Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Towards Conversational Syntax for Domain-Specific Languages using Chatbots

Sara Pérez-Soler[a] Mario González-Jiménez[a] Esther Guerra[a] Juan de Lara[a]

a. Modelling and Software Engineering Group (http://miso.es)
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract Traditionally, users have interacted with computers through graphical or command line interfaces. However, these may still be too technical for certain users, or cumbersome to use in some scenarios (e.g., in mobility). To tackle this issue, recent advances in natural language (NL) processing have boosted the proliferation of *chatbots*: programs whose user interface is NL and are frequently integrated within social networks.

In this paper, we explore the usage of NL as concrete syntax for domain-specific modelling languages, and propose an approach to automate the creation of modelling chatbots that converse with users to assist them in building domain-specific models. As chatbots are deployed on social networks, modelling becomes collaborative. We provide an implementation of our approach on top of Google's DialogFlow, and illustrate its usefulness on the basis of a case study to build and deploy streaming data applications using a conversational interface.

Keywords Model-Driven Engineering; Domain-Specific Languages; Chatbots; Natural Language Processing; DialogFlow.

1 INTRODUCTION

Traditionally, humans have interacted with computers through graphical or command line user interfaces [Jac12]. While these interaction mechanisms are well-known and widely accepted, some users may lack the technical skills required to use them, or may be inappropriate in certain scenarios, e.g., involving mobility.

Recent advances in natural language (NL) processing have boosted the proliferation of so-called *chatbots*. These are programs whose user interface is based on NL conversation, and are integrated within social networks like Telegram[1], Facebook messenger[2]

[1] https://telegram.org/
[2] https://www.messenger.com/

Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, Juan de Lara. *Towards Conversational Syntax for Domain-Specific Languages using Chatbots*. Licensed under Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). In Journal of Object Technology, vol. 18, no. 2, 2019, pages 5:1–21. doi:10.5381/jot.2019.18.2.a5

or Slack[3]. This approach to interact with software services has the advantage of avoiding the need to install new apps or swapping between the social network and an app to access the service. Moreover, chatbots are accessible by potentially large user communities, and in collaboration. According to a recent Gartner report [Moo18], 25% of worldwide customer service operations are expected to use chatbots by 2020.

Model-driven engineering (MDE) [Sch06] uses models to automate all phases in software development. Models in MDE are built using modelling languages, frequently domain-specific ones. Domain-specific languages (DSLs) [KT08] are languages tailored to a specific area, like logistics, urban planning or game development. Their concrete syntax is normally textual (similar to a programming language) or graphical (typically graph-like). Modelling using DSLs is an activity not only performed by developers, but there are proposals targeted to end-users, e.g., to define touristic routes [VPGdL17], build mobile apps [DP14], control molding machines [PP09], or create IoT applications [MNPP17].

Following that philosophy, this paper explores the usage of NL as concrete syntax for modelling languages. Hence, we propose an approach where models are built by dialoguing with a supporting chatbot in NL. As chatbots are deployed on social networks, modelling becomes collaborative and more amenable to be used in mobility than desktop applications. This approach would be particularly useful for DSLs oriented to end-user collaborative tasks, like organizing meetings or planning trips. These activities would be performed within the social network in NL and mediated by a bot, which reflects the user conversations in a domain-specific model. Then, this model could be executed, e.g., calling external APIs to book the meeting rooms or the trip hotels.

The increasing popularity of chatbots has raised numerous frameworks for their creation, like DialogFlow [Goo19], the IBM Watson Assistant [IBM19], the Microsoft Bot Framework [Mic19], or FlowXO [Flo19]. These frameworks offer cloud-based environments to describe the different aspects of the chatbot. However, creating a chatbot to instantiate a meta-model is time-consuming, repetitive and requires programming modelling services to take care of creating the models. Hence, we propose a novel approach to automate the generation of modelling chatbots for DSLs, show an implementation atop Google's DialogFlow, and illustrate its usefulness on a case study to create, deploy and execute streaming data applications using a modelling chatbot.

The objective of our work is threefold. First, to complement traditional modelling tools based on graphical or textual editors (e.g., within Eclipse) with another interaction paradigm. The use of NL requires less expertise from users than typical desktop-based modelling tools, while collaboration facilities and use in mobility are additional benefits. Second, we pursue the more ambitious goal of making available complete MDE solutions to end-users via conversation within social networks, realizing the vision of "*conversation as a platform*" (CaaP)[4]. Finally, we can use our approach to automate the generation of chatbot interfaces for existing information systems.

This paper follows our previous work [PGdLJ17, PGdL18], where we proposed a chatbot called SOCIO to assist in the creation of meta-models (i.e., class diagrams) via conversation. In this paper, we propose a methodology and prototype tool support to create NL concrete syntaxes for arbitrary meta-models (i.e., not limited to class diagrams), and demonstrate its practical value with a non-trivial case study.

[3] `https://slack.com/`
[4] A term coined by Satya Nadella, CEO of Microsoft.

The rest of this paper is organized as follows. Section 2 motivates our proposal using a running example. Section 3 overviews the basic concepts behind chatbots, focusing on DialogFlow. Section 4 presents our approach to create NL concrete syntaxes, and Section 5 tool support. Section 6 reports on a case study where we create a chatbot to build streaming data applications over a tool called Datalyzer [GdL18a]. Section 7 compares our approach with related research, and Section 8 concludes.

2 MOTIVATION AND RUNNING EXAMPLE

As a running example, we build a chatbot to define project plans conformant to the meta-model shown in Figure 1. `Projects` have a `name` and optionally a `goal`. They comprise a number of `TaskUnits` that can be organised in sequences through reference `next`, and have an `id`. There are three kinds of task units: `Tasks`, which may have a start date and end after a number of days; `Milestones`, which may have a start date but no duration, and are related to exactly one task; and `CompositeTasks` to group one or more task units. `Tasks` may have assigned `Resources`, both `Human` and `Technical`. The information of the former kind of resources is retrieved on-demand from an external database, i.e., class `Human` is a proxy to access the real data.

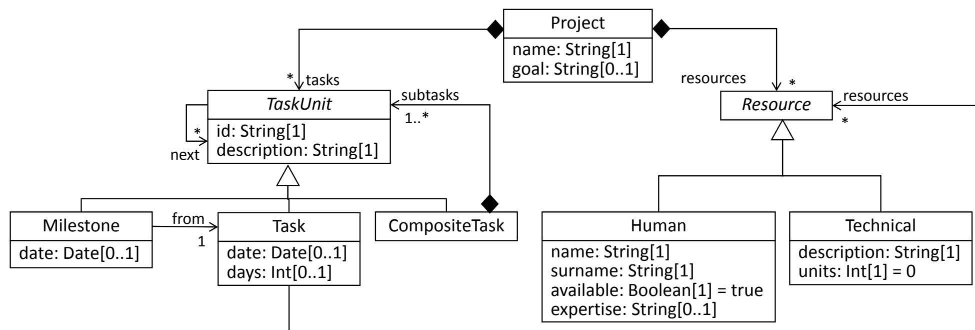


Figure 1 – Running example: A domain meta-model to describe project plans.

We may decide designing a concrete syntax that is based on NL to instantiate the meta-model, so that project managers can create their project plans using the terminology they are used to. For instance, projects may be configured using sentences like the following: *"the project has two task units starting the 1st of April and the 1st of May"*, *"task t1 follows task t2"*, *"Peter Parker will participate in the first task"*, or *"the task t1 requires 2 personal computers"*.

To help in the creation of project plans, there will be a dedicated chatbot that aids managers in completing any missing data and refining the meaning of ambiguous user sentences. For example, in the first sentence (*"the project has two task units..."*), the chatbot would need to ask the user about the kind of task units to create. Since the sentence includes dates, candidate classes are `Milestone` and `Task` as both define a date, but not `CompositeTask` which has none. In addition, the chatbot would ask the user the `id` of the created task units, as it is a mandatory feature in the meta-model. This way, models of project plans would be iteratively built by means of a conversation between the user and the chatbot.

Our aim is automating the creation of this kind of modelling chatbots. As we will see in the following sections, this requires specifying and customizing several aspects of the NL-based concrete syntax such as the identifier to be used to refer to objects (e.g.,

`name` and `surname` for human resources, or `id` for task units); the level of conformance required from models, which in the stricter case would make the chatbot request the user a value for any mandatory field of new objects; synonyms for the class and field names (e.g., using the verb *to follow* as an alternative to reference `next`); or whether the objects of a certain type should not be retrieved from the model being constructed but from an external resource, like a data base or an external API. In our example, available human resources are stored in a company database, and hence they need to be retrieved and the model populated with them when the model is created.

In the next section, we describe the building blocks of chatbot specifications, to which we will map the different elements of domain meta-models.

3 DEVELOPING CHATBOTS WITH DIALOGFLOW

Chatbots are software programs with a NL user interface. They are typically accessible through social networks (e.g., Slack, Telegram, Facebook messenger) and can be used in mobility without the need to install new apps. Chatbots emulate the interaction with a human assistant, and are becoming very popular for customer support, marketing, or access to services like bookings, food delivery and gamification-based learning.

Many dedicated frameworks to create chatbots are emerging, like DialogFlow, the IBM Watson Assistant, or the Microsoft Bot Framework. As a representative, this section describes the main concepts of DialogFlow. This provides a cloud-based development environment to describe chatbots with voice and text-based conversational interfaces, and it offers support for NL processing in more than 20 languages. Our choice is motivated by its automated support for deploying the bot in many different social networks, its flexibility to link external services (which do not need to be deployed on specific clouds, like Azure, hence avoiding vendor lock-in), and the possibility to define the chatbot using a JSON specification (in addition to using the cloud development environment) which facilitates chatbot synthesis.

Figure 2 shows the simplified working scheme of DialogFlow's chatbots. These are called *agents* and define behaviour by means of *intents*. Each intent represents some user's aim (e.g., booking a ticket). The agent waits for user inputs in the form of NL sentences (label 1 in the figure). Then, it tries to match the user input with some available intent (label 2), optionally calling an external service (also called *fulfillment*, label 3). Finally, the agent produces a response, typically a NL sentence among a predefined set (label 4).

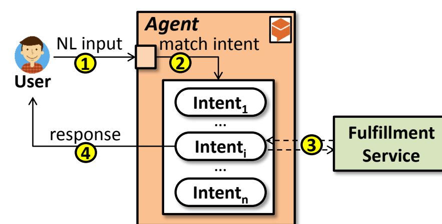


Figure 2 – Agent working scheme.

Figure 3 shows a meta-model we have created for DialogFlow. As it can be seen, *agents* define *intents*, which are configured with a set of phrases that are used to train a NL processor. An intent also declares a set of responses that the agent answers when the user inputs a NL sentence that matches the intent. In addition, a *fallback* intent is usually available for the case when no other intent is matched, with a predefined set of responses which typically show the user the available alternatives.

Intents may have zero or more *followup* intents that can only be activated right after the parent intent has been activated. Intents can also define *contexts*. These represent the current state of a user's request and allow the agent to carry information from one intent to another (i.e., a followup intent). Input and output contexts, together

Figure 3 – A meta-model for DialogFlow.

with the followup relations, are used to control the conversational path the user takes through the dialogue with the agent.

Entities are the mechanism to identify and extract data from the users' NL inputs. For this purpose, an agent can define *entity types* (e.g., vegetables) which provide a list of *entries* of admissible values (e.g., scallion) and their synonyms (e.g., green onion).

Intents save these data in *parameters* that refer to an entity type and can take one of its entries as their value. The *values* are obtained from the user input according to the NL patterns extracted during the training phase, and the agent can use these values in its responses. If a parameter is *required* but the user input does not include a value for it, then the agent *prompts* the user for a specific value. For instance, in the running example, we may define an intent to create `TaskUnits`; however, if the user does not state the specific type of `TaskUnit` (`Milestone`, `Task` or `CompositeTask`), or does not specify a value for some of its mandatory attributes (e.g., `id`), the chatbot will prompt the user, asking the required information.

Besides responding to users, agents can send the information gathered by an intent to an external service by enabling a *webhook*. This allows the chatbot to do complex tasks, like booking a ticket. The configuration of external services is defined by a *fulfillment*. If an intent declares *actions*, these will be sent to the service declared in the fulfillment. In our approach, we will use the chatbot as a conversational frontend for the model concrete syntax, while the model abstract syntax will be created and modified in an external service.

Finally, intents can also be triggered by *events*, which depend on the particular deployment platform. For example, a chatbot in Telegram can display buttons, and so it is possible to activate an intent upon clicking on these buttons.

4 CONVERSATIONAL SYNTAX FOR DSLs

To simplify the creation of modelling chatbots, we propose an automated process, depicted in Figure 4.

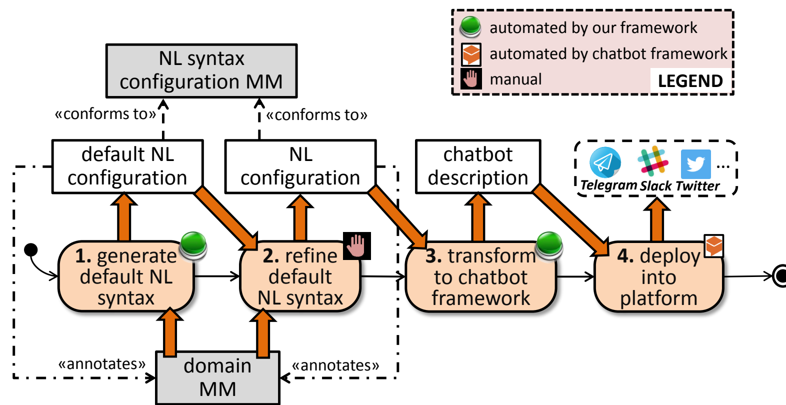


Figure 4 – Steps for creating a modelling chatbot with our approach.

As usual in MDE, we rely on a domain meta-model to describe the abstract syntax of the DSL. With regards to its concrete syntax, we rely on a meta-model to define the conversational syntax, similarly to when it is graphical or textual. To facilitate this definition, first, we automatically derive a default configuration of the NL syntax from the domain meta-model. This configuration declares how to refer to objects and features of the instantiable classes, and the level of tolerable inconsistency allowed during the modelling process. The latter is useful to enable more flexible modelling by relaxing the need for models to be fully compliant with their meta-model at all times, as this may interfere with the modelling/conversation flow [RdLP17, GdL18b]. Next, in a second step, the language designer may refine the default NL concrete syntax description, e.g., to include synonyms for the name of classes and features, or to declare that some classes are non-instantiable.

Once the conversational syntax is ready, our framework synthesizes a chatbot description from it, and subsequently, deploys the chatbot into a platform (e.g., Telegram, Slack or Twitter). Currently, the chatbot description follows the DialogFlow structure presented in the previous section, and the deployment platforms are those supported by DialogFlow. Our approach can be adapted to work with other chatbot frameworks that provide similar concepts. As we will see in Section 5, the deployed chatbot interacts with a modelling service we have created to handle the model modifications at the abstract syntax level (e.g., object creation and deletion).

In the following, Section 4.1 presents our meta-model to describe the NL concrete syntax, and Section 4.2 shows the mapping of NL syntax models into DialogFlow's chatbot descriptions.

4.1 Configuring the NL concrete syntax

Figure 5 shows our meta-model to configure the conversational NL syntax of DSLs. Some of its classes contain references to the domain meta-model elements they define the syntax for. Since we assume the Eclipse Modeling Framework (EMF) [SBPM09] as meta-modelling technology, the classes in our NL syntax configuration meta-model refer to the `EPackages`, `EClasses`, `EAttributes` and `EReferences` in the domain meta-model. However, our approach is easily adaptable to other meta-modelling frameworks.

`NLModel` is the root class. It contains one `NLClass` for each domain meta-model

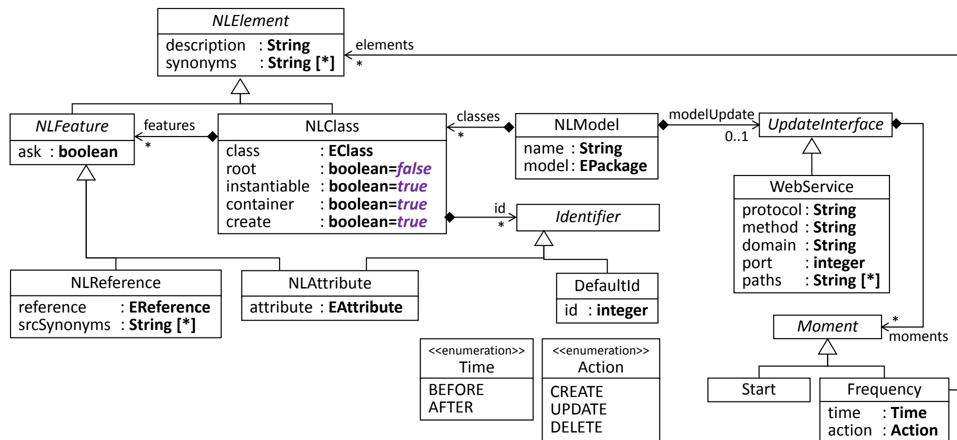


Figure 5 – The meta-model for configuring the NL syntax.

class, to configure its concrete syntax. The configuration includes a description of the class, a list of synonyms (usually nouns) of the class name, flags to indicate whether the class is `root` or `instantiable`, and one or more `Identifiers` that will be used to refer to the objects of the class. An object identifier may consist of one or more attributes of its class, or be a `DefaultId` which takes values from a counter. Two additional flags provide flexibility in the way objects of a class are to be created: `container` permits customising whether users should always indicate a container object for the new instances of a class (otherwise, the objects would be added to a virtual temporary container); and `create`, to indicate whether any object mentioned by the user should be automatically created in case the object does not exist (otherwise, the chatbot would just inform the user that the object does not exist).

`NLClass`es contain one `NLFeature` for each feature of the associated domain meta-model class. `NLFeature`s have a flag `ask` to make the bot ask for the feature value when a new instance of the class is created. By default, this attribute is true for mandatory features, and false for optional ones, though this can be modified. `NLFeature`s have a description and a list of synonyms, usually nouns for attributes and verbs for references. In addition, references can define additional synonyms to refer to their source end, which in the running example would permit using the sentences *"task t1 is next to task t2"* and *"task t2 is previous to task t1"* interchangeably.

Finally, in addition to the creation of objects using NL sentences, we also support the retrieval of external objects through `WebServices`. For this purpose, it is necessary to specify the `protocol`, `method`, `domain`, `port` and `paths` of the web service; and to configure the `Moments` in which these requests are made: either when the model is created, or *before*/*after* the *creation*/*update*/*deletion* of certain model elements.

Given the domain meta-model of a DSL, we automatically produce a default NL configuration model. This contains one `NLClass` for each domain class, and one `NLAttribute` or `NLReference` for each attribute and reference of the classes. The `NLClass` corresponding to the domain class that can reach more classes directly or indirectly through containment relations, is marked as `root`. Abstract classes are marked as non-instantiable, and concrete classes as instantiable. By default, the NL is configured to require a container for each new object (`NLClass.container = true`), alluding to non-existing objects implies their automated creation (`NLClass.create =`

`true`), and the chatbot will ask a value for any feature with cardinality greater than zero (`NLFeature.ask = true`). If a domain class has an attribute called *"name"*, *"id"* or *"identifier"*, this is assigned as the class identifier; otherwise, the class is assigned a default counter-based identifier.

Example. Figure 6 shows on the left an excerpt of the NL syntax model generated by default for the running example. Its elements refer to elements of the domain meta-model, which is shown on the right. The object `model` with type `NLModel` represents the model, and points to the `EPackage` containing the domain meta-model. The object `project` configures the syntax of class `Project`, which is the root class as it contains all other domain classes. The two `NLAttribute` objects specify the syntax of the attributes of `Project`. Attribute `name` is identified as the class `id`. The bot will ask a value for `name` as it is a mandatory attribute, but not for `goal` as it is non-mandatory.

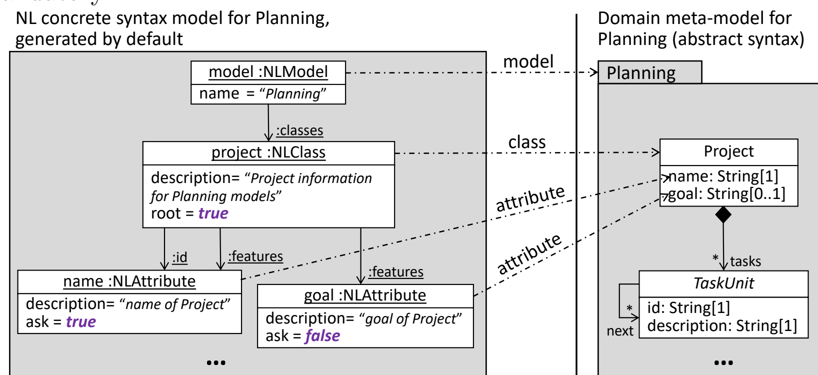


Figure 6 – Excerpt of default NL concrete syntax model for the running example.

The language engineer can refine the generated NL concrete syntax model, e.g., to change the default root class, to set a concrete class to non-instantiable (abstract classes must remain non-instantiable), to change the default identifiers assigned to classes, or to define a list of synonyms for class and feature names if so desired. We assign a generic description to elements (like *"Project information for Planning models"* in object `project`), which typically need to be refined as well. Finally, it is possible to configure an update interface using a web service, together with its application policy (i.e., when to obtain the information from the service).

Example. In our running example, the language designer would set class `Human` as non-instantiable, as human resources are to be gathered from a resource database (an external service). The model will be populated with `Human` objects upon creating the model (`Start`). The designer also needs to refine the identifiers of classes (e.g., the identifier of `Humans` is made of both attributes `name` and `surname`), and set synonyms to refer to some classes (e.g., *Activity* and *Job* for *Task*), references (e.g., *follow* and *subsequent* for *next*) and source end of references (e.g., *precede* for *next*).

4.2 Mapping NL syntax models into a chatbot framework

Starting from the refined NL syntax configuration model, we generate a chatbot description model conformant to the DialogFlow meta-model in Figure 3.

Figure 7 shows a high-level scheme of the generated chatbots, omitting the definition of parameters for simplicity. The chatbots rely on an external service (the `modellingBot Fulfillment`) to perform the model modifications at the abstract syntax level.

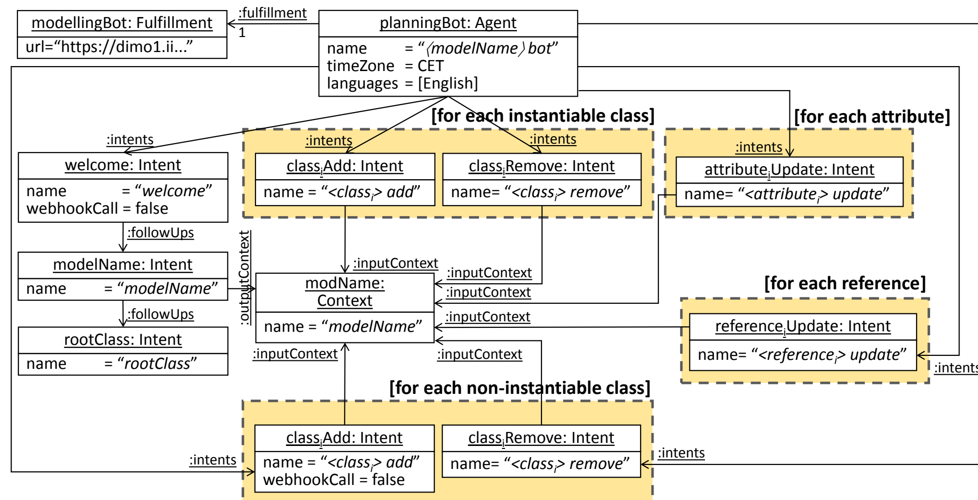


Figure 7 – Scheme of the generated DialogFlow chatbots.

Welcome intent. Each chatbot contains a *welcome* intent that is trained with typical greeting phrases (e.g., *"hello"*, *"hi"*, *"hey"*, *"hi there"*...). Welcome events of certain social networks (e.g., the `/start` command in Telegram) can trigger the welcome intent as well. The chatbot responds to this intent by introducing itself and the actions it can do. This information is extracted from the element descriptions in the NL syntax model. Then, the chatbot asks for the name of the model the users are going to work with. The answer is collected by a followup intent called *modelName*. This intent has a parameter with entity type *any*, meaning that it can receive anything, and it has the webhook enabled to invoke the REST web service indicated in the fulfillment URL in order to check if the model exists. If it does, it is not necessary to configure anything else; otherwise, a new model is created, and the chatbot uses the followup intent *rootClass* to ask the value of all the `NLFeatures` with attribute `ask=true` of the root class.

Object creation intents. The chatbot has several intents to recognise model editing actions, which become available only after the *welcome* and *modelName* intents have been triggered. The model update intents have the output context of the *modelName* intent as their input context, as Figure 7 shows.

Specifically, we create two intents for each instantiable class, one to create instances of the class and the other to remove them. The training phrases for the intents are automatically generated according to regular expression templates that combine the element names and synonyms specified in the NL syntax model.

Listing 1 shows the template used to synthesize training phrases for creating objects of a class and initializing their features. In the template, ⟨**create**⟩ represents the set of words or expressions that indicate the intention to create something. These include *"there is/are"*, *"I want to create"*, *"add"*, *"create"*, *"the model has"*, etc. Using one of these creation expressions is optional. ⟨**natural-number**⟩ can be optionally used to

indicate the number of objects to create. ⟨**class-name**⟩ stands for the class name and its synonyms specified in the `NLClass`. Next, the user can optionally assign values to the object's features. This way, ⟨**feature-name**⟩ corresponds to the feature name and synonyms specified in the `NLFeature`s of the `NLClass`, including the ids of the class; and ⟨**feature-value**⟩ defines samples of possible feature values (nouns for attributes with type String, integer numbers for attributes with type Integer, and so on). We do not take into account the meta-model integrity constraints for generating these sample values, as it is not required to train the NL processor. Instead, correctness of values is checked at runtime at the abstract syntax level by the modelling service.

```
1 <create>? <natural-number>? <class-name>
2    ( with <feature-name> <feature-value>+ ( (, | and) <feature-name> <feature-value>+ )* )?
```

Listing 1 – Template to synthesize training NL phrases for creating objects.

Example. Some training phrases of the creation intent derived from the `NLClass` `Task` are: *"I want to create one task"*, and *"add two tasks with id t1 and id t2"*.

Object creation intents have one parameter for each `NLFeature` in the `NLClass`, and one additional parameter accounts for the object container. The parameter names are equal to the feature names, and the chatbot will ask for the feature value if the NL syntax model defines so. In the case of `NLAttribute`s, the type of the parameter depends on the attribute's type, while in the case of `NLReference`s, it is the identifier of the reference target class. Table 1 shows the mapping between attribute primitive types and DialogFlow entity types. In addition, we create a custom-made `EntityType` to represent booleans. This defines two `Entries`: true and false. The former entry has affirmations as synonyms (*"yes"*, *"that's right"*, *"okay"*, *"sure"*...), and the latter negations (*"not"*, *"nah"*, *"don't"*, *"not really"*...). We do so because, when asking a value for boolean parameters, the answers typically have this form.

Primitive type	Entity type
String	sys.any
Integer/Long	sys.number-integer
Double/Float	sys.number
Date	sys.date-time
Boolean	boolean

Table 1 – Mapping primitive types into DialogFlow entity types.

The object creation intents have the webhook enabled. Hence, when all data is collected, the information is sent to the external modelling service to create the object.

Object deletion intents. We use the template in Listing 2 to synthesize training phrases for the intents that take care of deleting objects of the instantiable classes. ⟨**remove**⟩ represents the set of words or expressions indicating the intention to delete something (e.g., *"delete"*, *"remove*, *"erase*...); ⟨**class-name**⟩ is the name of its class or a synonym; and ⟨**id-value**⟩ represents the value of the object's identifier. These intents define one required parameter for the value of the object identifier, and its type is given by the mapping in Table 1. They also have the webhook enabled to trigger the object deletion by means of the modelling service.

```
1 <remove> ( <class-name> (with (id | <id-name>))? )? <id-value>
```

Listing 2 – Template to synthesize training NL phrases for removing objects.

Example. Some training phrases for the deletion of instances of the `NLClass` `Task` are: *"delete t1"*, *"remove task t1"*, and *"erase the task with id t2"*.

Feature modification intents. Starting from each `NLFeature`, we create an intent to modify its value. We handle attributes and references in a different way. Listing 3 shows three of the templates we use to generate training phrases for attribute modification intents. In these templates, ⟨**att-name**⟩ corresponds to the name and synonyms of the attribute to be updated; ⟨**att-value**⟩ is its new value; ⟨**update**⟩ represents the words to express the intent to modify something (e.g., *"update"*, *"modify*, *"set"*, *"change"*...); ⟨**id-name**⟩ is the name and synonyms used to refer to the identifier of a class; and the rest of the elements have the same meaning as before. The intent has three parameters: the new attribute value (⟨**att-value**⟩), the identifier of the attribute's owner object (⟨**id-value**⟩), and the class of the attribute's owner object (⟨**class-name**⟩ with entity type sys.any). The first two parameters are mandatory, while the third one is required only if there is more than one attribute with that name in the model. These intents have the webhook enabled and trigger the update of the attribute value.

```
1 <att-name> of <class-name>? <id-value> (is | are) <att-value>
2 <id-value>('s)? <att-name> is <att-value>
3 <update> <att-name> of (<class-name> (with <id-name>)?)? <id-value> to <att-value>
```

Listing 3 – Templates to synthesize training NL phrases for updating attribute values.

Example. Some phrases that fit in the attribute modification intent are: *"the units of technical pcs are 4"*, *"Peter's surname is Parker"*, and *"set date of t2 to May 24th"*.

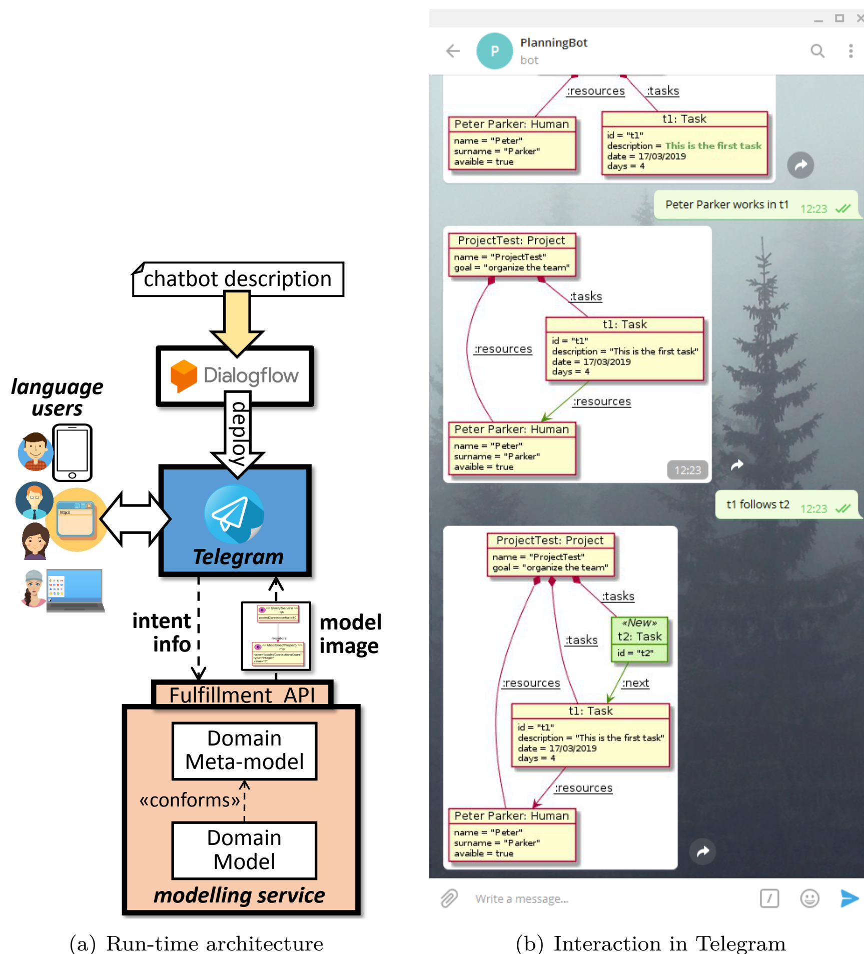
Regarding references, if their name is a noun, then we generate the training phrases using the templates for attribute modification in Listing 3. However, when their name is a verb, we use the two templates in Listing 4. In these templates, ⟨**ref-name**⟩ is the name and synonyms of the reference to be updated; ⟨**class-name**⟩ is the owner class of the reference; ⟨**ref-value**⟩ is the id of the object to be set in the reference; ⟨**ref-class-name**⟩ is the target class of the reference; and ⟨**ref-src**⟩ is the set of names used to refer to the source end of the reference. These intents have four parameters: the names of the reference source and target classes (with entity type sys.any, not needed if the reference name is unique in the model), and the identifiers of the source and target objects (mandatory). The webhook of the intents is enabled.

```
1 <class-name>? <id-value> <ref-name> <natural-number>? <ref-class-name>? <ref-value>
2 <ref-class-name>? <ref-value> <ref-src> <class-name>? <id-value>
```

Listing 4 – Templates to synthesize training NL phrases for updating reference values.

Example. Some examples for this intent are: *"Peter participates in task t2"*, and *"task t2 follows task t1"*.

Non-instantiable classes. Finally, we create two intents for each non-instantiable class with instantiable children, one for object creation and another for object deletion. The former is trained with sentences obtained from the object creation template in Listing 1, but in this case, the intent has no parameters, and the webhook is disabled. Instead, the chatbot asks the user to select one instantiable children of the non-instantiable class, and redirects the flow to the intent to create the selected class. The intents for deleting objects of non-instantiable classes work the same as the ones for instantiable ones, though in case there are several children with the same identifier, then the chatbot asks to select one of them to disambiguate.



(a) Run-time architecture (b) Interaction in Telegram

Figure 8 – Modelling chatbots.

Example. Our method generates 35 intents, 108 parameters and 2600 training phrases for the running example (in average, 74 training phrases and 3 parameters per intent). Without our method, this information would need to be created manually.

5 TOOL SUPPORT

We have developed prototype tool support for automating the creation of modelling chatbots. Our solution includes an EMF implementation of the meta-model in Figure 5 for configuring the NL syntax, an Eclipse plugin that instantiates this meta-model for a given domain meta-model, and a transformer into DialogFlow.

Figure 8(a) shows the runtime architecture of our generated chatbots. They can be deployed on social networks, like Telegram in the figure. This enables collaborative modelling as discussions among the language users and model update indications integrate seamlessly, because both happen within the chat. Moreover, since social networks typically provide different clients (e.g., for mobile devices, desktop computers or web browsers) we obtain multi-platform modelling for free.

When the chatbot matches an intent with the webhook enabled, it sends a request to a modelling service that we have developed. The request contains a JSON with the user text message, the social network, and the content of the intent (name, context, parameters, etc.). The service processes the request and makes the necessary modifications in the abstract syntax of the model. Next, the service sends back to the social network an image of the updated model created with PlantUML [Pla19]. The image highlights the elements that have been modified in green. The `validate` command shows possible inconsistencies in the model, which then can be corrected by the users.

Example. Figure 8(b) illustrates the interaction with the chatbot for creating project plans. The user first inputs the sentence *"Peter Parker works in t1"*. Since we have configured the NL syntax to accept *work* to refer to the source end of reference `resources`, the chatbot creates a link with this type between the `Human` object with identifier *Peter Parker* (`name` and `surname`) and the `Task` object with identifier *t1* (`id`). Then, the user inputs the sentence *"t1 follows t2"*, which triggers the creation of a link with type `next` as *follows* is a synonym for the source reference end. Moreover, the chatbot creates a new task with identifier *t2* as the source of the link because it does not exist in the model. A video illustrating these interactions is available at `https://saraperezsoler.github.io/ModellingBot/`.

6 CASE STUDY

In this section, we use our approach to develop a conversational front-end for Datalyzer [GdL18a]. Datalyzer is a cloud system, based on a graphical DSL, to develop streaming data applications and execute them on the cloud. The goal of this case study is to answer the following research questions:

RQ1 Is it feasible to create a NL front-end for an existing DSL-based information system?

RQ1.1 What are the steps that require manual programming?

RQ2 What is the added value – in terms of functionality – that a modelling chatbot brings?

In the following, Section 6.1 introduces Datalyzer; then, Section 6.2 describes its abstract syntax, and Section 6.3 its new conversational syntax; Section 6.4 details the integration of Datalyzer and the chatbot; and Section 6.5 discusses the benefits of the approach.

6.1 Datalyzer

Datalyzer [GdL18a] is an open web platform that generates and executes data streaming applications in a simple and intuitive way using MDE techniques. The data applications can be connected to several heterogeneous data sources. They generate a data output stream which can be connected with external services and be visualized on a dashboard as charts, tables or other interactive elements in real time.

Datalyzer can be used in two ways: to build services that transform data on the cloud, or to build complete data monitoring applications using the dashboard. The

applications are modelled using a graphical DSL developed in Javascript. The left of Figure 9 shows the DSL editor of Datalyzer with a simple application model that we will use as an example. The model collects streaming data from Twitter, filters the tweets by a set of keywords (*"London"* in the figure), puts the data into a pipeline, and displays the tweets in a table on the dashboard. The right of Figure 9 shows the generated dashboard for the example.

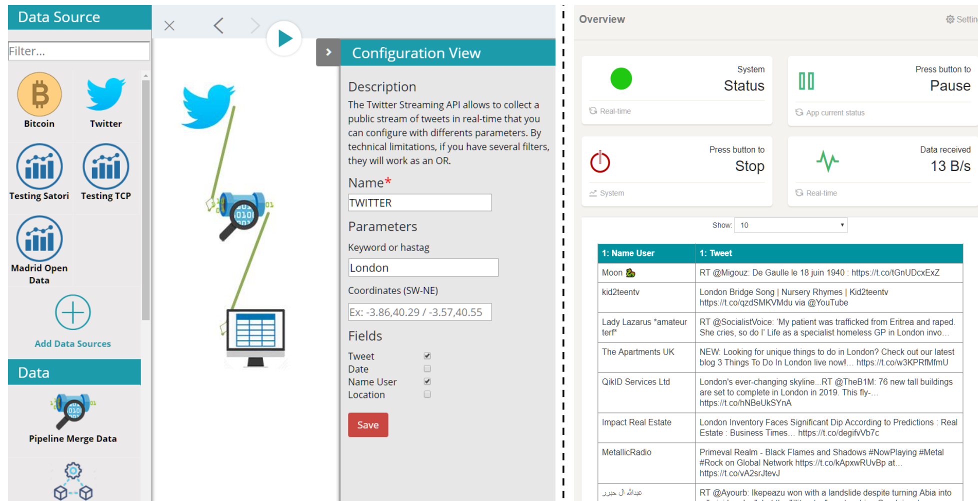


Figure 9 – Example designed using Datalyzer's DSL (left). Generated dashboard (right).

We would like to complement Datalyzer with a chatbot that enables the collaborative construction of data application models using conversation on social networks. This is a challenging, realistic case study for our approach for two reasons. First, the chatbot would become a NL front-end for an existing information system, and therefore, needs to integrate not only modelling with Datalyzer's DSL, but also with commands like saving a project or running the application. Second, data sources (e.g., Twitter, Bitcoin market values, or Madrid traffic data) in the application models are non-instantiable but should be retrieved from a database.

6.2 Domain meta-model

Figure 10 shows the meta-model to describe Datalyzer applications by instantiating and connecting different types of primitives. `DataSourceInstance` represents an instance of a `DataSourceType`. Data source types are created and maintained in an external database, and include descriptions of services like openWeather, the Twitter API, open data APIs (e.g., the Madrid traffic data), or connections using sockets (e.g., to sensor data streams). Data source instances may provide values to required configuration parameters, as well as to a selection of the fields to be received (omitted in the figure for simplicity). For example, a data source instance for Twitter requires specifying filtering keywords (e.g., hashtags), an optional location and an authentication. For reception, we may be interested in the user name and the tweet text.

Data source instances are connected to at least one `DataPipeline`. There are two types of pipelines: a `Basic` one and a `Join` pipeline that merges data from multiple sources. Pipelines can be connected to other pipelines or to `DataProcessors`. The latter define data processing operations like filters and transformers. `IntermediateNodes`

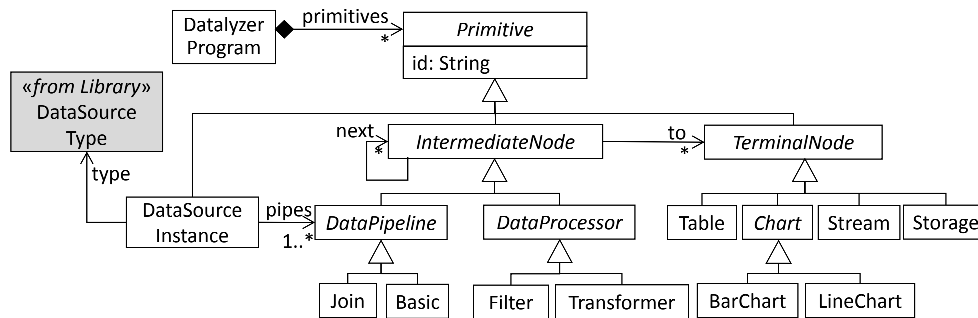


Figure 10 – Datalyzer meta-model excerpt.

(i.e., pipelines and data processors) can be connected to `TerminalNode`s which implement features such as storing data or displaying data in charts.

As mentioned above, `DataSourceType`s are not explicitly created by the user, but read from an external database. Hence, we need to prepare a special NL syntax configuration for the chatbot, as the next section explains.

6.3 Configuration of the NL concrete syntax

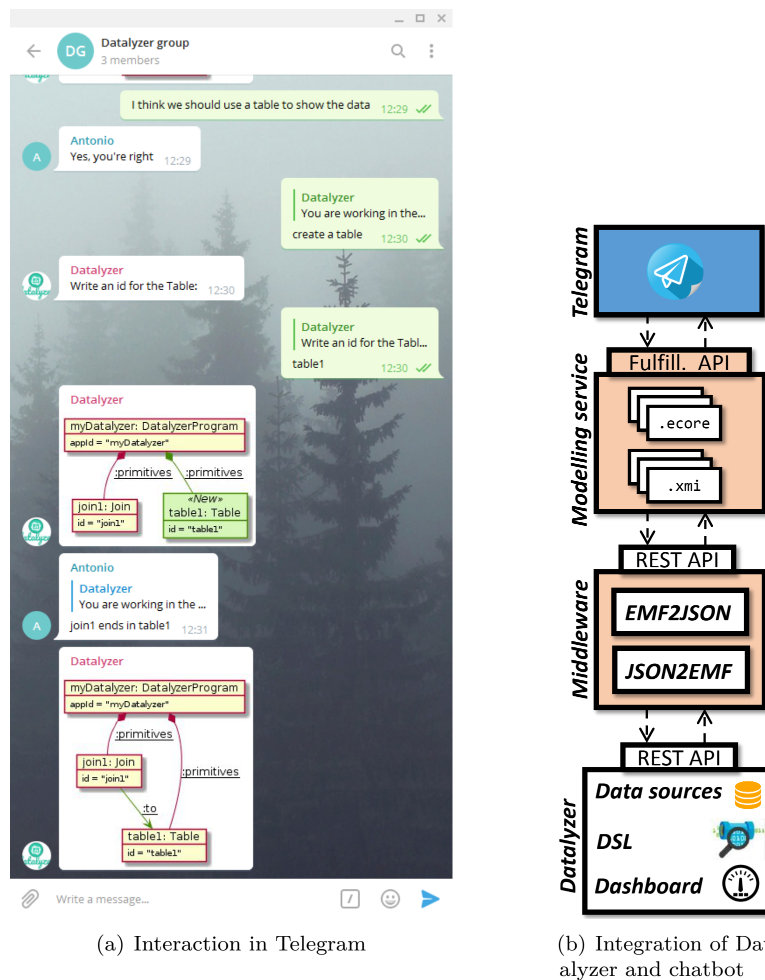
We used our approach to automatically generate a default NL concrete syntax model from the Datalyzer meta-model. In this model, the `NLClass` pointing to `DatalyzerProgram` was correctly identified as root, and all non-abstract domain classes were set to instantiable.

Next, we manually refined the NL model to add synonyms. For instance, we added the synonyms *"basic pipeline"* and *"basic pipe"* for class `Basic`, and *"data source"* and *"instance"* for class `DataSourceInstance`. In addition, we modified the `NLClass` pointing to `DataSourceType` to make it non-instantiable because its objects are stored in an external library, and created a `WebService` that reads those objects upon creating or loading a Datalyzer model (`Start`). The web service describes a REST API with http `method`, the url as `domain`, and `port` 8080.

Starting from the modified concrete syntax model, we produced a DialogFlow chatbot that is able to process sentences like *"create data source Twitter with keyword London"*, or *"connect the pipeline to table1"*. Figure 11(a) shows the interaction with the chatbot. The first two messages correspond to a discussion between two users about the application they are modelling. Then, one user addresses the chatbot to *"create a table"*, the chatbot asks for its identifier as it is mandatory, the user answers *"table1"*, and a new table is created.

6.4 Integration of Datalyzer and the chatbot

Datalyzer can be used as a web service via a REST API. This way, external systems can receive data from applications running on Datalyzer and perform some actions such as executing or stopping a project. However, this API did not support creating Datalyzer models, but this was only possible on a web browser. Hence, we created a middleware providing model management support (e.g., uploading Datalyzer models) and supporting all commands available in the browser. Figure 11(b) shows the resulting architecture that connects Datalyzer and the chatbot. The middleware is connected to the chatbot as a REST API, and implements the following functionalities:



(a) Interaction in Telegram

(b) Integration of Datalyzer and chatbot

Figure 11 – Modelling chatbot for Datalyzer streaming data applications.

- **Model transformation.** Datalyzer is a cloud application. For this reason, it does not use EMF models but non-standard JSON models that can be processed in Javascript. To make JSON models compatible with the chatbot's modelling service, we have developed two transformers, from EMF to JSON and back.
- **Model updates.** The chatbot sends requests to the middleware to obtain the data source types, as these are instances of a non-instantiable class. In its turn, the middleware retrieves the data source types by sending a request to the Datalyzer REST API, and invokes the transformer to convert the JSON data into EMF models that the modelling service can process.
- **Service encapsulation.** The chatbot performs some actions implicitly. When the middleware receives a petition requesting the data source types, it means that a new model is being created in the chatbot. This triggers the creation of a Datalyzer project associated to a generic and public user. A similar process is done when the chatbot sends the application model to the middleware: the model is saved in the database, the application is generated and executed, and the

middleware sends the link of the dashboard to the chatbot.

6.5 Discussion

Next, we answer the research questions, and discuss limitations.

RQ1: Feasibility. This question can be answered positively: using our approach, we easily added a NL interface to an existing information system through Telegram.

RQ1.1: Automation. Configuring the NL syntax was easy as it is highly automated. From the meta-model of Datalyzer, the approach automatically generated 40 intents and 2500 training phrases that otherwise should have been defined manually. However, we had to implement the middleware that connects Datalyzer and the chatbot to bridge their different modelling technologies (JSON and EMF).

RQ2: Added value. Social networks are common in our lives, and we are familiar with their interaction style. Hence, some users may find modelling using NL and a conversational assistant easier or more appealing than learning to use a graphical or textual DSL and its editing environment. Moreover, "chatbot-izing" Datalyzer has expanded its capabilities as follows:

(i) As the chatbot is integrated into Telegram, it is possible to use the collaborative capabilities of this social network, e.g., to build Datalyzer models collaboratively, intertwine discussion messages and editing actions in real time and trace them back in the chat history, organize private or public on-line meetings, invite collaborators to existing projects, etc. These features were not initially available in Datalyzer.

(ii) Telegram can be installed on smartphones, tablets and computers, and there is a web version as well. Hence, we can use the Datalyzer chatbot from *any* device regardless of the OS, and from *many* devices at the same time as they remain synchronized by a personal account. This makes Datalyzer portable and permits using it in mobility. Although Datalyzer is a web platform, its interface is not as well adapted to phones and tablets as Telegram.

Limitations. While we produce fully-functional chatbots trained with sensible NL phrases, evaluating the completeness of these phrases or the efficacy of the generated conversational flow is something that we plan to assess in the near future. Also, the chatbot uses a default concrete syntax (object diagrams) in the images, instead of the concrete syntax for the DSL supported by Datalyzer. We plan to improve this support in future work.

7 RELATED WORK

In this section, we revise related works on chatbot creation frameworks, the usage of bots or NL processing techniques within MDE, and collaborative modelling.

Chatbot frameworks. In this paper, we synthesize chatbots using the DialogFlow chatbot creation framework. Our decision is motivated by its popularity, high degree of customizability, support for NL processing, and the possibility to integrate the chatbot with external services (a modelling service in our case) via a REST API. In addition to a cloud-based chatbot editor, DialogFlow also supports uploading chatbot

descriptions in JSON. However, we may have used other frameworks (see [LSZ18] for a survey). In the following, we revise some of the most popular ones.

The IBM Watson Assistant [IBM19] allows building conversational interfaces. As in DialogFlow, intents and entities can be used to train a machine learning model that will understand similar NL requests from users. Hence, adapting our approach to this framework would be easy. It provides an SDK to build applications around chatbots, but integrating the chatbots into social networks is less direct than in DialogFlow.

The Microsoft Bot Framework [Mic19] permits building and deploying chatbots in websites and social networks. Its main components are the channel connectors, to connect chatbots to messaging channels, and the BotBuilder SDK, to implement the business logic and integrate NL understanding services. It offers some advanced cognitive services like image-processing algorithms and recommending services.

Amazon Lex [Ama19] is a service to create conversational interfaces, with support for NL processing (i.e., it extracts a NL model from training sentences). FlowXo [Flo19] permits creating conversational flows by connecting triggers to actions. The framework provides over 100 integrations, most of which can trigger a flow or be the output action of a flow. These include utility modules (e.g., webhooks or email) and integration with third-party services (e.g., Github or Google Sheets). Unlike DialogFlow, it does not provide support for NL processing.

Chatbots are created in Landbot.io [Lan19] by visually linking blocks and messages. Extra functionality can be coded using a built-in development tool, or integrating external services using a REST API (like in DialogFlow). It does not integrate artificial intelligence intentionally, as it advocates simplicity as its main feature.

Bots and NL processing in MDE. Our work proposes using NL as a particular kind of concrete syntax for DSLs. NL processing techniques have been used within Software Engineering to derive UML diagrams/domain models from text [ASBZ16, LKT14]. In this context, our contribution is to use an interactive incremental approach to building models, the use of social networks to embed assistance, and the generalization from UML models to arbitrary DSLs.

The ModelByVoice [LCA18] modelling tool supports voice recognition and speech synthesis for editing models. The tool assumes a diagrammatic concrete syntax for models, and editing actions are generic commands. For instance, creating any kind of object is done through the command "*create node*", after which the tool prompts the user about the node type and its attributes. The tool VoiceToModel [SAW15] is similar but for goal-oriented models, object models and feature models. Compared to ModelByVoice, it supports a smaller set of modelling languages, but their commands are less generic (e.g., there is a create command for each object type) though still rigid. In contrast, we generate a flexible NL syntax adapted to the DSL, support synonyms, the conversation flow is configurable, and do not assume a diagrammatic model.

In [PNFL17], the authors define a feature model with the commonalities and variations of chatbot features. Variability can come from the platform (e.g., Telegram or Slack), the way to access external services (e.g., via REST web service calls), the chatbot application core, the chatbot personality processing, and the dialog services. This feature model can be used as a reference framework to guide chatbot creation. While this work complements ours by focusing on the technical aspects of chatbot implementations, we are more concerned with the usage of NL as frontend of domain models, modelling services and information systems, and we provide tool support.

In the vision paper [CCBG18], the authors propose cognifying MDE to promote its adoption. Cognification is the application of knowledge extracted from existing

information, to boost a given process. Among other applications, the paper mentions the possibility of having modelling bots that suggest missing model properties based on the analysis of previous models in the same domain. Such facilities might be added to our bots as external services complementing our modelling service.

Collaborative modelling. Collaborative modelling has been used for model construction [GBR12] and collaborative creation of DSLs [IC16]. However, these works do not use social networks or NL processing, but they rely on collaborative graphical model editors [GBR12] or ad-hoc tools [IC16] without assistant support.

More recently, in [PGdL18], we embedded meta-modelling chatbots within social networks, to enable the collaborative creation of meta-models by domain and meta-modelling experts. The present paper follows this line of research, extending the use of chatbots for modelling using arbitrary DSLs, and not just for building meta-models. Moreover, we automate the creation of such domain-specific modelling chatbots.

Altogether, from the analysis of the state of the art, we conclude that the usage of NL as concrete syntax for domain-specific modelling languages, assisted by modelling chatbots that help in constructing models using a configurable conversational style, and being the frontend for modelling services, is highly novel.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a novel approach to define a conversational syntax for DSLs based on NL processing and chatbots. The approach is based on annotating domain meta-models with configuration information for the NL syntax, and translating these data into a chatbot creation framework (DialogFlow in our case). The chatbots can be deployed on platforms like Telegram, and use a modelling service to create the model abstract syntax at run-time. We have demonstrated the feasibility of our solution by means of a case study where we have created a modelling chatbot atop an existing cloud system to define and run streaming data applications. The case study illustrates the functionality added by the chatbot, which includes support for collaboration in NL, multi-platform, mobility, and traceability.

While our prototype tool demonstrates the feasibility of our proposal, evaluating the quality and usability of our generated chatbots still remains future work. Hence, in the near future, we plan to perform a usability study with users, as well as to apply existing quality frameworks for chatbots like [PD18a, PD18b]. We are currently extending our tooling with a full-fledged environment to edit the NL concrete syntax models. We are also currently working on integrating further services into our modelling chatbots, like code generators and model transformation engines deployed in the cloud, in order to provide a complete MDE solution interfaced by NL.

References

[Ama19] Amazon. Amazon lex. `https://aws.amazon.com/lex/`, 2019.

[ASBZ16] C. Arora, M. Sabetzadeh, L. C. Briand, and F. Zimmer. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proc. MoDELS*, pages 250–260. ACM, 2016.

[CCBG18] J. Cabot, R. Clarisó, M. Brambilla, and S. Gérard. Cognifying model-driven software engineering. In *Proc. STAF Collocated Workshops*, volume 10748 of *LNCS*, pages 154–160. Springer, 2018.

[DP14] J. Danado and F. Paternò. Puzzle: A mobile application development environment using a jigsaw metaphor. *J. Vis. Lang. Comput.*, 25(4):297–315, 2014.

[Flo19] FlowXO. Flow xo for chatbots. `https://flowxo.com/`, 2019.

[GBR12] J. Gallardo, C. Bravo, and M. A. Redondo. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications*, 35(3):1086–1105, 2012.

[GdL18a] M. González-Jiménez and J. de Lara. Datalyzer: Streaming data applications made easy. In *Proc. ICWE*, volume 10845 of *LNCS*, pages 420–429. Springer, 2018.

[GdL18b] E. Guerra and J. de Lara. On the quest for flexible modelling. In *Proc. MODELS*, pages 23–33. ACM, 2018.

[Goo19] Google. DialogFlow. `https://dialogflow.com/`, 2019.

[IBM19] IBM Watson Assistant. `https://www.ibm.com/cloud/watson-assistant/`, 2019.

[IC16] J. L. Cánovas Izquierdo and J. Cabot. Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science*, 2:e84, 2016.

[Jac12] J. A. Jacko. *Human Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications.* CRC Press, 3rd edition, 2012.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley, 2008.

[Lan19] Landbot.io. `https://landbot.io`, 2019.

[LCA18] J. Lopes, J. Cambeiro, and V. Amaral. ModelByVoice - towards a general purpose model editor for blind people. In *Proc. MODELS Workshops*, volume 2245 of *CEUR Workshop Proceedings*, pages 762–769. CEUR-WS.org, 2018.

[LKT14] M. Landhäußer, S. J. Körner, and W. F. Tichy. From requirements to UML models and back: How automatic processing of text can support requirements engineering. *Software Quality Journal*, 22(1):121–149, 2014.

[LSZ18] C. Lebeuf, M.-A. D. Storey, and A. Zagalsky. Software bots. *IEEE Software*, 35(1):18–23, 2018.

[Mic19] Microsoft Bot Framework. `https://dev.botframework.com/`, 2019.

[MNPP17] P. Markopoulos, J. Nichols, F. Paternò, and V. Pipek. Editorial: End-user development for the internet of things. *ACM Trans. Comput.-Hum. Interact.*, 24(2):9:1–9:3, 2017.

[Moo18] S. Moore. Gartner press release on chatbots. `http://tiny.cc/bkss7y`, 2018.

[PD18a] J. Pereira and O. Díaz. Chatbot dimensions that matter: Lessons from the trenches. In *Proc. ICWE*, volume 10845 of *LNCS*, pages 129–135. Springer, 2018.

[PD18b] J. Pereira and O. Díaz. A quality analysis of facebook messenger's most popular chatbots. In *Proc. SAC*, pages 2144–2150. ACM, 2018.

[PGdL18] S. Pérez-Soler, E. Guerra, and J. de Lara. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software*, 35(6):48–54, 2018.

[PGdLJ17] S. Pérez-Soler, E. Guerra, J. de Lara, and F. Jurado. The rise of the (modelling) bots: towards assisted modelling via social networks. In *Proc. ASE*, pages 723–728. IEEE Computer Society, 2017.

[Pla19] PlantUML. `http://plantuml.com/`, 2019.

[PNFL17] A. Di Prospero, N. Norouzi, M. Fokaefs, and M. Litoiu. Chatbots as assistants: an architectural framework. In *Proc. CASCON*, pages 76–86. IBM / ACM, 2017.

[PP09] M. Pfeiffer and J. Pichler. A DSM approach for end-user programming in the automation domain. In *Proc. INDIN*, pages 142–148, 2009.

[RdLP17] D. Di Ruscio, J. de Lara, and A. Pierantonio. Special issue on flexible model driven engineering. *Computer Languages, Systems & Structures*, 49:174–175, 2017.

[SAW15] F. Soares, J. Araújo, and F. Wanderley. VoiceToModel: an approach to generate requirements models from speech recognition mechanisms. In *Proc. SAC*, pages 1350–1357. ACM, 2015.

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2009.

[Sch06] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[VPGdL17] D. Vaquero-Melchor, J. Palomares, E. Guerra, and J. de Lara. Active domain-specific languages: Making every mobile user a modeller. In *Proc. MODELS*, pages 75–82. IEEE Comp. Soc., 2017.

About the authors

Sara Pérez-Soler is PhD student in the *miso* group of the Universidad Autónoma of Madrid. Contact her at `sara.perezs@uam.es`.

Mario González-Jiménez is MSc student in the *miso* group of the Universidad Autónoma of Madrid. Contact him at `mario.gonzalezj@uam.es`.

Esther Guerra is a professor at the Universidad Autónoma of Madrid. She leads the *miso* group together with Juan de Lara. Contact her at `Esther.Guerra@uam.es`.

Juan de Lara is a professor at Universidad Autónoma of Madrid. He leads the *miso* group together with Esther Guerra. Contact him at `Juan.deLara@uam.es`.

Acknowledgments Work funded by the R&D programme of the Madrid Region (S2018/TCS-4314) and the Spanish Ministry of Science (RTI2018-095255-B-I00). We thank the anonymous referees for their useful and encouraging comments.