Multi-Platform Chatbot Modeling and Deployment with the Jarvis Framework

Gwendal Daniel[1], Jordi Cabot[1,2], Laurent Deruelle[3], and Mustapha Derras[3]

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC)
gdaniel@uoc.edu
[2] ICREA
jordi.cabot@icrea.cat
[3] Berger-Levrault
{first.last}@berger-levrault.com

Abstract. Chatbot 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. They typically rely on existing cloud infrastructures and artificial intelligence techniques to efficiently process user inputs and extract conversation information. While these frameworks are efficient to design simple chatbot applications, they still require advanced technical knowledge to define complex conversations and interactions. In addition, the deployment of a chatbot application usually requires a deep understanding of the targeted platforms, increasing the development and maintenance costs. In this paper we introduce the Jarvis framework, that tackles these issues by providing a Domain Specific Language (DSL) to define chatbots in a platform-independent way, and a runtime engine that automatically deploys the chatbot application and manages the defined conversation logic. Jarvis is open source and fully available online.

Keywords: MDE, DSL, Chatbot Design, Chatbot Deployment

1 Introduction

Instant messaging platforms have been widely adopted as one of the main technology to communicate and exchange information [22,9]. 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 [18]. Chatbots have proven useful in various contexts to automate tasks and improve the user experience, such as automated customer services [32], education [16], and e-commerce [30]. Moreover, existing reports highlight the large-scale usage of chatbots in social media [29], and emphasize that chatbot design will become a key ability in IT hires in the near future [12].

This widespread interest and demand for chatbot applications has emphasized the need to be able to quickly build complex chatbot applications supporting natural language processing [13], custom knowledge base definition [27], as well as complex action responses including external service composition. However, the definition of chatbots remains a challenging task that requires expertise in a variety of technical domains,

ranging from natural language processing to a deep understanding of the API of the targeted instant messaging platforms and third-party services to be integrated.

So far, chatbot development platforms have mainly addressed the first challenge, typically by relying on external *intent recognition providers*, that are natural language 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.

This work aims to tackle both issues by raising the level of abstraction at what chatbots are defined, and can be summarized by the following design research question [31]

Can we improve the development of chatbot applications by abstracting out the platforms complexity and deployment configurations in order to allow designers to focus on the logic of the designed chatbot?

In this paper we introduce Jarvis, a novel model-based chatbot development framework that aims to address this question using Model Driven Engineering (MDE) techniques. Jarvis embeds a dedicated chatbot-specific modeling language to specify user intentions, computable actions and callable services, combining them in rich conversation flows. The resulting chatbot definition is independent of the intent recognition provider and messaging platforms, and can be deployed through the Jarvis runtime component on a number of them while hiding the technical details and automatically managing the conversation. Jarvis is employed in a joint project with the Berger-Levrault company.

The rest of the paper is structured as follows: Section 2 introduces preliminary concepts used through the article. Section 3 shows an overview of the Jarvis framework, while Section 4 and 5 detail its internal components. Section 6 presents the tool support, and Section 7 compare our approach with existing chatbot design techniques. Finally, Section 8 summarizes the key points of the paper, draws conclusions, and present our future work.

2 Background

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

Chatbot design [19] 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 [8] or IBM Watson Assistant [11], which rely on Machine Learning (ML) techniques to understand the user input 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 [24]. 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*, that 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[4]. 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.

3 Jarvis Framework

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 [4]. Existing reports have emphasized the benefits of MDE in terms of productivity and maintainability compared to traditional development processes [10], making it a suitable candidate to address chatbot development and deployment. In the following we first introduce a running example and then we present an overview of our MDE-based chatbot approach and its main components.

3.1 Running Example

Our case study is a simple example of a multi-platform chatbot aiming to assist newcomers in the definition of issues on the Github platform, a reported concern in the open source community [15]. 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 class/es affected by the issue, 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) and, beyond creating the issue itself, the chatbot sends an alert message to the repository's development team channel hosted on the Discord platform.

Although this chatbot is obviously a simplification of what a chatbot for GitHub could look like, we believe it is representative enough of the current chatbot landscape, where chatbots usually need to interact with various input/output platforms to provide rich user experiences.

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

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

3.2 Framework Overview

Figure 1 shows the overview of the Jarvis Framework. A designer specifies the chatbot under construction using the **Jarvis Modeling Language**, that defines three core packages:

- **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)
- **Platform Package** to specify the possible *actions* available in the potential target platforms, including those existing only on specific environments (e.g. posting a message on a Slack channel, opening an issue on Github, etc)..
- **Execution Package** to bind user *intentions* to *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).

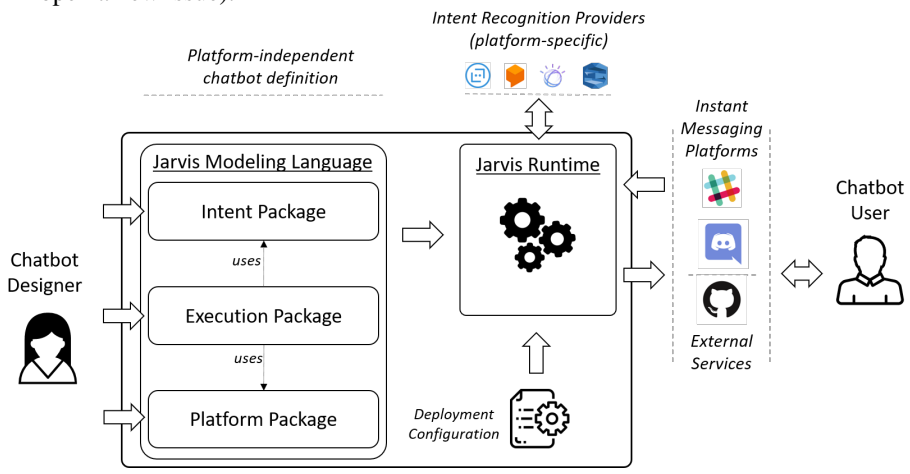

Fig. 1. Jarvis Framework Overview

These models are complemented with a *Deployment Configuration* file, that specifies the *Intent Recognition Provider* platform to use (e.g Google's DialogFlow [8] or IBM Watson Assistant [11]), platform specific configuration (e.g. Slack and Discord credentials), as well as custom execution properties.

These assets constitute the input of the **Jarvis Runtime** component that starts by deploying the created chatbot. This implies registering the user *intents* to the selected *Intent Recognition 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 based on the chatbot *execution* model.

This infrastructure provides three main benefits:

- The *Jarvis Modeling Language* packages decouple the different dimensions of a chatbot definition, facilitating the reuse of each dimension across several chatbots (e.g. the Slack platform definition can be reused in all chatbots interacting with Slack).

– Each sublanguage is totally independent of the concrete deployment and intent recognition platforms, easing the maintenance and evolution of the chatbot.
– The *Jarvis 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 that are further discussed in Section 5.

In the following we detail the **Jarvis Modeling Language** (Section 4) and **Jarvis Runtime** components (Section 5), and we show how they are used to define and deploy our example chatbot over multiple platforms.

4 Jarvis Modeling Language

In the following we introduce the Jarvis Modeling Language, a chatbot Domain Specific Language (DSL) that provides primitives to design the user intentions, execution logic, and deployment platform of the chatbot under construction.

The DSL is defined through two main components [17]: (i) an abstract syntax (metamodel) defining the language concepts and their relationships (generalizing the primitives provided by the major intent recognition platforms [8,11,1]), and (ii) a concrete syntax in the form of a textual notation to write chatbot descriptions conforming to the abstract syntax [5]. In the following we use the former to describe the DSL packages, and the latter to show instance examples based on our running case study. A modeling IDE for the language is also introduced in our tool support.

4.1 Intent Package

Figure 2 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 an 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.

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 *EntityTypes*, specifying which parts of the *TrainingSentences* contain information to extract and store. 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,

[5] A graphical notation sharing the same metamodel is left as further work

the *follow* association defines *IntentDefinition* matching precedence, and can be coupled with *inContext* conditions to finely describe complex conversation paths.

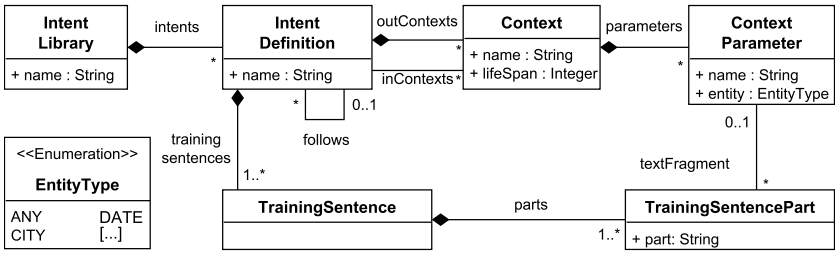

Fig. 2. Intent Package Metamodel

Listing 1 shows an example instance of the *Intent Package* from the running example introduced in Section 3.1. The model defines the *IntentLibrary* `Example`, that contains three *IntentDefinitions*: `OpenNewIssue`, `SpecifyRepository`, and `SpecifyClass`.

`OpenNewIssue` 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 three training sentences specifying alternative inputs used to initiate the conversation. The `SpecifyRepository` intent follows the `OpenNewIssue` one, and defines one *outContext* `RepositoryContext`, with a *lifespan* of 5[6], and a single parameter `name`. Note that this example shows the two syntax variants used to define parameters, the first one (line 13) is an inline definition, while the second one (line 14-21) is an explicit definition that matches the user input replacing the `MyRepo` fragment. While inline definitions are simpler to specify, explicit definitions allow to express advanced matching rules, such as parameters spanning over multiple *TrainingSentenceParts* or multi-context parameters.

Finally, the `SpecifyClass` *IntentDefinition* defines a single training sentence, and the *inContext* rule specifying that the `RepositoryContext` must be active in order to match it. This implies that the `SpecifyRepository` *IntentDefinition* must have been matched in the last 5 interactions, according to the context's *lifespan*.

Listing 1. Example Intents for the Github Case Study

```
1  library Example
2
3  OpenNewIssue {
4    inputs {
5      "I want to create an issue", "Open an issue", "New issue"
6    }
7  }
8  SpecifyRepository follows OpenNewIssue {
9    inputs {
10     "In repository (RepositoryContext:name=@any)",
11     "The issue is located in repo MyRepo",
```

[6] DialogFlow uses a default lifespan value of 5 that allows 5 unrecognized user inputs before forgetting the conversation contexts

```
12        "MyRepo"
13      }
14      outContext "RepositoryContext" (lifespan=5) {
15        param name <- "MyRepo" (@any)
16      }
17    }
18    SpecifyClass {
19      inputs {
20        "In class (ClassContext:name=@any)"
21      }
22      inContext "RepositoryContext"
23    }
```

4.2 Platform Package

The *Platform Package* (Figure 3) defines the capabilities of a given implementation platform (e.g. Slack, Discord, and Github) through a set of *ActionDefinitions* and *InputProviderDefinitions*.

A *Platform* is defined by a *name*, and provides a *path* attribute that is used by the **Jarvis Runtime** component (see Section 5) 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 2 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 3) defines a single *ActionDefinition* `OpenIssue` with the parameters `repository`, `title`, and `content`.

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, *InputProviderDefinitions* are named entities representing message processing capabilities that can be used as inputs for the chatbot under design. Messaging *Platforms* typically define a *default provider*, that can be complemented with additional *providers* with specific capabilities (e.g. listen to a specific channel or user). Note that *default providers* are implicitly set in the *Platform* language.

Listing 2. Chat Platform Example

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

Listing 3. Github Platform Example

```
1  Platform Github
2
3  path "jarvis.Github"
4
5  actions
6    OpenIssue(repository, title,
7      content)
```

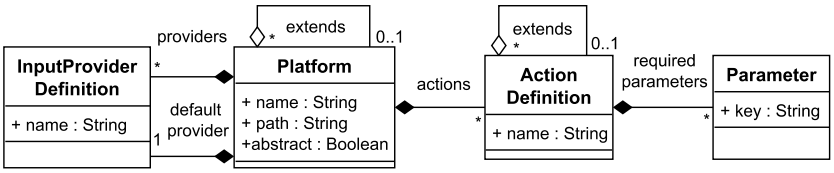

Fig. 3. Platform Package Metamodel

4.3 Execution Package

The *Execution Package* (Figure 4) is an event-based language that represents the chatbot execution logic.

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

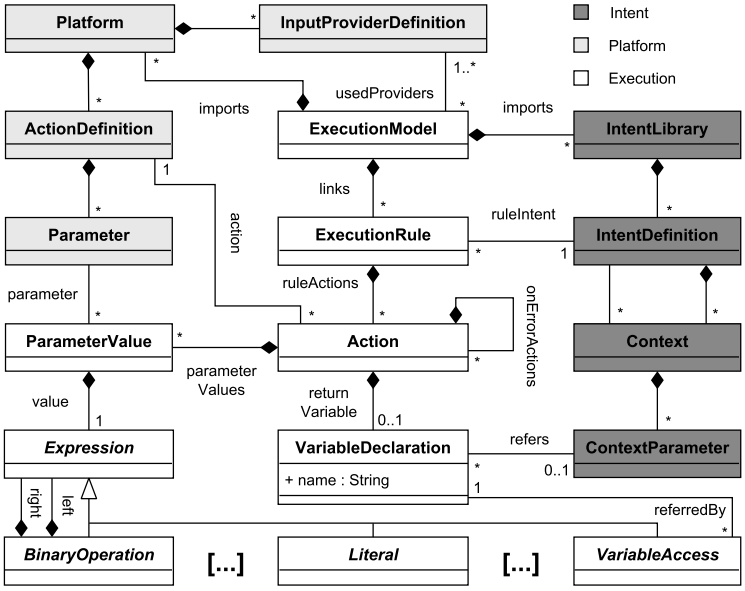

Fig. 4. Execution Package Metamodel

The *Action* class represents the reification of a *Platform ActionDefinition* with concrete *ParameterValues* bound to its *Parameter* definitions. The *value* of a *ParameterValue* is represented as an *Expression* instance. Jarvis *Execution* language currently supports *Literals*, *Unary* and *Binary Operations*, as well as *VariableAccesses* that are read-only operations used to access *ContextParameters*.

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 4 shows the *Execution* model from our running example. It *imports* the `Example` *IntentLibrary*, the generic `Chat` *Platform*, as well as the concrete `Github` and `Discord` *Platforms*. Note that Jarvis *Execution* language allows to *import* both concrete and *abstract Platforms*, the concrete implementations of the latter can be specified to the **Jarvis Runtime** component (see next section).

The defined *ExecutionModel* specifies a single *InputProviderDefinition* that will receive user inputs from the `Chat` *Platform*. Resulting *IntentDefinitions* are handled by three *ExecutionRules* following the conversation path. The first one (lines 8-9) is triggered when the `OpenNewIssue` *IntentDefinition* is matched, and posts a simple `Reply` message starting the conversation. The second one (lines 11-13) matches the `SpecifyRepository` *IntentDefinition*, and posts two *Replies*: the first one echoing the provided `RepositoryContext.name` *ContextParameter*, and the second one asking the user to specify the class related to the issue. Finally, the third *ExecutionRule* is triggered when the `SpecifyClass` *IntentDefinition* is matched, and performs three *Actions* on different platforms: the first one posts a `Reply` through the `Chat` *Platform* and displays the gathered *ContextParameter* values, the second one relies on the `Github` *Platform* to open a new issue by accessing the `name` of the `Repository` and the corresponding `Class` from the context, and the third one posts a reminder message on the Discord channel used by the development team. Note that the second *Action* defines an *onError* clause that prints an error message to the user if the chatbot was not able to compute the `openIssue` action.

Listing 4. Chatbot Execution Language Example

```
1  import library  Example
2  import platform Chat
3  import platform GithubModule
4  import platform DiscordModule
5
6  listen to Chat
7
8  on intent OpenNewIssue do
9    Chat.Reply("Sure, I'll help you to write it! Which repository would you like to
         report an issue for?")
10
11 on intent SpecifyRepository do
12   Chat.Reply("Alright, I have noted that your issue is related to repository {
         $RepositoryContext.name}")
13   Chat.Reply("Which class is affected by the issue?")
14
15 on intent SpecifyClass do
16   Chat.Reply("Ok! I am opening an issue in repository {$RepositoryContext.name}
         for the class {$ClassContext.name}, thanks!"
17   GithubModule.openIssue({$RepositoryContext.name}, {$ClassContext.name}, "There
         is an issue in class {$ClassContext.name}")
18         on error do Chat.Reply("I can't open the issue on the repository, please
               try again later")
19   DiscordModule.PostMessage("A new issue has been opened on repository {
         $RepositoryContext.name}", "dev-channel")
```

5 Jarvis Runtime

The **Jarvis 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 **Jarvis 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 **Jarvis Runtime** component. Finally, we introduce a dynamic view of the framework showing how input messages are handled by its internal components.

5.1 Jarvis Deployment Configuration

The Jarvis *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 5 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 4). 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.

Listing 5. Chatbot Deployment Configuration Example

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

5.2 Architecture

Figure 5 shows an overview of the **Jarvis Runtime** internal structure, including illustrative instances from the running example (light-grey). The *JarvisCore* class is the cornerstone of the framework, which is *initialized* with the *Configuration* and *ExecutionModel* previously defined. This initial step starts the *InputProvider* receiving user messages, and setups the concrete *IntentRecognitionProvider* (in our case `DialogFlow`) employed to extract *RecognizedIntents*, which represent concrete instances of the specified *IntentDefinitions*.

The input *ExecutionModel* is then processed and its content stored in a set of *Registries* managing *IntentDefinitions*, *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 *JarvisCore* also manages a set of *Sessions*, used to store *Context* information and *ActionInstance* return variables. Each *Session* defines an unique identifier associated to a user, allowing to separate *Context* information from one user input to another.

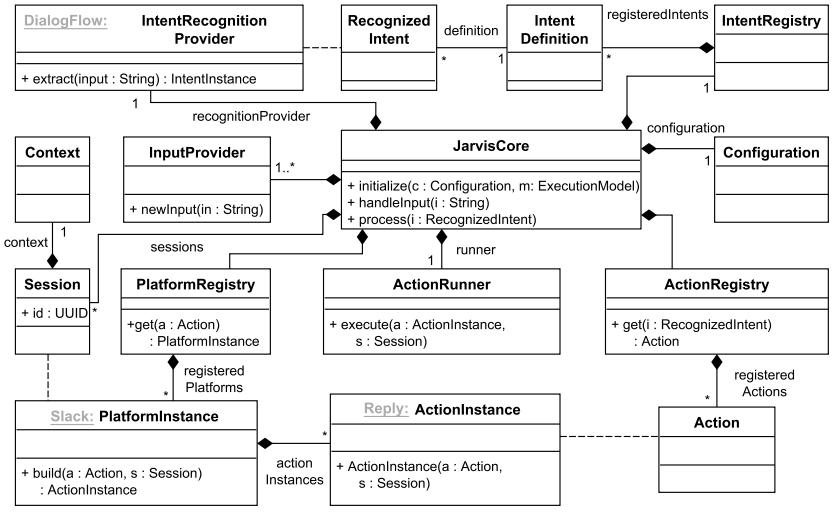

Fig. 5. Jarvis Runtime Engine Architecture Overview

Figure 6 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 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 *JarvisCore* 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 *JarvisCore* (5).

The *JarvisCore* instance then performs a lookup in its *ActionRegistry* (6) and retrieves the list of *Actions* associated to the *RecognizedIntent* (7). The *JarvisCore* 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 *JarvisCore*'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 *JarvisCore* 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 *JarvisCore* 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 a *ActionInstance* returns an error. Additional information on fallback and *on error* clauses can be found in the project repository.

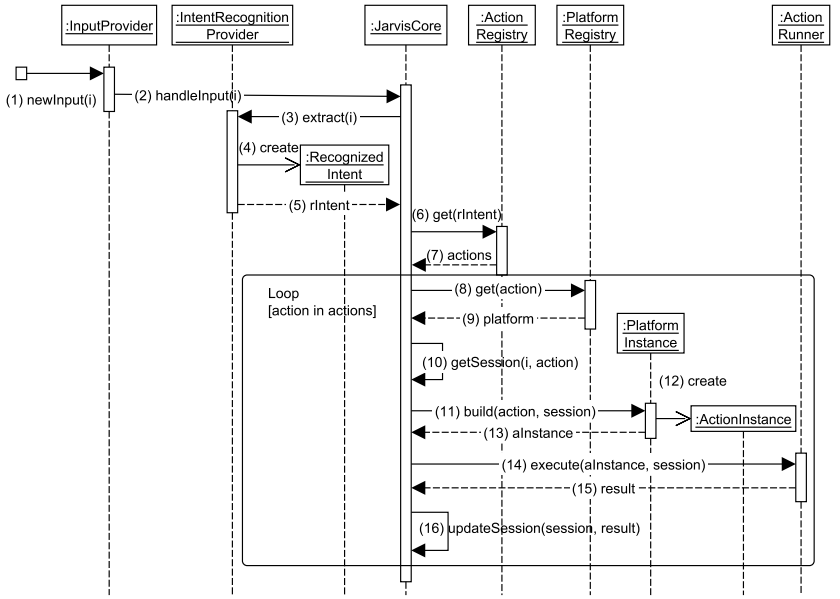

Fig. 6. Runtime Engine Sequence Diagram

6 Tool Support

The Jarvis framework is open source and released under the Eclipse Public License v2[7]. The source code of the project and the Eclipse update site are available on Github[8], which also includes a wiki providing installation instructions as well as developer resources to extend the framework.

The concrete syntax of the Jarvis modeling language is implemented with Xtext [2], 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 provided editors support auto-completion, syntactic and semantic validation, and can be installed from the Jarvis Eclipse update site.

The **Jarvis Runtime** engine is a Java library that implements all the execution logic presented in this paper. In addition, Jarvis provides a full implementation of the *Inten-*

[7] `https://www.eclipse.org/legal/epl-2.0/`
[8] `https://github.com/SOM-Research/jarvis`

tRecognitionProvider interface for Google's DialogFlow engine [8], as well as the concrete *PlatformInstance* implementations for the Slack, Discord, and Github platforms used in the running example. 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.

7 Related Work

Our chatbot modeling approach reuses concepts from agent-oriented software engineering [14] and event-based system modeling [26] and adapts them to the chatbot/-conversational domain. As far as we know, Jarvis is the first attempt to provide a fully platform-independent chatbot modeling language.

So far, chatbot development has been mostly performed by means of directly defining the chatbot intentions and responses within a specific chatbot platform such as DialogFlow[8], Watson Assistant [11] or Lex [1]. They all provide an online 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. However, they all have limited integration capabilities with other platforms. Any complex chatbot response (beyond purely giving a text-based answer) requires manual coding and API management, making them unfit for non-professional developers.

Bot coding frameworks like Microsoft Bot Frameworks [20] provide a set of programming libraries to implement and deploy chatbot applications. They usually facilitate the integration with intent recognition engines and some messaging platforms, but still require manual integration of those and any other external service. Other platforms like Botkit [3] or some low-code platforms [5,28,21] predefine a number of integrations with major messaging platform. While this helps if we are aiming at building simple conversational chatbots they still need to be tightened to one of the above platforms (for powerful language skills, e.g. for intent recognition) and require, as before, manual coding of advanced chatbot action. Finally, a few low-code platforms such as FlowXO [7] also provide support for triggering actions within the conversation. However, they are typically defined as a closed environment that cannot be easily extended by the designer with new actions and/or intent recognition platforms.

Conversely, Jarvis proposes an MDE approach that combines the benefit of platform-independent chatbot definition, including non-trivial chatbot actions and side effects, together with an easy deployment on any major chatbot platform for optimal natural language processing. Moreover, the extensibility of our modular design facilitates the integration of any external API and services as input/output source of the chatbot. These integrations can be shared and reused in future projects. On the other hand, Jarvis' generic chatbot design may hide useful platform-specific features that are not supported by all the vendors (e.g. DialogFlow's small talk). This could be addressed by adding a step in the design process that would refine Jarvis' platform independent model into a platform-specific model where designers could enrich the bot with specific features of the platform.

8 Conclusion

In this paper we introduced Jarvis, a multi-platform chatbot modeling framework. Jarvis decouples the chatbot modeling part from the platform-specific aspects, increasing the reusability of the conversational flows and facilitating the deployment of chatbot-enabled applications over a variety of chatbot service providers. The runtime component can be easily extended to support additional platform-specific actions and events.

Jarvis is the core component of an industrial case study in collaboration with Berger-Levrault that aims to generate chatbots for citizen portals (chatbots' mission is to help citizens navigate the portal to autonomously complete a number of city-related obligations). As part of this project we plan to perform a detailed evaluation of the expressiveness of the chatbot modeling language and the overall usability and productivity of the framework relying on evaluation techniques such as [25,23].

We also plan to enrich the Jarvis framework with advanced conversation capabilities such as intent recognition confidence level and conditional branching that are not supported for the moment. We are also exploring the support of generic events such as webhooks and push notifications. This would allow the modeling of reactive bots that can actively listen and respond to non-conversational events as well (e.g. a bot that wakes up as soon as a new issue is created on Github and immediately engages with the user to help clarifying the issue description). We are also studying how to extend our approach to support chatbot deployment over smart assistants such as Amazon Alexa. Another future work is the extension of the presented DSLs to support variation points at the metamodel level allowing to generate families of chatbots, e.g. using product line techniques [6].

Acknowledgement

This work has been partially funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant management No. 737494 (MegaMRt2 project) and the Spanish government (TIN2016-75944-R project).

References

1. Amazon. Amazon Lex Website, 2018. URL: `https://aws.amazon.com/lex/`.
2. L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.
3. Botkit. Botkit Website, 2018. URL: `https://botkit.ai`.
4. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.
5. Chatfuel. Chatfuel Website, 2018. URL: `https://chatfuel.com/`.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*, volume 3. Addison-Wesley Reading, 2002.
7. FlowXO. FlowXO Website, 2019. URL: `https://flowxo.com/`.
8. Google. DialogFlow Website, 2018. URL: `https://dialogflow.com/`.
9. R.E. Grinter and L. Palen. Instant Messaging in Teen Life. In *Proc. of the 5th CSCW Conference*, pages 21–30. ACM, 2002.

10. J. Hutchinson, J. Whittle, and M. Rouncefield. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors that Lead to Success or Failure. *SCP*, 89:144–161, 2014.
11. IBM. Watson Assistant Website, 2018. URL: `https://www.ibm.com/watson/ai-assistant/`.
12. Gartner Inc. *The Road to Enterprise AI*. RAGE Frameworks, 2017.
13. P. Jackson and I. Moulinier. *Natural Language Processing for Online Applications: Text Retrieval, Extraction and Categorization*, volume 5. John Benjamins Publishing, 2007.
14. N.R. Jennings and M. Wooldridge. Agent-Oriented Software Engineering. *Handbook of Agent Technology*, 18, 2001.
15. 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. of the 32nd ASE Conference*, pages 72–83. IEEE, 2017.
16. A. Kerlyl, P. Hall, and S. Bull. Bringing Chatbots into Education: Towards Natural Language Negotiation of Open Learner Models. In *Applications and Innovations in Intelligent Systems XIV*, pages 179–192. Springer, 2007.
17. A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education, 2008.
18. L.C Klopfenstein, S. Delpriori, S. Malatini, and A. Bogliolo. The Rise of Bots: a Survey of Conversational Interfaces, Patterns, and Paradigms. In *Proc. of the 12th DIS Conference*, pages 555–565. ACM, 2017.
19. J. Masche and N-T. Le. A Review of Technologies for Conversational Systems. In *Proc. of the 5th ICCSAMA Conference*, pages 212–225. Springer, 2017.
20. Joe Mayo. *Programming the Microsoft Bot Framework: A Multiplatform Approach to Building Chatbots*. Microsoft Press, 2017.
21. Mendix. Mendix Website, 2018. URL: `https://www.mendix.com/`.
22. B.A. Nardi, S. Whittaker, and E. Bradner. Interaction and Outeraction: Instant Messaging in Action. In *Proc. of the 3rd CSCW Conference*, pages 79–88. ACM, 2000.
23. J. Pereira and Ó. Díaz. A Quality Analysis of Facebook Messenger's most Popular Chatbots. In *Proc. of the 33rd SAC Symposium*, pages 2144–2150. ACM, 2018.
24. J. Pereira and Ó. Díaz. Chatbot Dimensions that Matter: Lessons from the Trenches. In *Proc. of the 18th ICWE Conference*, pages 129–135. Springer, 2018.
25. N.M. Radziwill and M.C. Benton. Evaluating Quality of Chatbots and Intelligent Conversational Agents. *arXiv preprint arXiv:1704.04579*, 2017.
26. S. Rozsnyai, J. Schiefer, and A. Schatten. Concepts and Models for Typing Events for Event-Based Systems. In *Proc. of the 1st DEBS Conference*, pages 62–70. ACM, 2007.
27. A. Shawar, E. Atwell, and A. Roberts. Faqchat as in Information Retrieval System. In *Proc. of the 2nd LTC Conference*, pages 274–278. Poznan: Wydawnictwo Poznanskie, 2005.
28. Smartloop. Smartloop Website, 2018. URL: `https://smartloop.ai/`.
29. VS Subrahmanian, A. Azaria, S. Durst, V. Kagan, A. Galstyan, K. Lerman, L. Zhu, E. Ferrara, A. Flammini, F. Menczer, et al. The DARPA Twitter bot challenge. *arXiv preprint arXiv:1601.05140*, 2016.
30. NT. Thomas. An E-business Chatbot using AIML and LSA. In *Proc. of the 5th ICACCI Conference*, pages 2740–2742. IEEE, 2016.
31. Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.
32. A. Xu, Z. Liu, Y. Guo, V. Sinha, and R. Akkiraju. A new Chatbot for Customer Service on Social Media. In *Proc. of the 35th CHI Conference*, pages 3506–3510. ACM, 2017.