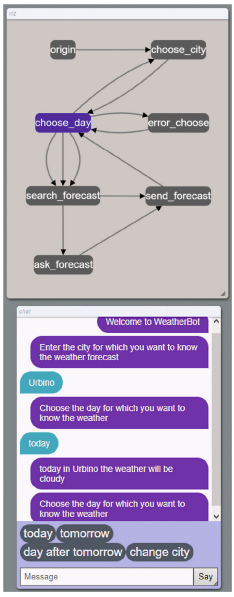

Fig. 2. Simple weather chatbot implemented using *Bottery* (left), states graph (top right) and example conversation (bottom right).

memory of past queries. An actual implementation of the chatbot could of course retrieve real weather information within the `ask_forecast` state by calling an arbitrarily complex Javascript function.

2.4 Advanced Messaging Features

Most current online messaging platforms are not limited to the exchange of simple text messages, but also allow users and bots to exchange pictures, audio messages, and locations. Chatbots in particular have the ability to use advanced UI elements, such as buttons, commands, and structures messages on some platforms (e.g., Telegram and Facebook Messenger) [12]. The basic *Bottery*-based system described previously provides some extension points that would allow chatbot authors to make use of these features.

Audio Output: actions marked as "`onEnterPlay`" instruct the chatbot to play a given audio file. On supported platforms (such as Telegram) the audio file can be sent and received by the user as a voice note.

Suggestion Chips: one of the optional components of a *Bottery* state. They take the form of one or more strings that are shown to users as suggestions while waiting for the next input. When a suggestion chip is clicked, the chatbot

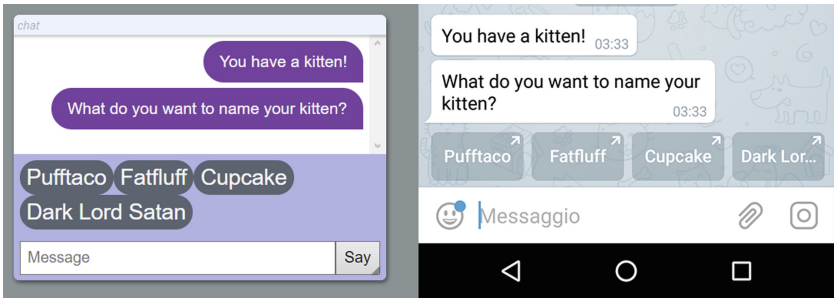

Fig. 3. Comparison of "suggestion chips" in *Bottery* (left) and the equivalent "inline keyboard" in a Telegram bot conversation (right).

behaves just as if the user sent its content as a message. Chips perfectly map to "quick replies" (on Facebook Messenger) or an "inline keyboard" (on Telegram, shown in Fig. 3), both features providing a list of buttons that can be tapped to provide a preset reply to the chatbot.

Global Exits: just like normal *exits* from a state, they are defined providing a condition, a target state, and optional actions to execute. However, while local exits are evaluated only on the state where they are defined, global ones are evaluated every time the chatbot receives input. This provides a straightforward opportunity to provide support for top-level always-on interface elements, such as "commands" in Telegram (messages that start with '`/`' and are shown in the bot's UI) or "persistent menus" in Facebook Messenger.

3 Discussion

Bottery is a complete conversational agent system that provides a full-featured editing and simulation environment. In its adaptation for online messaging, it is well suited to implement chatbots with a structured conversation flow—given its internal representation as an FSM—while retaining the capability to generate rich and varied responses, thanks to a grammar-based text generator, and the ability to keep track of the conversation's context.

3.1 Future Work

The proposed system is currently in development and is available under an open source license on GitHub[6]. While the development version of the system supports Telegram as a messaging platform, integration with additional platforms is highly desirable. These additions would also provide additional opportunities for making use of advanced messaging platform features, outlined in Sect. 2.4.

[6] Official repository: https://github.com/ComputerScienceUniUrb/messaging-bottery.

```
{
  "grammar": {
    "forecast": ["fair", "a little cloudy", "cloudy", "rain", "storm"],
    "ask_preamble": ["what's the", "how's the"],
    "ask_weather": ["weather in", "weather like in"],
    "ask": ["#ask_preamble# #ask_weather#", "#ask_weather#"],
    "conjunction": ["and", ""]
  },
  "states": {
    "origin": {
      "onEnterSay": "Welcome to WeatherBot"
    },
    "forecast_today": {
      "onEnterSay": "Today in #/city# the weather will be #forecast#"
    },
    "forecast_tomorrow": {
      "onEnterSay": "Tomorrow in #/city# the weather will be #forecast#"
    }
  },
  "exits": [
    "'#ask# *?' ->forecast_today city=STAR",
    "'#ask# * today?' ->forecast_today city=STAR",
    "'#ask# * tomorrow?' ->forecast_tomorrow city=STAR",
    "'#conjunction# today?' ->forecast_tomorrow",
    "'#conjunction# tomorrow?' ->forecast_tomorrow",
  ],
  "initialBlackboard": {
    "city": "Urbino"
  }
}
```

Fig. 4. Proposed weather chatbot implementation using extensions to *Bottery* for enhanced pattern matching.

The *Bottery* grammar could also be extended in order to introduce support for pictures, locations, and other data types that are not contemplated by the original syntax.

Pattern matching capabilities provided by *Bottery* are very limited, being constrained to using the '*' character to match the whole user input. Future versions of the system will introduce AIML-like pattern matching, where the asterisk can match a sequence of any length. The matched sequence will be available to the agent as a variable for further processing.

Similarly, pattern matching will be further extended to exploit the agent's generative text capabilities: user input will positively match if it can be generated by any symbol expansion in the pattern, according to a given *Tracery* grammar. The ability of using *Tracery* to easily express variations of the same concept (including typos) would allow even unskilled chatbot authors to flexibly match user inputs.

Both these enhancements to pattern matching are shown in Fig. 4, describing a possible weather chatbot implementation similar to the one in Fig. 2. In this example user input is processed using global exits, using both partial '*' and *Tracery* expansions matching, thus providing a very concise implementation and a more natural conversation for users.

The introduction of a more capable pattern matching mechanism would make the system potentially equivalent to AIML-based interpreters. Efforts will be made to formally map the capabilities of the proposed system to AIML constructs, with the purpose of providing a way of automatically transforming an AIML-based bot into a *Bottery* agent.

Finally, while the transactions between states provided by *Bottery* allow the agent to freely jump from one branch of dialogue to another, keeping track

of the switch is left to the script author. The system can be extended to allow "context digressions" that maintain a stack-based context of the conversation, in a fashion similar to function calls in a programming language. This would make it easier for authors to keep track of dialogue state and to structure conversations into bite-sized sections, for instance providing procedures that handle specific sub-questions or general conversations for when the chatbot needs to ask for clarification [1].

3.2 Conclusions

Online messaging chatbots based on the proposed system have several significant capabilities. They can: (a) easily define a structured dialogue tree, using states connected by exits; (b) perform question-answer dialogues using a pattern matching system based on categories (correspondence between user input and a string), in particular if extended with capabilities described in Sect. 3.1; (c) rely on a general-purpose memory that can be used to guide conversation, provide context, or drive more advanced "business" logic; (d) make use of a *Tracery* grammar for text generation, either to make chatbot output more variegated or for more advanced story generation; (e) make use of logical and arithmetic conditions; (f) call any procedural function, using the power of a Turing complete language like Javascript with access to the chatbot's memory.

Tracery's text generation features can be used in this context to provide distinctive and various output with minimal programming expertise. The system's elementary text generation capabilities gain much higher potential thanks to the interaction with *Bottery*'s states graph and the adoption of a general-purpose memory. Both measures allow the agent to maintain a detailed knowledge of the state of the conversation, which can easily be used to model more complex agent behavior and more appropriate text generation.

While the proposed system places itself firmly among other retrieval-based closed-domain chatbot systems, such as those based on AIML, there are some noteworthy improvements that *Bottery* provides. According to the categories established by Augello et al., the use of tree-structured dialogues provides facilities to support context management at practice level: independent branches of conversation can be assigned to a specific situation and can be split up into scenes, thus granting a much more powerful alternative to AIML "topics". At a functional level, while AIML and ChatScript offer the "that" tag or "rejoinder" rules to manage small one-level conversation trees, the same functionality can be replicated using a dialogue branch [2]. Structured memory, that can be accessed by the chatbot's logic, offer a powerful alternative to the "get" and "set" tags in AIML. On the other hand, while AIML offers the ability to add new rules (through the "learn" tag), the proposed system offers no such option to dynamically extend the chatbot's intelligence.

The overall syntax used to define a chatbot is mostly declarative and can be expressed in simple, structured JSON. The option to make use of imperative Javascript code provides a powerful extension point, which can be also exploited

to provide service integration or access to external knowledge bases [21]. However, it weakens the declarative nature of the chatbot's JSON-only script.

A graph-based chatbot script provides several options for visualization, which allow users to more easily explore and edit the flow of a conversation. The system's ease of use—especially if compared with editing complex XML files with pattern matching strings or unfathomable NLP rules—harks back to *Tracery*'s long history of making advanced text generation tools available to technically unskilled users [5].

References

1. Angara, P., et al.: Foodie fooderson a conversational agent for the smart kitchen. In: Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, pp. 247–253 (2017). http://dl.acm.org/citation.cfm?id=3172795.3172825
2. Augello, A., Gentile, M., Dignum, F.: An overview of open-source chatbots social skills. In: Diplaris, S., Satsiou, A., Følstad, A., Vafopoulos, M., Vilarinho, T. (eds.) INSCI 2017. LNCS, vol. 10750, pp. 236–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77547-0_18
3. Black, J.B., Wilensky, R.: An evaluation of story grammars. Cogn. Sci. **3**(3), 213–229 (1979). https://doi.org/10.1016/S0364-0213(79)80007-5
4. Brandtzaeg, P.B., Følstad, A.: Why people use chatbots. In: Kompatsiaris, I., et al. (eds.) INSCI 2017. LNCS, vol. 10673, pp. 377–392. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70284-1_30
5. Compton, K., Filstrup, B., Mateas, M.: Tracery: approachable story grammar authoring for casual users. In: Proceedings of the AIIDE Workshop, Intelligent Narrative Technologies 2014, pp. 64–67 (2014). https://www.aaai.org/ocs/index.php/INT/INT7/paper/view/9266
6. Compton, K., Kybartas, B., Mateas, M.: Tracery: an author-focused generative text tool. In: Schoenau-Fog, H., Bruni, L.E., Louchart, S., Baceviciute, S. (eds.) ICIDS 2015. LNCS, vol. 9445, pp. 154–161. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27036-4_14
7. Compton, K., Leigh, N., et al.: "Bottery" official source code repository. https://github.com/google/bottery. Accessed 23 Aug 2018
8. Compton, K., et al.: Online Tracery editor. http://brightspiral.com/tracery/. Accessed 23 Aug 2018
9. Compton, K., et al.: "Tracery" official source code repository. https://github.com/galaxykate/tracery. Accessed 13 Sept 2018
10. Dale, R.: The return of the chatbots. Nat. Lang. Eng. **22**(5), 811–817 (2016). https://doi.org/10.1017/S1351324916000243
11. Kamphaug, Å., Granmo, O.-C., Goodwin, M., Zadorozhny, V.I.: Towards open domain chatbots—a GRU architecture for data driven conversations. In: Diplaris, S., Satsiou, A., Følstad, A., Vafopoulos, M., Vilarinho, T. (eds.) INSCI 2017. LNCS, vol. 10750, pp. 213–222. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77547-0_16
12. Klopfenstein, L.C., Delpriori, S., Malatini, S., Bogliolo, A.: The rise of bots: a survey of conversational interfaces, patterns, and paradigms. In: Proceedings of the 2017 ACM Conference on Designing Interactive Systems, DIS, pp. 555–565 (2017). https://doi.org/10.1145/3064663.3064672

13. Klopfenstein, L.C., Delpriori, S., Paolini, B.D., Bogliolo, A.: Code hunting games: a mixed reality multiplayer treasure hunt through a conversational interface. In: Diplaris, S., Satsiou, A., Følstad, A., Vafopoulos, M., Vilarinho, T. (eds.) INSCI 2017. LNCS, vol. 10750, pp. 189–200. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77547-0_14
14. Kuboň, D., Metheniti, E., Hladká, B.: Politician – an imitation game. In: Diplaris, S., Satsiou, A., Følstad, A., Vafopoulos, M., Vilarinho, T. (eds.) INSCI 2017. LNCS, vol. 10750, pp. 201–212. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77547-0_15
15. Kybartas, B., Verbrugge, C., Lessard, J.: Subject and subjectivity: a conversational game using possible worlds. In: Nunes, N., Oakley, I., Nisi, V. (eds.) ICIDS 2017. LNCS, vol. 10690, pp. 332–335. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71027-3_37
16. Luger, E., Sellen, A.: "Like having a really bad PA": the gulf between user expectation and experience of conversational agents. In: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, CHI 2016, pp. 5286–5297 (2016). https://doi.org/10.1145/2858036.2858288
17. McTear, M.F.: The rise of the conversational interface: a new kid on the block? In: Quesada, J.F., Martín Mateos, F.J., López-Soto, T. (eds.) FETLT 2016. LNCS (LNAI), vol. 10341, pp. 38–49. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69365-1_3
18. Montfort, N., Pérez y Pérez, R., Harrell, D.F., Campana, A.: Slant: a blackboard system to generate plot, figuration, and narrative discourse aspects of stories. In: Proceedings of the Fourth International Conference on Computational Creativity, ICCC 2013, p. 168 (2013). http://www.computationalcreativity.net/iccc2013/download/iccc2013-montfort-et-al.pdf
19. Moore, R.J., Arar, R., Ren, G.J., Szymanski, M.H.: Conversational UX design. In: Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems, CHI EA 2017, pp. 492–497 (2017). https://doi.org/10.1145/3027063.3027077
20. Ramesh, K., Ravishankaran, S., Joshi, A., Chandrasekaran, K.: A survey of design techniques for conversational agents. In: Kaushik, S., Gupta, D., Kharb, L., Chahal, D. (eds.) ICICCT 2017. CCIS, vol. 750, pp. 336–350. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-6544-6_31
21. Satu, M.S., Parvez, M.H., Shamim-Al-Mamun: Review of integrated applications with AIML based chatbot. In: 1st International Conference on Computer and Information Engineering, ICCIE 2015, pp. 87–90 (2016). https://doi.org/10.1109/CCIE.2015.7399324
22. Smith, J.: The messaging apps report. Technical report (2018). https://www.businessinsider.com/messaging-apps-report-2018-4
23. Veale, T.: Appointment in samarra: pre-destination and bi-camerality in lightweight story-telling systems. In: Proceedings of the Ninth International Conference on Computational Creativity, ICCC 2018, pp. 128–135 (2018). http://computationalcreativity.net/iccc2018/sites/default/files/papers/ICCC_2018_paper_35.pdf
24. Wallace, R.S.: The anatomy of A.L.I.C.E. In: Epstein, R. (ed.) Parsing the Turing Test: Philosophical and Methodological Issues in the Quest for the Thinking Computer, pp. 181–210. Springer, Netherlands (2009). https://doi.org/10.1007/978-1-4020-6710-5_13

25. Weizenbaum, J.: ELIZA—a computer program for the study of natural language communication between man and machine. Commun. ACM **9**(1), 36–45 (1966). https://doi.org/10.5100/jje.2.3_1
26. Wiggins, J., Mott, B., Pezzullo, L., Wiebe, E., Boyer, K., Lester, J.: Conversational UX design for kids: toward learning companions. In: Proceedings of the Conversational UX Design CHI 2017 Workshop (2017). http://researcher.watson.ibm.com/researcher/files/us-rjmoore/Wiggins.pdf