IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 28, 2019, accepted January 9, 2020, date of publication January 15, 2020, date of current version January 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2966919

Xatkit: A Multimodal Low-Code Chatbot Development Framework

GWENDAL DANIEL[1], JORDI CABOT[2], (Member, IEEE), LAURENT DERUELLE[3], AND MUSTAPHA DERRAS[3]

[1]IN3, UOC, 08060 Castelldefels, Spain
[2]ICREA, 08010 Barcelona, Spain
[3]Berger-Levrault, 54250 Champigneulles, France

Corresponding author: Gwendal Daniel (gdaniel@uoc.edu)

This work was supported in part by the Spanish Government under Project TIN2016-75944-R, and in part by the Electronic Component Systems for European Leadership Joint Undertaking under Grant 737494.

ABSTRACT Chatbot (and voicebot) applications are increasingly adopted in various domains such as e-commerce or customer services as a direct communication channel between companies and end-users. Multiple frameworks have been developed to ease their definition and deployment. While these frameworks are efficient to design simple chatbot applications, they still require advanced technical knowledge to define complex interactions and are difficult to evolve along with the company needs (e.g. it is typically impossible to change the NL engine provider). In addition, the deployment of a chatbot application usually requires a deep understanding of the targeted platforms, especially back-end connections, increasing the development and maintenance costs. In this paper, we introduce the Xatkit framework. Xatkit tackles these issues by providing a set of Domain Specific Languages to define chatbots (and voicebots and bots in general) in a platform-independent way. Xatkit also comes with a runtime engine that automatically deploys the chatbot application and manages the defined conversation logic over the platforms of choice. Xatkit's modular architecture facilitates the separate evolution of any of its components. Xatkit is open source and fully available online.

INDEX TERMS Modeling, DSL, chatbot design, chatbot deployment.

I. INTRODUCTION

Instant messaging platforms have been widely adopted as one of the main technologies to communicate and exchange information [1], [2]. Nowadays, most of them provide built-in support for integrating *chatbot applications*, which are automated conversational agents capable of interacting with users of the platform [3]. Chatbots have proven useful in various contexts to automate tasks and improve the user experience, such as automated customer services [4], education [5], and e-commerce [6]. Moreover, existing reports highlight the large-scale usage of chatbots in social media [7], and emphasize that chatbot design will become a key ability in IT hires in the near future [8]. Additional predictions say that by 2022, 80% of the companies will use chatbots and banks will be able to automate up to 90% of their customer interaction with them.[1] The global chatbot market is projected to reach 2 billion dollars by 2024, growing at a CAGR (compound annual growth rate) of 29.7%.[2]

This widespread interest and demand for chatbot applications has emphasized the need to be able to quickly build complex chatbot applications supporting AI-based natural language processing [9] in order to be able to fluently chat with the user. Moreover, any non-trivial chatbot requires accessing an orchestration [10] of internal and external services in order to perform the requested user actions (e.g. to check and query the data to be served back to the user or to actually execute some processes/actions in response). As such, chatbots are becoming complex software artifacts that require a more methodical development approach to be developed with the proper quality standards.

As such, the definition of chatbots becomes a challenging task that requires expertise in a variety of technical domains, ranging from natural language processing to a deep understanding of the APIs of the targeted instant messaging platforms and third-party services to be integrated.

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone.

[1]https://chatbotsmagazine.com/chatbot-report-2019-global-trends-and-analysis-a487afec05b

[2]https://www.alliedmarketresearch.com/chatbot-market

So far, chatbot development platforms have mainly addressed the first challenge, typically by relying on external *intent recognition providers*, which are natural language (NL) processing frameworks providing user-friendly interfaces to define conversation assets. As a trade-off, chatbot applications are tightly coupled to their *intent recognition providers*, hampering their maintainability, reusability and evolution. Typically, once the chatbot designer chooses a specific chatbot development platform, she ends up in a vendor lock-in scenario, especially with the NL engine coupled with the platform. Similarly, current chatbot platforms lack proper abstraction mechanisms to easily integrate and communicate with other external platforms the company may need to interact with.

This work aims to tackle all these issues by raising the level of abstraction at what chatbots are defined. To this purpose, we introduce Xatkit, a novel model-based chatbot development framework that aims to address this question using Model Driven Engineering (MDE) techniques: domain-specific languages, platform independent bot definitions, and runtime interpretation. Indeed, Xatkit embeds a dedicated chatbot-specific modeling language to specify user intentions, computable actions and callable services, combining them in rich conversation flows. Conversations can either be started by a user awakening Xatkit or by an external event that prompts a reaction from Xatkit (e.g. alerting a user that some event of interest fired on an external service the bot is subscribed to).

The resulting chatbot definition[3] is independent of the intent recognition provider (which can be configured as part of the available Xatkit options) and frees the designer from the technical complexities of dealing with messaging and backend platforms as Xatkit can be deployed through the Xatkit runtime component on them without performing any additional steps. Xatkit is the result of a collaboration work between the Open University of Catalonia and the Berger-Levrault company who is interested in adapting chatbots as part of its citizen portal service offering.

This paper extends our previous work [11][4] in the following directions:

- Ability to define event-based conversations. Now Xatkit can subscribe to external events that may induce Xatkit to start a conversation and not just respond to conversations started by the user
- A significant growth in the tools maturity, both in the number of platforms and features offered in each platform.
- A new regex-based NLP parser to be used for testing purposes or very simple bots (e.g. as a way to check the Xatkit installation was successfully completed without requiring to setup as well a connection to a remote NLP engine).
- Better support for the Platforms mechanism and extended list of platforms, including the support for voicebots.
- An initial validation as part of an ongoing initiative to use Xatkit in an education setting
- A specific packaging for a Xatkit Development Toolkit that lowers the barrier to entry for those potential contributors that want to start tinkering with the code.
- A completely reworked related work section.
- Plus many other minor changes (e.g. refinements on the concrete syntax) based on the feedback and experience we got since the first release.

The rest of the paper is structured as follows: Section II introduces preliminary concepts used through the article. Section III shows an overview of the Xatkit framework, while Section IV, V and VI detail its internal components. Section VII presents the tool support, Section IX compare our approach with existing chatbot design techniques and Section VIII a first empirical evaluation. Finally, Section X summarizes the key points of the paper, draws conclusions, and present our future work.

II. PRELIMINARIES AND RUNNING EXAMPLE

This section defines the key concepts of a chatbot application that are reused through this article.

Chatbot design [12] typically relies on parsing techniques, pattern matching strategies and Natural Language Processing (NLP) to represent the chatbot knowledge. The latter is the dominant technique thanks to the popularization of libraries and cloud-based services such as DialogFlow [13] or IBM Watson Assistant [14], which rely on Machine Learning (ML) techniques to understand the user input (based on a set of training sentences provided as part of the chatbot definition) and provide user-friendly interfaces to design the conversational flow.

However, Pereira and Díaz have recently reported that chatbot applications can not be reduced to raw language processing capabilities, and additional dimensions such as complex system engineering, service integration, and testing have to be taken into account when designing such applications [15]. Indeed, the conversational component of the application is usually the front-end of a larger system that involves data storage and service execution as part of the chatbot reaction to the user intent. Thus, we define a chatbot as an application embedding a *recognition engine* to extract *intentions* from user inputs, and an *execution component* performing complex event processing represented as a set of *actions*.

Intentions are named entities that can be matched by the recognition engine. They are defined through a set of *training sentences*, which are input examples used by the recognition engine's ML/NLP framework to derive a number of potential ways the user could use to express the intention.[5]

[3]In this paper, we use the terms bot, chatbot and voicebot indistinctly as Xatkit supports all of them via its set of supported platforms

[4]Xatkit was previously known as Jarvis but we changed the name since that paper was published

[5]In this article we focus on ML/NLP-based chatbots, but the approach could be extended to alternative recognition techniques

Matched intentions usually carry *contextual information* computed by additional extraction rules (e.g. a typed attribute such as a city name, a date, etc) available to the underlying application. In our approach, *Actions* are used to represent simple responses such as sending a message back to the user, as well as advanced features required by complex chatbots like database querying or external service calling. Finally, we define a *conversation path* as a particular sequence of received user *intentions* and associated *actions* (including non-messaging actions) that can be executed by the chatbot application.

In Xatkit, bots can also be triggered by events. They may subscribe to external *events* that trigger a reaction on their side without being prompted by a user starting a conversation with them. Same as for intents, the chatbot designer can define the set of actions to be executed in response to an event.

A. RUNNING EXAMPLE

We now present how Xatkit's concepts are put into practice through a running example: our case study will be a multi-platform chatbot aiming to optimize the collaboration between project owners and end-users of a given GitHub open source project.

On the one hand, our chatbot will aim to assist newcomers in the definition of issues on the Github platform, a reported concern in the open source community [16]. Instead of directly interacting with the GitHub repository, users of our software could use the chatbot to report a new issue they found. The chatbot helps them to specify the repository to open the issue in and the relevant error information, and opens the issue on their behalf. The chatbot is deployed as a Slack app (i.e. the conversation between the user and the chatbot takes place on the Slack messaging platform). In particular, in this example, we will assume that the project in question is a WordPress plugin and therefore the bot will take care of asking two crucial questions: what WordPress version the user is on and what PHP version the host is running. These two data points are critical for the maintainer to reliably reproduce the issue and efficiently debug the user error, saving time for everybody.

On the other hand, this same chatbot can also be useful to alert the owner every time the status of the GitHub repository changes (either because somebody has used the chatbot as a user or because somebody directly interacted with the repository's issue tracker in GitHub). Instead of forcing the owner to keep an eye on GitHub or subscribe to its complex notification system, our bot will ping him on her platform of choice (in this case, we assume that she wants to be pinged on both, Slack and Discord, since it is there where she spends most of its online time). Once alerted, she will be able to respond back to the bot to perform some immediate reply action like labeling or assigning the opened issue. Note that in this second scenario it is not the user who starts the conversation but the bot.

Although this chatbot is obviously a simplification of what a proper chatbot for GitHub could look like (with more complex information flows and richer set of alerts after updates on the GitHub side), we believe it is representative enough of the current chatbot landscape, where chatbots usually need to interact with various input/output platforms and keep track of contextual information and partial responses in order to provide richer user experiences.

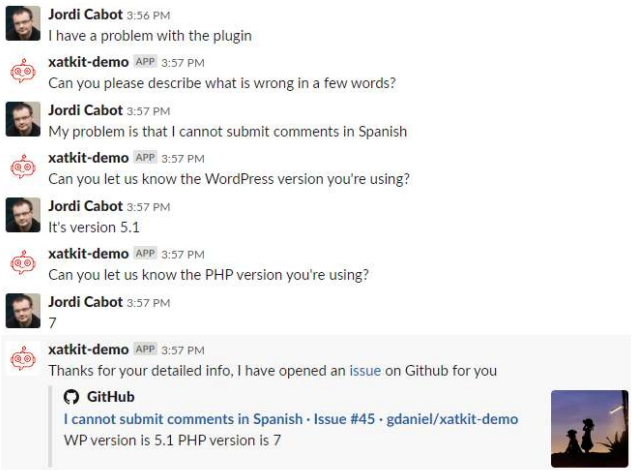

FIGURE 1. The chatbot collecting user information before opening the issue.

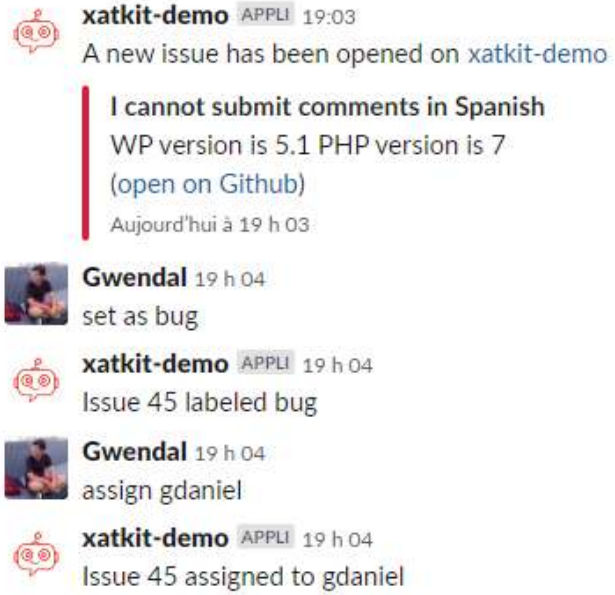

FIGURE 2. The bot alerting the project owner that a new issue has just been opened.

In the following we show how this chatbot is defined with the help of the Xatkit modeling language, and we detail how the runtime component manages its concrete deployment and execution.

III. XATKIT FRAMEWORK OVERVIEW

Our approach applies Model Driven Engineering (MDE) principles to the chatbot building domain. As such, chatbot

models become the primary artifacts that drive all software (chatbot) engineering activities [17]. Existing reports have emphasized the benefits of MDE in terms of productivity and maintainability compared to traditional development processes [18], making it a suitable candidate to address chatbot development and deployment, which, as discussed before, goes much further than simply processing natural language sentences.

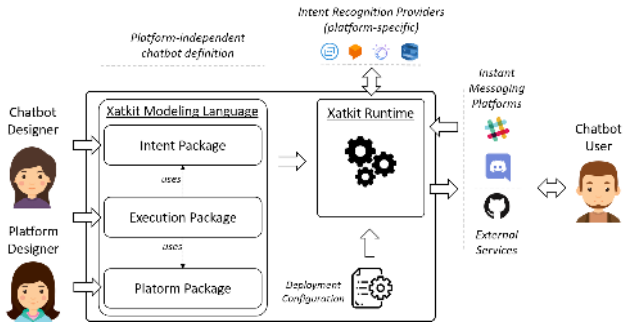

FIGURE 3. Xatkit framework overview.

The following Figure 3 presents an overview of our MDE-based chatbot approach and its main components. At design time the chatbot designer specifies the chatbot under construction using two domain-specific languages (DSLs) part of **Xatkit Modeling Infrastructure**:

- **Intent Package** to describe the user *intentions* using training sentences, contextual information extraction, and matching conditions (e.g. the intention to open an issue or the intention to select a repository, in our running example)
- **Execution Package** to bind user *intentions* to response *actions* as part of the chatbot behaviour definition (e.g. sending a welcome message to the user when he intents to open a new issue or actually creating the issue on the GitHub platform once the user has completed explaining it).

The actions in the Execution part of the bot often involve a set of orchestrated calls to services provided by the available *Platforms*. Platforms are defined by a Platform designer via a separate **Platform Package** and, once available, are enabled for all existing bots. Platforms are organized in a taxonomy so the chatbot designer can choose generic actions (e.g. a textual reply, something available in all chat-based platforms) or more specific ones (e.g. attaching a file to a message, only available in some specific platforms like Slack). The resulting platform definition hides all the technical details of the communication with the platforms.

These models are complemented with a *Deployment Configuration* file that specifies the *Intent Recognition Provider* to use (e.g Google's DialogFlow [13] or IBM Watson Assistant [14]), platform specific configuration parameters (e.g. OAuth credentials), as well as custom execution properties, which for instance can introduce some limited variability in the bot behaviour. Note that in the Xatkit infrastructure, all the *intent recognition providers* implement a common interface that allows switching from one to another transparently through configuration properties. Support for new providers can be easily achieved by implementing this common interface.

These assets constitute the input of the **Xatkit Runtime** component that starts by deploying the created chatbot. This implies registering the user *intents* to the selected *Intent Recognition Provider*(which involves translating the intents in the bot definition into the primitives/mechanisms available in that specific provider), connecting to the *Instant Messaging Platforms*, and starting the *External Services* specified in the *execution* model. Then, when a user input is received, the runtime forwards it to the *Intent Recognition Provider*, gets back the recognized intent and performs the required action associated to that intent based on the chatbot *execution* model.

This infrastructure provides three main benefits:

- The *Xatkit Modeling Language* packages decouple the different dimensions of a chatbot definition, facilitating the reuse of each dimension across several chatbots.
- Each sublanguage is totally independent of the concrete deployment and intent recognition platforms, easing the maintenance and evolution of the chatbot.
- The *Runtime* architecture can be easily extended to support new platform connections and computable actions. This aspect, coupled with the high modularity of the language, fosters new contributions and extensions of the framework.

Next sections cover each of these components and languages in more detail.

IV. XATKIT MODELING LANGUAGE

In the following we introduce the Xatkit Modeling Language, composed by a set of interrelated chatbot Domain Specific Languages (DSL) that provides primitives to design the user intentions, execution logic, and deployment platforms of the chatbot under construction (this latter one will be described in Section VI).

The Xatkit language is defined through two main components [19]: (i) an abstract syntax (metamodel) defining the language concepts and their relationships (generalizing the primitives provided by the major intent recognition platforms [13], [14], [20]), and (ii) a concrete syntax in the form of a textual notation to write chatbot descriptions conforming to the abstract syntax.[6] In the following we use the abstract syntax to describe the DSL packages and primitives, and the textual to show, via examples based on our running case study, how those concepts can be used to create bots. A modeling IDE for the language is also introduced in our tool support.

[6]A graphical notation sharing the same metamodel is left as further work. Curiously enough, business users are far more interested in having automatic importers that could generate the bot definition itself from internal documents than on having a graphical drag&drop interface. As such, importers have now higher priority

To decouple the definition of the user intentions the chatbot should recognize from the actions the chatbot should execute in response to those intents, our language is split up into two different sublanguages: the *Intent* and the *Execution* packages.

A. INTENT PACKAGE

Figure 4 presents the metamodel of the *Intent Package*, that defines a top-level *IntentLibrary* class containing a collection of *IntentDefinitions*. An *IntentDefinition* is a *named* entity representing a user intention. It contains a set of *Training Sentences*, which are input examples used to detect the user intention underlying a textual message. *Training Sentences* are split into *TrainingSentenceParts* representing input text fragments — typically words — to match.

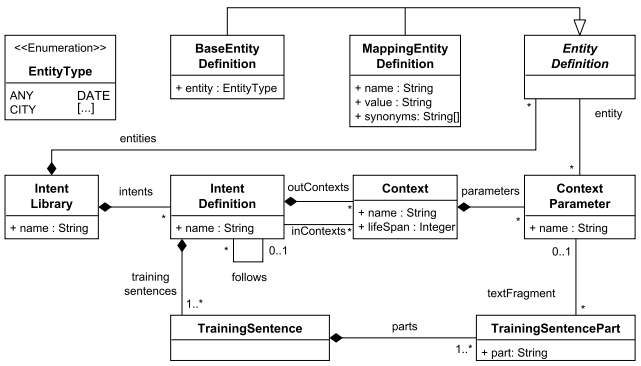

FIGURE 4. Intent package metamodel.

Each *IntentDefinition* defines a set of *outContexts*, that are named containers used to persist information along the conversation and customize intent recognition. A *Context* embeds a set of *ContextParameters* which define a mapping from *TrainingSentenceParts* to specific *EntityDefinitions*, specifying which parts of the *TrainingSentences* contain information to extract and store. In the current version of Xatkit *EntityDefinitions* can be either *BaseEntityDefinitions*, i.e. generic entities that are provided for all the intent recognition platforms such as *city* or *date*, or *MappingEntityDefinitions* that represent user-designed entities represented by a value and a list of synonyms. Note that a *Context* also defines a *lifespan* representing the number of user inputs that can be processed before deleting it from the conversation, allowing to specify information to retain and discard, and customize the conversation based on user inputs.

IntentDefinitions can also reference *inContexts* that are used to specify matching conditions. An *IntentDefinition* can only be matched if its referenced *inContexts* have been previously set, i.e. if another *IntentDefinition* defining them as its *outContexts* has been matched, and if these *Contexts* are active with respect to their *lifespans*. Finally, the *follow* association defines *IntentDefinition* matching precedence, and can be coupled with *inContext* conditions to finely describe complex conversation paths.

```
1  library Example
2
3  // Intents to report issues to the bot
4  // (Users of the WordPress plugin)
5
6  intent OpenBug {
7    inputs {
8      "The WordPress plugin is not working"
9      "I have a problem with the WP plugin"
10     "I'd like to report an error"
11     "I want to open a bug"
12     "I want to report a bug"
13     "There is an error in the WP plugin"
14   }
15 }
16
17 intent DescribeBug follows OpenBug {
18   inputs {
19     "My error is Error"
20     "The problem is Error"
21     "I get this error: Error"
22     "My error is that Error"
23     "The problem is that Error"
24     "I get the error Error"
25   }
26   creates context issue {
27     sets parameter title
28         from fragment Error (entity any)
29   }
30 }
31
32 intent TellWPVersion follows DescribeBug {
33   requires context issue
34   inputs {
35     "My version number is WPVersion"
36     "I use number WPVersion"
37     "It's version WPVersion"
38   }
39   creates context issue {
40     sets parameter wpversion
41         from fragment WPVersion (entity number)
42   }
43 }
44
45 // Intents to manage issues
46 // (Users of the WordPress plugin)
47
48  mapping LabelValue {
49   value "bug" synonyms: "error"
50   value "enhancement" synonyms: "improvement"
51   value "wontfix"
52 }
53
54 intent SetLabel {
55   requires context issue
56   inputs {
57     "Set label Label"
58     "Set as Label"
59     "This is a Label"
60   }
61   creates context  issue {
62     sets parameter issueLabel
63         from fragment Label (entity LabelValue)
64   }
65 }
```

LISTING 1. Example intents for the github case study.

Listing 1 shows a (partial) instance of the *Intent Package* from the running example introduced in Section II-A. The model defines the *IntentLibrary* `Example`, that contains four *IntentDefinitions* and a *MappingEntityDefinition*. The three first *IntentDefinitions* (`OpenBug`, `DescribeBug`, and `TellWPVersion`) correspond to user-related intents (i.e. the users of our WordPress plugin who want to report and issue). The last *IntentDefinition* `SetLabel` and the associated *MappingEntityDefinition* `LabelValue` correspond to an issue management intents typically triggered by the plugin's maintainer.

`OpenBug` is a simple *IntentDefinition* that does not *follow* any other intent nor require *inContext* value, and thus will be

the first intent matched in the conversation. It contains several training sentences specifying alternative inputs used to initiate the conversation. The `DescribeBug` intent follows the `OpenBug` one, and defines one *outContext* `issue`, with a default *lifespan* of 5,[7] and a single parameter `title`. Finally `TellWPVersion` asks for the WordPress version the user has installed. Note that the *ContextParameters* of these *IntentDefinitions* use *BaseEntityDefinition* for their value extraction —respectively `any` and `number`—, the latter explicitly looking for a number as a parameter to store in the context.

The `LabelValue` *MappingEntityDefinition* represents the issue labels understood by the bot. For our example we defined three issue labels: *bug*, *enhancement*, and *wontfix* as well as their synonyms. This entity is used to extract the *issueLabel* parameter value from the `SetLabel` *IntentDefinition*'s training sentences.[8] This intent is for project owners triaging open issues after the bot alerts them of the creation of a new issue.

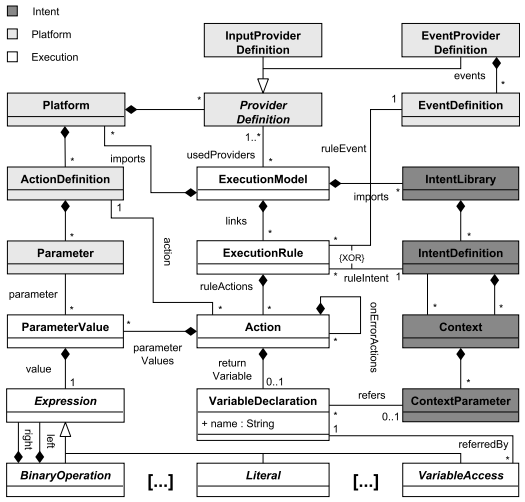

FIGURE 5. Execution package metamodel.

B. EXECUTION PACKAGE

The *Execution Package* (Figure 5) is an event-based language that represents the chatbot execution logic.

An *ExecutionModel* *imports* *Platforms* and *IntentLibraries*, and specifies the *ProviderDefinitions* used to receive user inputs and events. The *ExecutionRule* class is the cornerstone of the language, which defines the mapping between received *IntentDefinitions*/*EventDefinitions* and *Actions* to compute.

The *Action* class represents the reification of a *Platform ActionDefinition* with concrete *ParameterValues* bound to its *Parameter* definitions. These Actions are part of the definition of the Platform, as discussed in the next section. The *value* of a *ParameterValue* is represented as an *Expression* instance. Xatkit *Execution* language currently supports *Literals*, *Unary* and *Binary Operations*, as well as *VariableAccesses* that are read-only operations used to access *ContextParameters*.[9]

An *Action* can also define an optional *returnVariable* that represents the result of its computation, and can be accessed from other *Actions* through *VariableAccess Expressions*, allowing to propagate information between computed actions. Finally, an *Action* can also contain *onErrorActions*, which are specific *Actions* that are executed when the base one errored.

Listing 2 shows the *Execution* model from our running example. It *imports* the `Example` *IntentLibrary* and the `Slack` and `Github` *Platforms*.

```
1
2  import platform "GithubPlatform"
3  import platform "SlackPlatform"
4  import platform "DiscordPlatform"
5
6  use provider SlackPlatform.SlackIntentProvider
7  use provider GithubPlatform.GithubEventProvider
8
9  on intent OpenBug do
10   action SlackPlatform.Reply(message : "Can you please
         describe what is wrong in a few words?")
11
12 on intent TellPHPVersion do
13   action SlackPlatform.Reply(message : "Thanks for your
         detailed info")
14   def newissue = action GithubPlatform.OpenIssue(user : "
         jcabot", repository : "xatkit-tests", issueTitle :
         context(issue).get("title") ,
15       issueContent : "WP version is " + context(issue).
             get("wpversion") + " PHP version is " + context
             (issue).get("phpversion")
16   )
17   action GithubPlatform.SetLabel(issue : newissue, label
         : "bug")
18   action SlackPlatform.PostMessage(message : "A GitHub
         issue has been opened on your behalf with the title
          " + context(issue).get("title"), channel: config(
         slack.channel))
```

LISTING 2. Chatbot execution language example.

The defined *ExecutionModel* specifies two *ProviderDefinitions* that will receive user inputs from the `Slack` *Platform* and from GitHub generated events so that the bot is useful to both users attempting to report a bug and project owners that want to get an automatic notification once the bug is actually opened.

Let's focus first on the sub bot facing the user (Listing 2). Once the `OpenBug` *IntentDefinition* is matched, the bot replies asking the user to provide more information. Note that, this bot directly uses the Slack platform but we could redefine it in a more generic way by using the abstract Chat Platform if the chatbot designer would like to redeploy the bot as a web chat window, for instance. We would write a very similar behaviour for the other intents that collect information about the bug until we have everything we need and are ready to open the issue reporting the bug in GitHub. This is simply done by calling the *OpenIssue* method provided by the GithubPlatform using as parameters the data stored so far in the *bug* context.

[7]the lifespan indicates how many failed intents can happen before the information collected so far is forgotten and the conversation needs to restart

[8]Adding such enumeration constraints and synonyms are a good practice in existing ML-powered NLP tools that may not manage efficiently free text

[9]Note that future releases of Xatkit will integrate Xtext's Xbase language as its default expression language

As we have discussed above, the bot can also react to events. As shown in Listing 3, when the event *Issue_Opened* (one of the events generated by the GitHub platform) arrives, we can get all the details of the new issue and immediately alert the project owner in Slack and Discord. Note that the execution language does not limit the number of platforms to use in an interaction.

```
1
2  on event Issue_Opened do
3    action SlackPlatform.PostMessage(message : "A GitHub
         issue has been opened with the title " + context(
         issue).get("issue->title"), channel: config(slack.
         channel))
4    action DiscordPlatform.PostMessage(message : "A GitHub
         issue has been opened with the title " + context(
         issue).get("issue->title"), channel: config(discord
         .channel))
```

LISTING 3. Chatbot execution language example.

V. XATKIT RUNTIME

The **Xatkit Runtime** component is an event-based execution engine that deploys and manages the execution of the chatbot. Its inputs are the chatbot model (written with the **Xatkit Modeling Language**) and a *configuration* file holding deployment information and platform credentials. In the following we detail the structure of this *configuration* file, then we present the architecture of the **Xatkit Runtime** component. Finally, we introduce a dynamic view of the framework showing how input messages are handled by its internal components.

```
1  // Intent Recognition Provider Configuration
2  Xatkit.intent.recognition      = DialogFlow
3  Xatkit.dialogflow.project      = <DialogFlow Project ID>
4  Xatkit.dialogflow.credentials  = <DialogFlow Credentials>
5  // Abstract Platform Binding
6  Xatkit.platform.chat           = Xatkit.SlackPlatform
7  // Concrete Platform Configuration
8  Xatkit.slack.credentials       = <Slack Credentials>
9  Xatkit.discord.credentials     = <Discord Credentials>
10 Xatkit.slack.channel           = <Slack channel name>
11 Xatkit.discord.channel         = <Discord channel name>
12 Xatkit.github.credentials      = <Github Credentials>
```

LISTING 4. Chatbot deployment configuration example.

A. XATKIT DEPLOYMENT CONFIGURATION

The Xatkit *deployment configuration* file provides runtime-level information to setup and bind the platforms with whom the chatbot needs to interact either to get user input or to call as part of an action response. Listing 4 shows a possible configuration for the example used through this article. The first part (lines 1-4) specifies `DialogFlow` as the concrete *IntentRecognitionProvider* service used to match received messages against *IntentDefinitions*, and provides the necessary credentials. The second part of the configuration (lines 5-6) binds the concrete `Slack` platform (using its *path* attribute) to the abstract `Chat` used in the *Execution* model (Listing 2). This runtime-level binding hides platform-specific details from the *Execution* model, that can be reused and deployed over multiple platforms. The last part of the configuration (lines 7-10) specifies platform credentials.

B. ARCHITECTURE

Figure 6 shows an overview of the **Xatkit Runtime** internal structure, including illustrative instances from the running example (light-grey). The *XatkitCore* class is the cornerstone of the framework, which is *initialized* with the *Configuration* and *ExecutionModel* previously defined. This initial step starts the *InputProviders* that receive the user messages, as well as the *EventProviders* used to extract *EventInstances* from received third-party events (e.g. the *Issue_Opened* event in our running example), and setups the concrete *IntentRecognitionProvider* (in our case `DialogFlow`) employed to extract *RecognizedIntents*, which represent concrete instances of the specified *IntentDefinitions*.

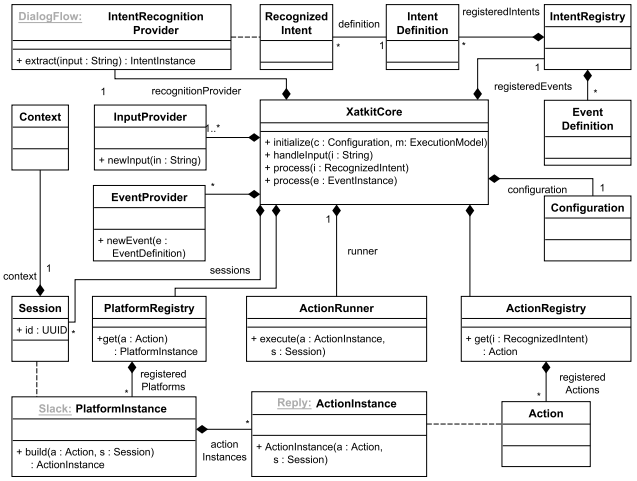

FIGURE 6. Xatkit runtime engine architecture overview.

The input *ExecutionModel* is then processed and its content stored in a set of *Registries* managing *IntentDefinitions*, *EventDefinitions*, *Actions*, and *Platforms*. The *PlatformRegistry* contains *PlatformInstances*, which correspond to concrete *Platform* implementations (e.g. the `Slack` platform from the running example) initialized with the *Configuration* file. *PlatformInstances build ActionInstances*, that contain the execution code associated to the *ActionDefinitions* defined in the *Intent* language, and are initialized with *Actions* from the *Execution* model. These *ActionInstances* are finally sent to the *ActionRunner* that manages their execution.

The *XatkitCore* also manages a set of *Sessions*, used to store *Context* information and *ActionInstance* return variables. Each *Session* defines a unique identifier associated to a user, allowing to separate *Context* information from one user input to another.

Figure 7 shows how these elements collaborate together by illustrating the sequence of operations that are executed when the framework receives a user message. To simplify the presentation, this sequence diagram assumes that all the internal structures have been *initialized* and that the

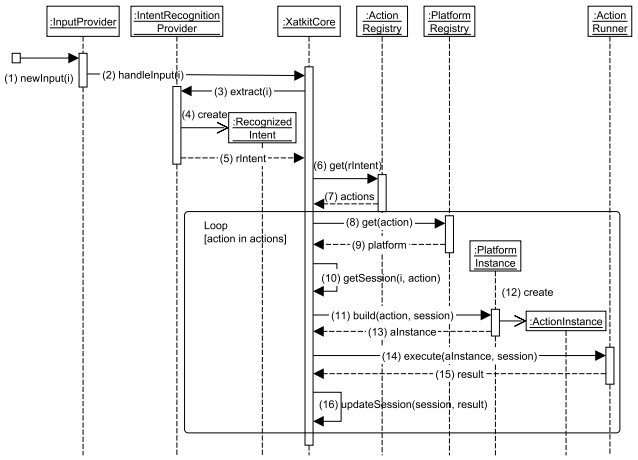

FIGURE 7. Runtime engine sequence diagram.

different registries have been populated from the provided *ExecutionModel*.

User inputs are received by the framework through the *InputProvider*'s *newInput* method (1), that defines a single parameter *i* containing the raw text sent by the user. This input is forwarded to the *XatkitCore* instance (2), that calls its *IntentRecognitionProvider*'s *extract* method (3). The input is then matched against the specified *IntentDefinitions*, and the resulting *RecognizedIntent* (4) is returned to the *XatkitCore* (5).

The *XatkitCore* instance then performs a lookup in its *ActionRegistry* (6) and retrieves the list of *Actions* associated to the *RecognizedIntent* (7). The *XatkitCore* then iterates through the returned *Actions*, and retrieves from its *PlatformRegistry* (8) their associated *PlatformInstance* (9). The user's *Session* is then retrieved from the *XatkitCore*'s *sessions* list (10). Note that this process relies on both the user input and the *Action* to compute, and ensures that a client *Session* remains consistent across action executions. Finally, the *XatkitCore* component calls the *build* method of the *PlatformInstance* (11), that constructs a new *ActionInstance* from the provided *Session* and *Action* signature (12) and returns it to the core component (13). Finally, the *XatkitCore* component relies on the *execute* method of its *ActionRunner* to compute the created *ActionInstance* (14) and stores its result (15), in the user's *Session* (16).

Note that due to the lake of space the presented diagram does not include the fallback logic that is triggered when the computation of an *ActionInstance* returns an error. Additional information on fallback and *on error* clauses can be found in the project repository.

VI. PLATFORM PACKAGE

Xatkit comes with a set of platforms packaged as part of the release.[10] Figure 8 shows a taxonomy of the 13 platforms

[10]Updated list of platforms available in the project wiki https://github.com/xatkit-bot-platform/xatkit-releases/wiki

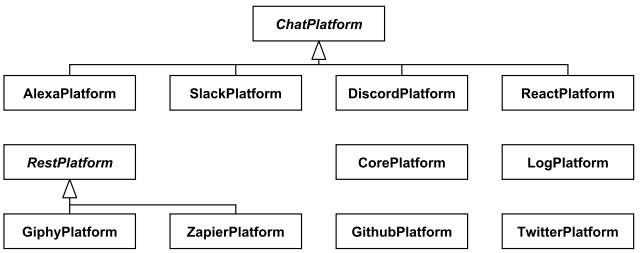

FIGURE 8. Taxonomy of the released platforms.

available so far including their inheritance links. Abstract classes (e.g. the text *ChatPlatform*) can be used in place of the concrete ones when the bot does not require any specific method only available in the concrete platform, thus facilitating the reusability of the bot.

Nevertheless, it is often the case that a chatbot designer requires a new platform (e.g. to talk with the internal services in the company). If so, a platform designer (remember Figure 3) will take care of defining and implementing the platform. This platform designer could be the same person that is designing the chatbot or somebody else hired to perform this specific task. One way or the other, once a platform is created, it can be reused by any past, present or future chatbot.

A. PLATFORM DEFINITION

To this purpose, Xatkit includes a DSL to define the capabilities of a given platform, both in terms of the actions that can be executed on the platform and the events the platform can emit, depending on the *ProviderDefinitions* offered by the platform. These are the capabilities that are explicitly used in the execution model to interact with the platform.

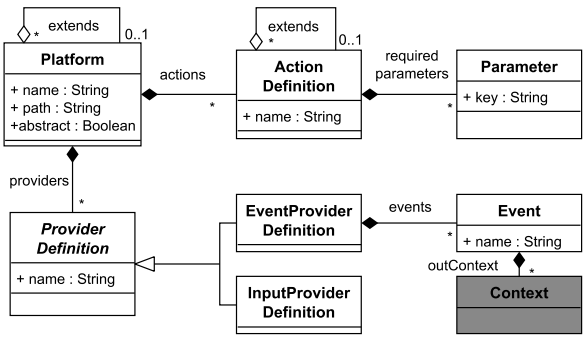

FIGURE 9. Platform package metamodel.

The *Platform Package* (Figure 9) defines the modeling primitives to define platforms. A *Platform* is defined by a *name*, and provides a *path* attribute that is used by the **Xatkit Runtime** component to bind the model to its concrete implementation. A *Platform* holds a set of *ActionDefinitions*, which are signatures of its supported operations. *ActionDefinitions* are identified by a *name* and define a set of *required Parameters*. A *Platform* can be *abstract*, meaning that it does not provide an implementation for its *ActionDefinitions* but it

represents, instead, a family of similar platforms. This feature allows to define chatbots in a more generic way.

As an example, the `Chat` *Platform* in Listing 6 is an *abstract* platform that defines three *ActionDefinitions*: `PostMessage`, `PostFile`, and `Reply`. The first two *ActionDefinitions* require two parameters (the message/-file and the channel to post it), and the third one defines a single parameter with the content of the reply. The `Github` *Platform* (Listing 7) defines a single *ActionDefinition* `OpenIssue` with the parameters `repository`, `title`, and `content`. Note that these are the actions that we have used to implement the bot described in the previous sections.

```
1  Abstract Platform Chat
2    path "Xatkit.ChatPlatform"
3    actions
4      PostMessage(message, channel)
5      PostFile(file, channel)
6      Reply(message)
7
8  Platform Slack   extends Chat
9    path "Xatkit.SlackPlatform"
10 Platform Discord extends Chat
11   path "Xatkit.DiscordPlatform"
```

LISTING 6. Chat platform example.

```
1  Platform Github
2
3  path "Xatkit.Github"
4
5  providers {
6          event GithubEventProvider {
7                  event Issue_Opened
8                  creates context "issue" {
9                      sets parameter "issue->title"
10                 }
11     }
12 }
13
14 actions
15   OpenIssue(repository, title, content)
```

LISTING 7. Github platform example (only showing the actions and events used in the running example).

A *Platform* can *extend* another one, and inherit its *ActionDefinitions*. This mechanism is used to define specific implementations of *abstract Platforms*. As an example, the concrete `Slack` and `Discord` *Platforms extend* the `Chat` one and implement its *ActionDefinitions* for the Slack and Discord messaging applications, respectively.

Finally, *ProviderDefinitions* are named entities representing either message processing capabilities that can be used as inputs for the chatbot under design (*InputProviderDefinition*), or dedicated event receiver that will trigger execution rules when specific events are received (*EventProviderDefinition*).As an example, Listing 7 shows the definition of the *GithubEventProvider*, that describes an *EventDefinition* that is generated when a new issue is created on the GitHub repository. Note that similarly to *IntentDefinitions*, *EventDefinitions* contains *Context* defining the information extracted from the received event.

B. PLATFORM IMPLEMENTATION

Xatkit platforms are implemented as standalone Java projects implementing the *Xatkit Platform Interface*. This interface is used internally by the *Xatkit Runtime* component to dynamically load platforms, create action instances, and run them when an execution rule is matched.

A Xatkit platform consist of a main class holding platform-specific data accessible to all the actions (e.g. OAuth token for our GitHub platform), and a collection of classes representing the *Action* that can be called on the platform (e.g. the GitHub platform contains an OpenIssue class that contains the code to actually open an issue on GitHub). In addition, optional classes can be defined to add *ProviderDefinitions* to the platform, and containing the code responsible of parsing received inputs/events into Xatkit *IntentDefinition* and *EventDefinition* instances.

To implement these classes, platform developers can rely on the generic architecture of the framework as well as a set of utility classes provided by Xatkit to perform REST requests, parse events (typically JSON payloads), and integrate their code in Xatkit's internal life-cycle with minimum efforts.

C. VOICE SUPPORT

Among all the predefined platforms, voice platforms are of special interest and deserve additional explanation.

Right now, Xatkit supports Alexa[11] while Google Assistant[12] is currently under development.

In both cases, the implementation strategy has been the same: rely on the speech-to-text functionality of platform to translate any voice input into plain text. As an example, we defined a generic Alexa *skill* accepting any kind of voice command and delegating the processing of the extracted text to a preset Xatkit server. The whole process is wrapped in a regular *InputProvider*, that can be imported the same way as standard messaging platform such as Slack.

With this approach, defining a voicebot does not require additional voice-specific techniques (nor more technical knowledge) than defining standard chatbots with Xatkit. As a side benefit, our approach makes Xatkit bots multimodal, since translating an existing chatbot to a voicebot can be done easily by switching from one *InputProvider* to another.

VII. TOOL SUPPORT

The Xatkit framework is open source and released under the Eclipse Public License v2.[13] The source code of the project and the Eclipse update site are available on the Xatkit GitHub organization.[14]

As part of this organization, we provide the different releases of the framework, the runtime component and the repositories for the several platform components and connectors. There is also a wiki (linked from the

[11] https://developer.amazon.com/en-US/alexa/alexa-voice-service
[12] https://assistant.google.com/
[13] https://www.eclipse.org/legal/epl-2.0/
[14] https://github.com/xatkit-bot-platform

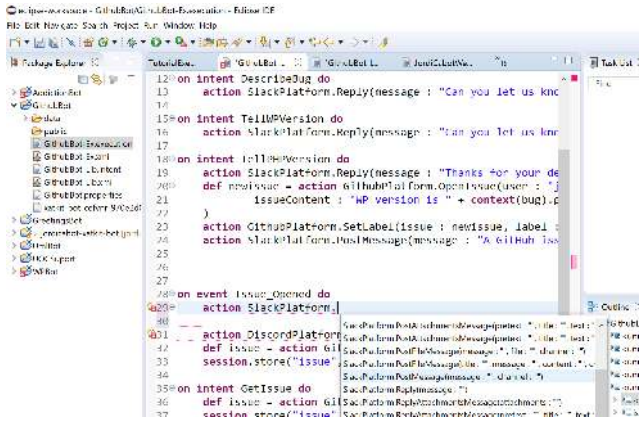

FIGURE 10. The xatkit editor.

GitHub organization) and an external website[15] that provide additional documentation, installation instructions and news around the project.

To facilitate the specification of the chatbots we provide and Eclipse editor that supports auto-completion, syntactic and semantic validation, and can be installed from the Xatkit Eclipse update site. The concrete syntaxes of the Xatkit modeling languages are implemented with Xtext [21], an EBNF-based language used to specify grammars and generate the associated toolkit containing a meta-model of the language, a parser, and textual editors.

The execution of the chatbots mainly relies on the **Xatkit Runtime** engine. At its core, the engine is a Java library that implements all the execution logic available in the chatbot languages. In addition, Xatkit provides a full implementation of the *IntentRecognitionProvider* interface for Google's DialogFlow engine [13], as well as the concrete *PlatformInstance* implementations for the Slack, Discord, and Github platforms used in the running example (and a few others as mentioned in the previous section). The runtime component can be downloaded and deployed on a server as a standalone application, or integrated in an existing application using a dedicated Maven dependency. Additional integrations (in particular with *nlp.js*[16] to enable a local NL analysis) are underway. Note that all these new integrations become immediately available to all existing bots due to the clean and modular architecture of Xatkit.

Xatkit also embeds a monitoring component that stores and computes metrics to evaluate the quality of the underlying *IntentRecognitionProvider*. These metrics can be accessed using a dedicated REST API provided by the Xatkit server,[17] and describe:

- The average number of intents matched per user session
- The average number inputs that have not been matched per user session
- The distribution of matched intents (to help designers understand which conversation flows are the most followed)
- The list of unmatched inputs to help designers integrate them in their existing intents when relevant
- The average recognition confidence level per user session, as well as per intent, in order to ease the detection of intents that should be improved
- The average user session time

Note that for now the computed metrics focus on the quality of the intent recognition, but we are currently working on it to cover other aspects of running Xatkit bots such as performance measurements, or user engagement.

Overall, the Xatkit organization in GitHub is composed of:

- 27 code repositories
- 1154 commits
- 13 supported platforms

We would like to highlight that Xatkit has already four external contributors that are developing additional platforms for the organization. To attract even more contributors, we have created a specific Xatkit Development Toolkit that facilitates the experimentation with the platform code.

VIII. VALIDATION

Xatkis is used internally by several colleagues that have adopted some of the examples[18] we have created. Several pilot projects are under evaluation at the moment to apply Xatkit in Student Support, eHealth and Public Administration scenarios. All these preliminary use cases have provided useful feedback that has been key to improve Xatkit.

Moreover, as part of our teaching initiative that aims to bring Xatkit to the classroom[19] as a tool to teach students concepts like DSLs, NLP, bots,... we are also starting joint teaching initiatives with several institutions.

Precisely, in this section, we would like to focus on the first completed experience that allowed us to conduct an initial validation of the usefulness and benefits of Xatkit with the students of a master seminar on model-driven engineering taught at the university "Universidad de la República" (Uruguay). Using students as participants remains a valid simplification of reality needed in laboratory contexts [22].

A. EMPIRICAL SETTING

The seminar lasted three days and was taught by Robert Clarisó (a member of our research group, but not a co-author of Xatkit). The first two days focused on teaching core modeling principles while the last one included a remote presentation by Gwendal Daniel introducing Xatkit as an example of the use of model-based solution to build advanced software systems. After the three teaching days, the students were asked to spend a minimum of 30 additional hours of work developing a model-based solution around Xatkit. This could either be:

[15] https://xatkit.com/

[16] https://github.com/axa-group/nlp.js

[17] Presenting this data in a more user-friendly way (e.g. via a dedicated dashboard) is left for future work.

[18] https://xatkit.com/chatbot-examples/

[19] https://modeling-languages.com/building-chatbots-use-case-modeling-course/

- A bot reusing existing Xatkit platforms
- An implementation of a new platform that extended Xatkit to a new domain

The first assignment was considered feasible for everybody while the second one was targeting students that could have a special interest in the chatbots domain and wanted to go deeper in their exploration of Xatkit.

The seminar was taken by 22 students, out of which 20 went for the first option and two chose the second one. Bots developed on top of existing Xatkit platforms included extensions of our GitHub bot with more complex conversation paths gathering additional information from the user, use of additional events (e.g. comments on issues and pull requests) to create advanced bots, and integration of additional actions to automatically close issues or reply to comments. The students who chose to implement a new platform developed an initial Twitter support for Xatkit allowing to retrieve trending tweets, check the user's direct message inbox, and post new tweets on behalf of the user.[20] At the end of the term, all solutions were collected and marked together with the local responsible of the course (Dr. Daniel Caligari).

Additionally, all students were asked to complete an online survey explaining their perception of Xatkit. The survey asked a few questions about their past experience with chatbots (only one had ever created a chatbot in the past even if 75% of them were already working as software developers, at least part-time) and then they were asked to evaluate Xatkit from several perspectives. Finally, they were provided with open text areas to explain what they liked the most and the least about Xatkit, what was missing, and any other general comment they wanted to discuss. The survey was optional and anonymous. Out of the 22 students enrolled in the seminar, 17 students completed it. Next section discusses some of the survey results.

B. SURVEY RESULTS

The following figures show that Xatkit was evaluated very positively in a number of categories, ranging from the overall experience with Xatkit (Figure 11), the usability of Xatkit's modeling language (Figure 12), the power of the platform abstraction mechanism (Figure 13), the benefits of defining chatbots at a higher-abstraction level (Figure 14), how this helps to the portability between messaging platforms (Figure 15) and the quality of the Xatkit IDE (Figure 16).

Regarding the improvement suggestions, most were around adding new platforms to Xatkit, especially input platforms such as Facebook and WhatsApp that would have allowed them to build chatbots for those platforms as part of their assignment. Two students also asked for support for other Intent Recognition Providers beyond DialogFlow. Also, more entry-level tutorials were requested.

Among the most appreciated elements of Xatkit, the DSLs themselves, the separation of concerns (between intent, execution and platform models) and the easy integration with external platforms were the most commented. Overall, they also said they saw a lot of potential in the platform and appreciated the fact that it was open-source.

[20]Their collaboration with us continued after the seminar and their initial proposal has become the official Twitter platform in Xatkit

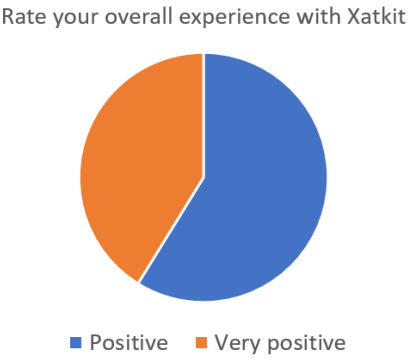

FIGURE 11. Overall experience with xatkit.

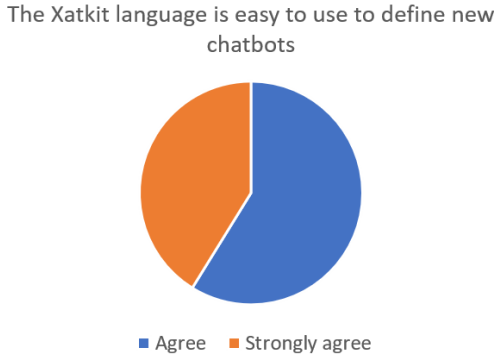

FIGURE 12. Is the xatkit language easy to use?.

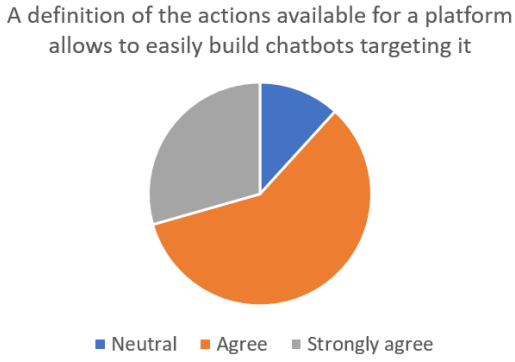

FIGURE 13. Does the platform definition DSL facilitate the definition of bots targeting that platform?.

IX. RELATED WORK

Our chatbot modeling approach reuses concepts from agent-oriented software engineering [23] and event-based system modeling [24] and adapts them to the chatbot/conversational domain. As far as we know, Xatkit is the first attempt to provide a fully modular and extensible platform-independent chatbot modeling language.

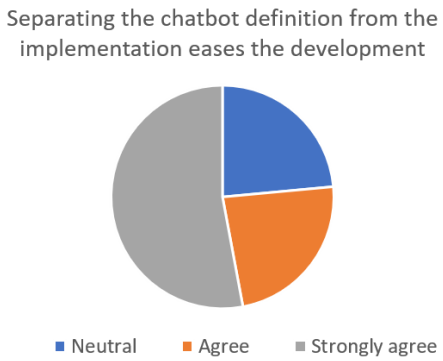

FIGURE 14. Does the separation between the chatbot specification and its implementation ease the development?.

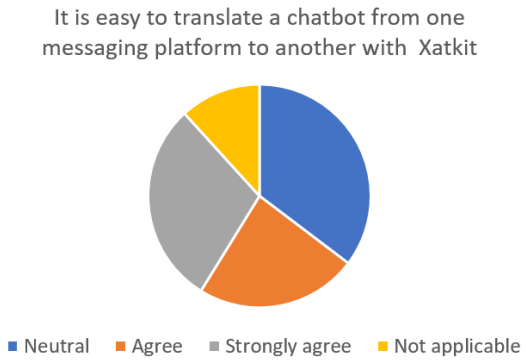

FIGURE 15. Is it easy to translate a chatbot from one messaging platform to another?.

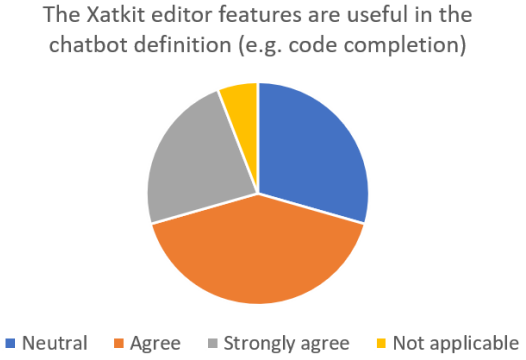

FIGURE 16. Rate the usefulness of the editor features (code completion, syntax highlighting,...)?.

In what follows we aim to compare in more detail Xatkit with the plethora of other chatbot development platforms. Note that most of the links are to software platforms and not to research works. Even if a few of the platforms derive from sound research results, most are recent initiatives showing that the chatbot market is still in its infancy[21] and needs consolidation and maturity.

A. NATURAL LANGUAGE UNDERSTANDING TOOLKITS

Some tools focus on the parsing/matching of user utterances (i.e. given the user text, understanding the intention the user is trying to express).

[21] Indeed, the first chatbot [25] was created more than 40 years ago but its real emergence is happening right now

Neural networks are typically used to attempt to classify the user utterance in one of the predefined intents in the chatbot definition. The neural network is trained with the example sentences provided together with the intents definition, often augmented with the use of synonyms and stemming procedures. Stemming reduces the derivations of a word to its root to improve the accuracy of the classification process. Well-known examples of these tools (many times provided as part of a cloud service) are Dialog Flow[22], IBM Watson,[23] Amazon Lex,[24] Microsoft LUIS [25] or nlp.js.[26] Others, like Stanford Core NLP[27] use a more traditional parsing approach.

A few of the above tools do offer a user interface to completely define a chatbot within the platform but with limited capabilities. In short, any complex chatbot response (beyond purely giving a text-based answer) requires manual coding and API management, making them unfit for non-professional developers. This is exactly one of the core design principles behind Xatkit.

As such, these tools are not competitors to Xatkit. On the contrary, Xatkit relies on them for the NL part of Xatkit's runtime engine. Still, to avoid vendor lock-in, Xatkit imposes a common interface that facilitates switching from one NLU provider to another (e.g. a company may want to change due to cost issues or due to a better support for non-English languages).

B. CHATBOT DEVELOPMENT PLATFORMS

There are dozens of chatbot platforms. We first start by filtering out popular companies like HelloMyBot,[28] Inbenta,[29] BotCore,[30] 1MilionBot[31] or Imperson[32] that offer bot consulting services but not a public tool to build the bot yourself.

Other proposals are more of building blocks for more advanced development platforms. E.g bot frameworks like Microsoft Bot Framework,[33] BotKit[34] or Hubot[35] provide a set of programming libraries/scripts to facilitate the coding and deployment of chatbot applications. They usually help you to integrate intent recognition engines and some messaging platforms (a specific solution for this would be Smooch[36] as well, even if this is not really their focus, but

[22] https://dialogflow.com/
[23] https://www.ibm.com/watson/services/natural-language-understanding/
[24] https://aws.amazon.com/lex/
[25] https://www.luis.ai/
[26] https://github.com/axa-group/nlp.js
[27] https://stanfordnlp.github.io/CoreNLP/
[28] https://hellomybot.io/
[29] https://www.inbenta.com/en/
[30] https://botcore.ai/
[31] https://1millionbot.com/en/
[32] https://imperson.com/
[33] https://dev.botframework.com/
[34] https://botkit.ai/
[35] https://hubot.github.com/
[36] https://smooch.io/

they require manual work to complete the process and to perform more advanced connections with external services. Also, they hardly ever provide any specific domain-specific language and mostly rely on JavaScript or some other programming language for the bot development.

More similar to Xatkit, we have full-fledged chatbot development platforms that users can execute on their own. They all provide a textual/graphical interface that allows to specify the user intentions, the conversation path, and the contextual information to be maintained through the conversation, and offer excellent natural language processing capabilities. For the latter, many of them rely on the NLU engines from the previous section (as Xatkit does) while others have their own proprietary engine. We believe this latter option has some important drawbacks (e.g. duplication of coding efforts) but it offers a larger control on the engine itself.

Relevant examples of this family of tools are Octane.ai[37] FlowXO,[38] Oracle Digital Assistant,[39] Botsify,[40] ManyChat,[41] Engati,[42] Chatfuel,[43] PandoraBots,[44] Rasa[45] or Hubtype.[46]

To begin with, most of these tools are closed source.[47] Moreover, while each of them supports a different number of input messaging platforms, they do not typically offer any extension capabilities. Therefore, they are only a good fit as long as your needs are in the scope of the platforms supported by the tool, hampering the evolution of your bot if the requirements change later on. This is especially worrisome regarding the plugging of external services. Most tools just offer an API to query the results of the intent recognition status and ask you to program yourself the integration with the third-party service. Instead, in Xatkit, the Platform DSL, its modular architecture and the possibility of working with abstract platforms are aimed to solve this issue. Finally, the possibility of creating chatbots that combine proactive and reactive behaviour (i.e. that can be activated by the user starting a conversation or by an external event relevant for the bot) is practically nonexistent in the above platforms.

All in all, Xatkit proposes a higher-abstraction solution to the chatbot domain that combines the benefit of platform-independent chatbot definition, including non-trivial chatbot actions and side effects, together with an easy deployment. Moreover, the extensibility of our modular design facilitates the integration of any external API/services as input/output source of the chatbot. These integrations can be shared and reused in future projects, which is when the benefits of modeling and abstraction are maximized [26]

C. CHATBOT COMPONENTS IN LOW-CODE SOLUTIONS

Given the model-based and low-code approach followed in Xatkit, we could also combine Xatkit with other low-code solutions (like Mendix,[48] OutSystems[49] or Genexus[50]) in order to generate complete software systems that need to integrate a chatbot as part of its user interface. Right now, low-code platforms are just starting to study the integration of chatbot components and could benefit from adopting a specific solution such as Xatkit.

And this intersection between bots and modeling can bring other additional advantages [27], like the use of chatbots to build the models themselves. This has explored in [28] and [29] and we are now collaborating with both teams to use Xatkit as core chatbot engine for their modeling efforts.

X. CONCLUSION

In this paper we introduced Xatkit, a multi-channel and multi-platform chatbot modeling framework. Xatkit proposes a set of domain-specific languages to decouple the chatbot definition from the technical details of the platform-specific aspects where the bot is going to be deployed. This increases the reusability of the chatbot and facilitates its redeployment when the needs of the company change, including the possibility of evolving the NLU engine used during the text analysis phase.

Moreover, the runtime component can be easily extended to support additional platform-specific actions and events beyond those already shipped with the current version of Xatkit. For instance, some platforms like Alexa or Trello have been recently added by external contributors to the core Xatkit team.

Xatkit is ready to be used in real-case scenarios. But it has still plenty of room for improvements. At the language level we plan to improve the variability of the bot specification, moving towards a product-line approach that enables companies to create and quickly update several versions of the same bot (e.g. to create localized versions of the bot for each branch of the company). At the framework level, we plan to work on the integration of chatbot generators, able to create partial bot specifications from existing data sources within the company (e.g. FAQs or user guides). We also plan to study the combination of sentiment analysis and behavioural design patterns [30] to create more likeable and effective chatbots [31]. Finally, security and access-control is another

[37] https://octaneai.com/
[38] https://flowxo.com/
[39] https://www.oracle.com/application-development/cloud-services/digital-assistant/
[40] https://botsify.com/
[41] https://manychat.com/
[42] https://www.engati.com/
[43] https://chatfuel.com/
[44] https://home.pandorabots.com/home.html
[45] https://rasa.com/
[46] https://www.hubtype.com/
[47] We cannot report here all the tools we have analyzed but out of the over 40 tools we explored, only 8 had an open source version
[48] https://www.mendix.com/
[49] https://www.outsystems.com/
[50] https://www.genexus.com/en/

important aspect of any chatbot design as we may want to allow users to query (or not) certain aspects of our data depending on their profile.

ACKNOWLEDGMENT

The authors would like to thank L. Baruffini, H. Ed-douibi, N. Erlichman, and F. Fernàndez Cecchetto for extending Xatkit with support for additional platforms and to Robert Clarisó and Daniel Caligari for his help during the empirical evaluation.

REFERENCES

[1] B. Nardi, S. Whittaker, and E. Bradner, "Interaction and outeraction: Instant messaging in action," in *Proc. 3rd CSCW Conf.*, 2000, pp. 79–88.

[2] R. Grinter and L. Palen, "Instant messaging in teen life," in *Proc. 5th CSCW Conf.*, 2002, pp. 21–30.

[3] L. C. Klopfenstein, S. Delpriori, S. Malatini, and A. Bogliolo, "The rise of bots: A survey of conversational interfaces, patterns, and paradigms," in *Proc. Conf. Designing Interact. Syst. (DIS)*, 2017, pp. 555–565.

[4] A. Xu, Z. Liu, Y. Guo, V. Sinha, and R. Akkiraju, "A new chatbot for customer service on social media," in *Proc. CHI Conf. Human Factors Comput. Syst. (CHI)*, 2017, pp. 3506–3510.

[5] A. Kerly, P. Hall, and S. Bull, "Bringing chatbots into education: Towards natural language negotiation of open learner models," *Knowl.-Based Syst.*, vol. 20, no. 2, pp. 177–185, Mar. 2007.

[6] N. T. Thomas, "An e-business chatbot using AIML and LSA," in *Proc. Int. Conf. Adv. Computing, Commun. Informat. (ICACCI)*, Sep. 2016, pp. 2740–2742.

[7] V. Subrahmanian, A. Azaria, S. Durst, V. Kagan, A. Galstyan, K. Lerman, L. Zhu, E. Ferrara, A. Flammini, and F. Menczer, "The DARPA Twitter bot challenge," *Computer*, vol. 49, no. 6, pp. 38–46, Jun. 2016.

[8] G. Inc, *The Road to Enterprise AI*. Pune, Maharashtra: RAGE Frameworks, 2017.

[9] P. Jackson and I. Moulinier, *Natural Language Processing for Online Applications: Text Retrieval, Extraction and Categorization*, vol. 5. Amsterdam, The Netherlands: John Benjamins, 2007,

[10] M. Brambilla, M. Dosmi, and P. Fraternali, "Model-driven engineering of service orchestrations," in *Proc. IEEE Congr. Services*, Los Angeles, CA, USA, Jul. 2009, pp. 562–569, doi: 10.1109/SERVICES-I.2009.94.

[11] G. Daniel, J. Cabot, L. Deruelle, and M. Derras, "Multi-platform chatbot modeling and deployment with the jarvis framework," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science), vol. 11483, P. Giorgini and B. Weber, Eds. Rome, Italy: Springer, Jun. 2019, pp. 177–193, doi: 10.1007/978-3-030-21290-2_12.

[12] J. Masche and N.-T. Le, "A review of technologies for conversational systems," in *Proc. 5th ICCSAMA Conf.* Springer, 2017, pp. 212–225. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-61911-8_19

[13] (2018). *DialogFlow Website*.[Online]. Available: https://dialogflow.com/

[14] (2018). *Watson Assistant Website*. [Online]. Available: https://www.ibm.com/watson/ai-assistant/

[15] J. Pereira and O. Díaz, "Chatbot dimensions that matter: Lessons from the trenches," in *Proc. 18th ICWE Conf.* Springer, 2018, pp. 129–135. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-91662-0_9

[16] D. Kavaler, S. Sirovica, V. Hellendoorn, R. Aranovich, and V. Filkov, "Perceived language complexity in GitHub issue discussions and their effect on issue resolution," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 72–83.

[17] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synth. Lectures Softw. Eng.*, vol. 1, no. 1, pp. 1–182, Sep. 2012.

[18] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Sci. Comput. Program.*, vol. 89, pp. 144–161, Sep. 2014.

[19] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Language Using Metamodels*. London, U.K.: Pearson, 2008.

[20] Amazon. (2018). *Amazon Lex Website*. [Online]. Available: https://aws.amazon.com/lex/

[21] L. Bettini, *Implementing Domain-Specific Language with Xtext Xtend*. Birmingham, U.K.: Packt, 2013.

[22] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empir Softw. Eng*, vol. 23, no. 1, pp. 452–489, Feb. 2018, doi: 10.1007/s10664-017-9523-3.

[23] N. Jennings and M. Wooldridge, "Agent-oriented software engineering," *Handbook Agent Technology*, vol. 18. 2001. [Online]. Available: https://link.springer.com/book/10.1007/978-3-642-54432-3

[24] S. Rozsnyai, J. Schiefer, and A. Schatten, "Concepts and models for typing events for event-based systems," in *Proc. Inaugural Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2007, pp. 62–70.

[25] J. Weizenbaum, "ELIZA – a computer program for the study of natural language communication between man and machine," *Commun. ACM*, vol. 26, no. 1, pp. 23–28, Jan. 1983, doi: 10.1145/357980.357991.

[26] O. Diaz and F. M. Villoria, "Generating blogs out of product catalogues: An MDE approach," *J. Syst. Softw.*, vol. 83, no. 10, pp. 1970–1982, Oct. 2010, doi: 10.1016/j.jss.2010.05.075.

[27] J. Cabot, R. Clarisó, M. Brambilla, and S. Gérard, "Cognifying model-driven software engineering," in *Software Technologies: Applications and Foundations*, Marburg, Germany, Jul. 2017, pp. 154–160, doi: 10.1007/978-3-319-74730-9_13.

[28] S. Perez-Soler, E. Guerra, and J. De Lara, "Collaborative modeling and group decision making using chatbots in social networks," *IEEE Softw.*, vol. 35, no. 6, pp. 48–54, Nov. 2018, doi: 10.1109/ms.2018.290101511.

[29] A. López, J. Sànchez-Ferreres, J. Carmona, and L. Padró, "From process models to chatbots," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science), vol. 11483, P. Giorgini and B. Weber, Eds. Rome, Italy: Springer, Jun. 2019, pp. 383–398, doi: 10.1007/978-3-030-21290-2_24.

[30] B. J. Fogg, *Persuasive Technology: Using Computers to Change What We Think and Do*. New York, NY, USA: Ubiquity, Dec. 2002. [Online]. Available: https://www.amazon.com/Persuasive-Technology-Computers-Interactive-Technologies/dp/1558606432, doi: 10.1145/764008.763957.

[31] R. Ren, J. W. Castro, S. T. Acu na, and J. de Lara, "Usability of chatbots: A systematic mapping study," in *Proc. 31st Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, Lisbon, Portugal, Jul. 2019, pp. 479–484, doi: 10.18293/SEKE2019-029.

[32] O. López-Pintado, M. Dumas, L. García-Bañuelos, and I. Weber, "Dynamic role binding in blockchain-based collaborative business processes," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science), vol. 11483, P. Giorgini and B. Weber, Eds. Rome, Italy: Springer, Jun. 2019, doi: 10.1007/978-3-030-21290-2_25.

GWENDAL DANIEL received the Ph.D. degree with the AtlanMod Team, Ecole des Mines de Nantes, France, in 2017. He is currently a Postdoctoral Fellow with the SOM Research Lab, Internet Interdisciplinary Institute (IN3), a Research Center, Universitat Oberta de Catalunya (UOC). He also funded by the MegaM@rt2 ECSEL-JU project. His research interests include model driven engineering, model persistence, query, and transformation techniques, domain specific languages, and applying model-based techniques for large-scale data applications. He received the Best Thesis Award from the GDR-GPL and the INFORSID Association, in 2018.

JORDI CABOT (Member, IEEE) received the B.Sc. and Ph.D. degrees in computer science from the Technical University of Catalonia.

He was a Leader of the INRIA and LINA Research Group, Ecole des Mines de Nantes, France, a Postdoctoral Fellow with the University of Toronto, a Senior Lecturer with the Open University of Catalonia, and a Visiting Scholar with the Politecnico di Milano. He is currently an ICREA Research Professor with the Internet Interdisciplinary Institute. His research interests include software and systems modeling, formal verification, and the role AI can play in software development (and vice versa). He has published over 150 peer-reviewed conference and journal articles on these topics. Apart from his scientific publications, he writes and blogs about all these topics in several sites. He is a member of the ACM.

LAURENT DERUELLE is currently a Research Manager with Berger-Levrault. He is also an in charge of managing collaborative projects with universities and research labs all the way towards the product industrialization. He actively participates in foundamental and applied research activities in the fields of big data/big analytics, cloud computing, distributed artificial intelligence and mulitagent systems, UI, and the IoT.

MUSTAPHA DERRAS is currently the R&D and Innovation Executive Director of Berger-Levrault. He has accomplished executive experience in management of technology and solutions realizations with more than 30 years of professional background working for major companies, such as General Electric, Cadence Design Systems, or Berger-Levrault. With extensive capability in leading teams of all sizes (10–500 persons) in software development, product marketing, research and innovation, he is also a decisive decision maker on many occasions in fields like organization, negotiation (M&A and alliances), and strategy. He also involving in business management, innovation funding, and activities leading.