Flexible modelling using conversational agents

Sara Pérez-Soler
Computer Science Department
Universidad Autónoma de Madrid
Madrid, Spain
sara.perezs@uam.es

Esther Guerra
Computer Science Department
Universidad Autónoma de Madrid
Madrid, Spain
esther.guerra@uam.es

Juan de Lara
Computer Science Department
Universidad Autónoma de Madrid
Madrid, Spain
juan.delara@uam.es

***Abstract*—The advances in natural language processing and the wide use of social networks have boosted the proliferation of *chatbots*. These are software services typically embedded within a social network, and which can be addressed using conversation through natural language. Many chatbots exist with different purposes, e.g., to book all kind of services, to automate software engineering tasks, or for customer support.**

In previous work, we proposed the use of chatbots for domain-specific modelling within social networks. In this short paper, we report on the needs for flexible modelling required by modelling using conversation. In particular, we propose a process of meta-model relaxation to make modelling more flexible, followed by correction steps to make the model conforming to its meta-model. The paper shows how this process is integrated within our conversational modelling framework, and illustrates the approach with an example.

***Index Terms*—Flexible Modelling; Conversational Agent; Natural Language Processing; Chatbots**

I. Introduction

Model-driven engineering (MDE) [1] uses models in all phases of software development. Models are usually built with a domain-specific language (DSL). DSLs are defined with an abstract syntax and a concrete syntax. The abstract syntax in MDE is defined with a meta-model, where the concepts of the domain are specified. The concrete syntax is usually graphical or textual. The creation of models is an activity that is not only performed by developers, but there are scenarios in which it is necessary to involve the end users, e.g., requirements modelling [2], to define touristic routes [3] or create IoT applications [4]. However, end users normally have low technical profiles and are not familiar with modelling tools or DSLs.

In recent years, the advances in natural language (NL) processing has boosted chatbots or conversational agents. These programs interact with the user in NL and are usually integrated into social networks. They are currently used to automate tasks such as customer support [5], shopping assistance [6], queries assistance [7] or education support [8]. Moreover, as social networks incorporate communication channels, chatbots are perfect for collaborative tasks. Due to the increase in the use of chatbots, several tools have emerged that facilitate their creation, e.g., Dialogflow from Google[1], IBM Watson[2] or Amazon Lex[3]. These frameworks offer an environment in the cloud, with a graphical interface that allows the user to configure the conversation flow of the chatbot. These frameworks work using machine learning to match the user message with an *intent*. For this process, the intent needs some training phrases and the key values or parameters collected from the phrases. Also, it is necessary to define a conversation flow, indicating the order of the intents.

In previous work [9], we proposed an approach to assign a conversational syntax to a DSL and generate a conversational agent from the DSL definition. This approach exploits the advantages of performing modelling tasks collaboratively using NL in social networks. Using NL to build models facilitates this activity to users unfamiliar with modelling. The use of social networks eliminates the need to install and learn to use a new tool for modelling. But making a conversational agent manually from a meta-model is a time-consuming and repetitive task that requires the design and creation of the NL interpreter and a modelling service to take care of the model creation. Therefore, we proposed to automate the task of designing and creating the agent, and using a dynamic modelling service based on a meta-model.

However, when using NL, people normally do not provide all the information in their phrases. Moreover, we want to let users express their ideas in a more free way, which can be refined later. For this reason, in this work, we present a flexible modelling approach – especially tailored to conversation-based interaction – which allows to save incomplete or incorrect information in a model, waiting for its later refinement.

The rest of this paper is organized as follows. Section II overviews the approach to automate the creation of conversational agents for modelling. Section III presents our approach to make modelling more flexible for chatbots. Section IV shows examples of flexible modelling for NL. Section V shows how the tool works in Telegram. Section VI compares our approach with related research, and Section VII concludes.

II. Creating modelling chatbots

In [9], we developed an approach to create modelling agents through NL in social networks, based on the DialogFlow framework. Specifically, starting from a domain meta-model, we automatically generate a conversational concrete syntax.

Work funded by the R&D programme of the Madrid Region (S2018/TCS-4314) and the Spanish Ministry of Science (RTI2018-095255-B-I00).

[1]https://dialogflow.com/

[2]https://www.ibm.com/watson

[3]https://aws.amazon.com/es/lex/

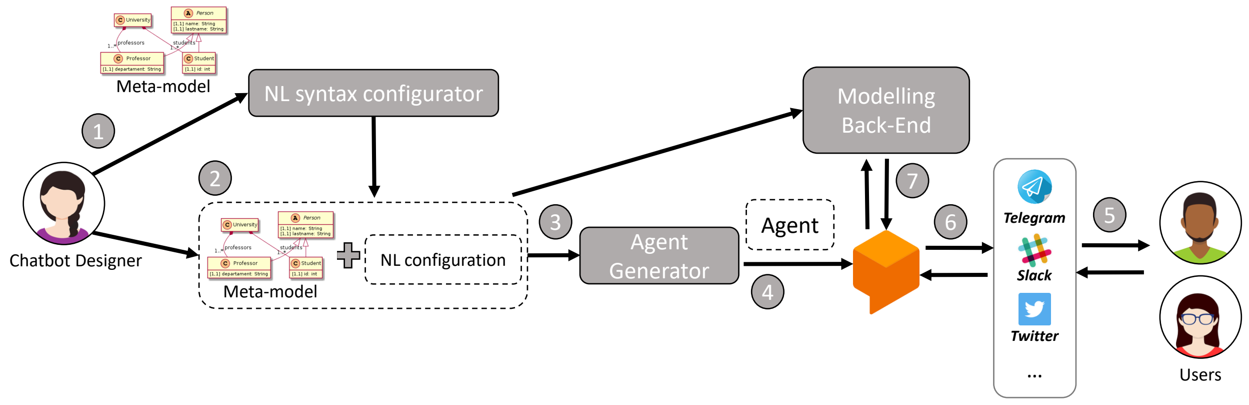


Fig. 1. Creation and use of the modelling agent

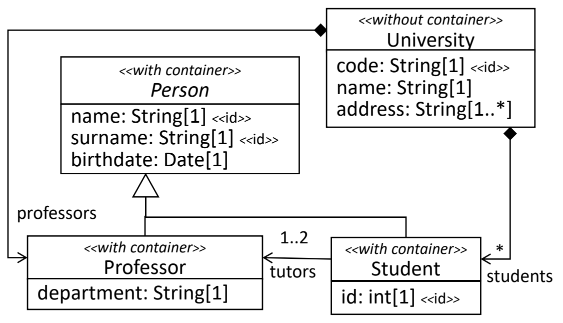


Fig. 2. University meta-model

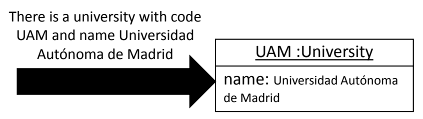

There is a university with code UAM and name Universidad Autónoma de Madrid
UAM :University
name: Universidad Autónoma de Madrid

Fig. 3. NL processing to create a University object

Figure 1 shows the creation process of a modelling agent with our approach. First, the chatbot designer must provide the domain meta-model (label 1). The NL syntax configurator automatically generates a configuration model that will be used to create the chatbot as well as the NL syntax to create models (label 2). This configuration can be later extended and modified by the designer, for example adding synonyms with which the user can refer to elements of the meta-model (classes, attributes and references). To create a model using conversation, it is necessary to be able to differentiate the objects; for this reason, this configuration needs to indicate the identifier attributes of each class. It is also possible to configure the instances of which classes can be outside of any container object and which ones can not, that is, which objects must be assigned to a container reference or which ones can be directly contained in the model. The designer of the chatbot typically reviews the configuration generated, adjusting it to the needs of the domain.

Once refined, the configuration and the domain meta-model are passed to an agent generator module (label 3). The agent generator generates the conversational flow, the intents, the training phrases and the parameters automatically, saving this job to the designer of the chatbot. Once the agent is ready, it is automatically deployed in Dialogflow (label 4) and users can interact with it through social networks (labels 5 and 6).

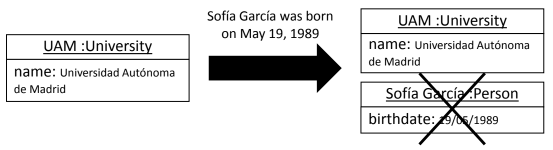

Sofía García was born on May 19, 1989
UAM :University
name: Universidad Autónoma de Madrid
UAM :University
name: Universidad Autónoma de Madrid
Sofía García :Person
birthdate: 19/05/1989

Fig. 4. Phrase that can not be handled due to the rigidity of the meta-model

Finally, there is a modelling back-end that transforms the user's intent into model actions (label 7). This back-end is the same for all the modelling agents, so it works generically and needs the meta-model and the configuration.

Figure 2 shows a meta-model and the configuration provided by the designer of the chatbot. The elements of configuration are represented with stereotypes, that is, they are between the symbols « and ». This example is a meta-model of a University. The *University* class has a *code*, which is an identifier (indicated with the stereotype «id»), a *name* and one or more *addresses*. The *University* has also a list of *professors* and *students*. Both *Professor* and *Student* inherit from *Person*, which is abstract. *Student* has the attribute *id* as identifier, while *Professor* and *Person* have *name* and *surname*. *Student* has one or two *tutors* with type *Professor*, and *Professor*s have a *department*. Finally, while objects that have a *University* type do not need to be contained in any other object (stereotype «without container»), objects of type *Person*, *Professor* and *Student* must have a container (stereotype «with container»).

Using the meta-model of the University in the process of creating a chatbot, we obtain an agent able to interpret sentences and generate University models. Figure 3 shows an example of a user sentence and how the agent interprets it to generate the model. The agent, after processing the sentence ("There is a university with code UAM and name Universidad Autónoma de Madrid"), infers that there is an object with type *University*, that the attribute *code* has the value "UAM" and

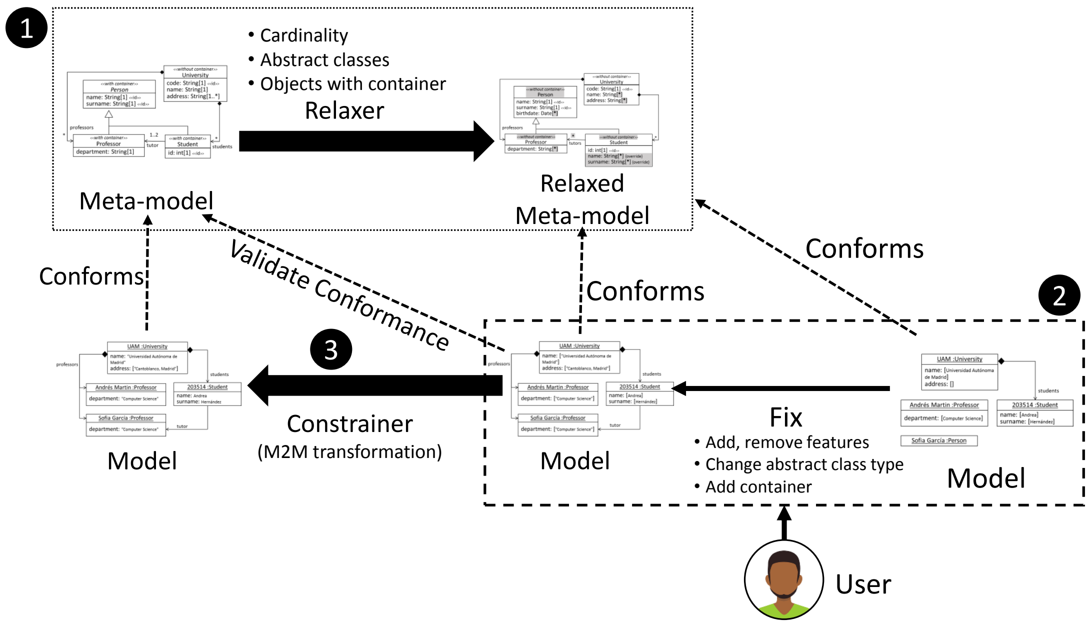


Fig. 5. Steps in flexible modelling. (1) Meta-model relaxation (2) User interaction to create a valid model. (3) Model constrainer

the attribute *name* is "Universidad Autónoma de Madrid"

When the agent processes the subsequent phrase “Sofía García was born on May 19, 1989” (Figure 4), it infers that there is an object of type *Person* with *name* “Sofía” *surname* “García” and *date of birth* “05/19/1989”. However, class *Person* is abstract, and so there is no way to save this partial information that the user gave to the agent.

III. Flexible Modelling with Chatbots

To allow the agent to save partial or incorrect information in the model, we propose to relax the meta-model with which the user will work. This relaxation takes place in the modelling back-end of Figure 1.

Figure 5 displays the steps we follow to make modelling with chatbots more flexible. The first step is to relax the meta-model. To do this, the tool changes the domain meta-model and the NL configuration as follows:

- **Cardinality:** It sets the cardinality of the features that are not identifiers to [0..*]. The identifiers can not change the cardinality because they can not be ambiguous, as users need them to refer to the objects.
- **Abstract classes:** The abstract classes become concrete.
- **With container class:** All classes are allowed to be outside of a container.

Then, users can build models according to the relaxed meta-model (step 2), so that they can instantiate abstract classes, or assign more values than permitted by the cardinality in a feature. At any moment, the user can validate the model to check its conformance to the original meta-model. The tool notifies all errors found in the model to the users. This way, users can fix the inconsistencies. The ways to resolve the inconsistencies are:

- **Cardinality:** If a feature has less values than the lower cardinality, it is necessary to add at least as many values as indicated by the lower cardinality. If the feature has more values than the upper cardinality, it is necessary to remove values until the size is equal or less than the upper cardinality.
- **Abstract classes:** There are several ways to retype an object with an abstract type into a concrete type:
  - The user specifies the type directly (e.g., “The Person Sofía García is a Student”).
  - The user sets a feature that only belongs to one of the subclasses of the abstract class (e.g. “Sofía García belongs to the Computer Science Department”).
  - The user adds the object to a reference whose type is a subclass of the abstract class (e.g. “Sofía García is a UAM professor”).
- **With container class:** The objects must be added in a container reference.

The last step is a model-to-model transformation. This transformation is necessary due to the limitation of the Eclipse Modeling Framework (EMF) [10], the technology we use to model. EMF treats features with cardinality greater than one and features with cardinality one in a different way when serializing models. This way, to permit opening the model created with the meta-model provided by the user, it is necessary to perform the transformation.

Figure 6 shows the relaxed meta-model from Figure 2 with the changes made shaded. The *Person* abstract class has been transformed into a concrete class, the classes configured with «with container» are configured with «without container» and all properties that are not class identifiers are set cardinality [0..*]. The *Student* features *name* and *surname* must be overri-

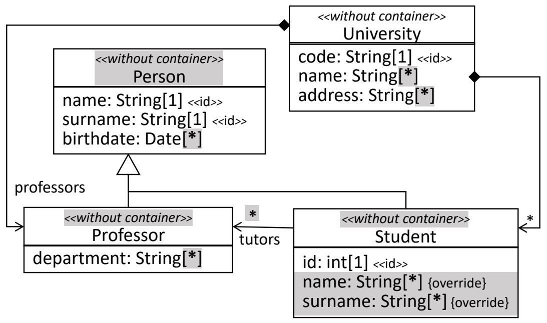
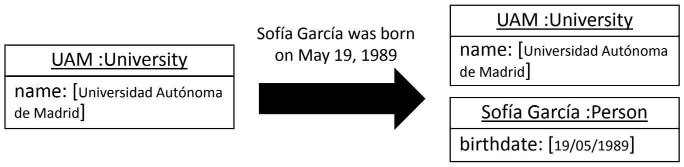


Fig. 6. Meta-model relaxation example



UAM :University
name: [Universidad Autónoma de Madrid]
Sofía García was born on May 19, 1989
UAM :University
name: [Universidad Autónoma de Madrid]
Sofía García :Person
birthdate: [19/05/1989]

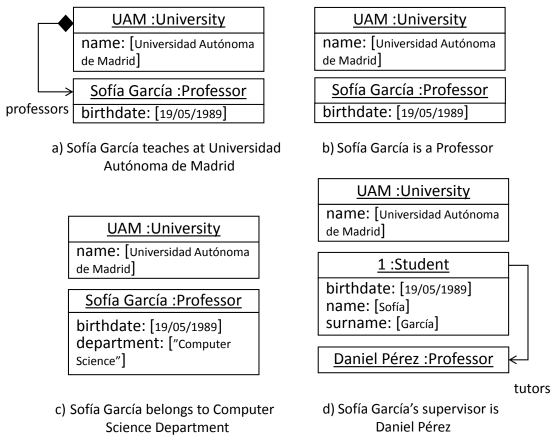
Fig. 7. Example of model creation in NL

den to increase the cardinality, since in *Person* and in *Professor* their cardinality must be [1..1], since they are identifiers.

IV. Example

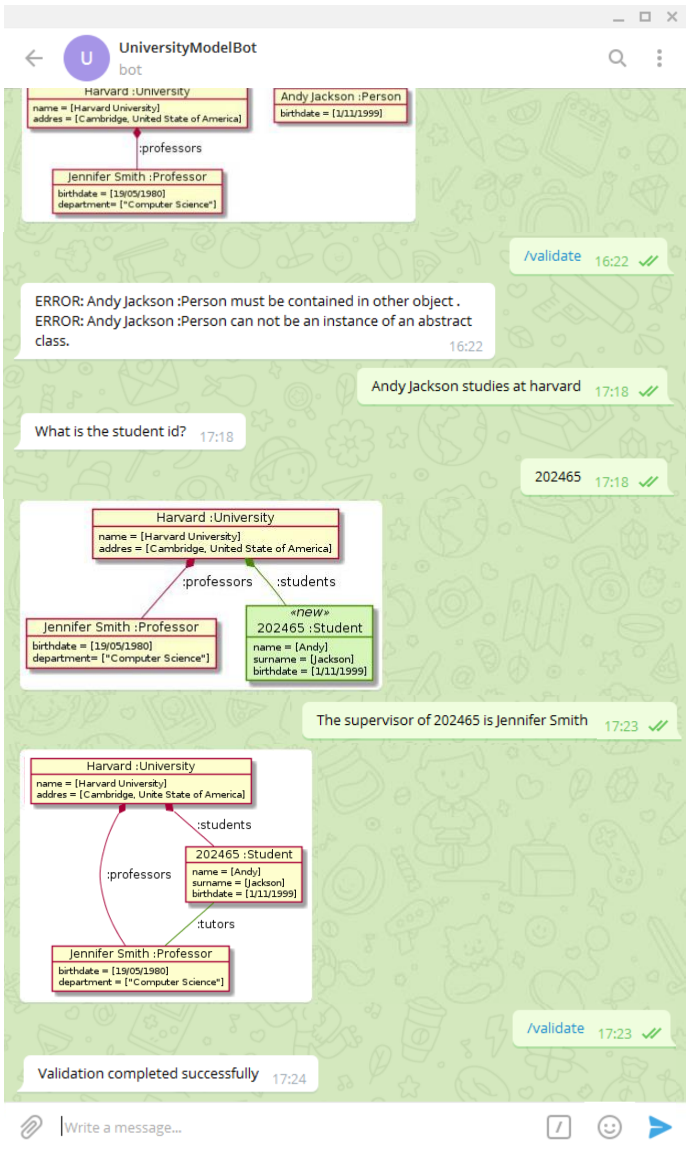
Figure 7 shows an example of a message in NL and how it is interpreted to generate the model according to the meta-model of Figure 6. From the message "Sofía García was born on May 19, 1989" the agent can infer that there is a *Person* with *name* Sofía, her *surname* is García and her *date of birth* is May 19, 1989, but it does not have information to classify her as a *Professor* or as a *Student*. With our flexible modelling approach the agent creates a *Person* object to save all the information provided by the user, and waits for the rest.

Figure 8 displays several ways to make the object type concrete. The most direct one is that the user says the type explicitly (Figure 4.b) with the phrase "Sofía García is a Professor". This results in an object retyping, which preserves existing attributes and links. However, there are other ways to concretize the type. For example when the object is assigned to the reference *professors* with type *Professor* (Figure 4.a), when the user sets feature *department*, which belongs to *Professor* (Figure 4.c) or *tutors*, which belongs to *Student* (Figure 4.d). Moreover, the phrase "Sofía García's supervisor is Daniel Pérez" creates Daniel Pérez as *Professor* because only *professors* can be supervisors of students.



UAM :University
name: [Universidad Autónoma de Madrid]
professors
Sofía García :Professor
birthdate: [19/05/1989]
a) Sofía García teaches at Universidad Autónoma de Madrid
UAM :University
name: [Universidad Autónoma de Madrid]
Sofía García :Professor
birthdate: [19/05/1989]
b) Sofía García is a Professor
UAM :University
name: [Universidad Autónoma de Madrid]
Sofía García :Professor
birthdate: [19/05/1989]
department: ["Computer Science"]
c) Sofía García belongs to Computer Science Department
UAM :University
name: [Universidad Autónoma de Madrid]
1 :Student
birthdate: [19/05/1989]
name: [Sofía]
surname: [García]
Daniel Pérez :Professor
tutors
d) Sofía García's supervisor is Daniel Pérez

Fig. 8. Four examples to type a Person object



UniversityModelBot
bot

/validate 16:22

ERROR: Andy Jackson :Person must be contained in other object .
ERROR: Andy Jackson :Person can not be an instance of an abstract class. 16:22

Andy Jackson studies at harvard 17:18

What is the student id? 17:18

202465 17:18

The supervisor of 202465 is Jennifer Smith 17:23

/validate 17:23

Validation completed successfully 17:24

Write a message...

Fig. 9. Example of model creation in NL

V. Tool support

Figure 9 displays the interaction of a user with a collaborative modelling agent for the University meta-model in Telegram. Telegram is a social network based on instant messaging. Users can communicate in chats that can be private (only two users exchanging messages) or groups (more than two users in the chat). Chatbots work almost the same as the rest of the users in the chats. On top, the figure shows a model with one *University* with code "Harvard", name "Harvard University" and address "Cambridge, Unite State of America". There is a Professor with name and surname "Jennifer Smith", birth date "19/05/1980", and department "Computer Science". Finally, there is a Person with name and surname "Andy Jackson", with birth date "1/11/1999". When the user validates the model ("/validate"), the agent sends back a list with all errors. The first error says that a *Person* object must be contained in a reference. The second says that the object "Andy Jackson :Person" cannot be *Person* because *Person* is an abstract class. Then, the user modifies the model using NL. The sentence "Andy Jackson studies in Harvard" indicates that Andy Jackson is a *Student* but students must have an id, so the bot asks the *id* to the user. From now on, Andy Jackson is identified by this *id*. The next phrase, "The supervisor of 202465 is Jennifer Smith", links Andy Jackson with Jennifer Smith through the reference *tutors*. Finally, the last validation shows there is no error in the model.

A video showing the interactions of two users with another modelling agent can be found at https://saraperezsoler.github.io/ModellingBot/.

VI. Related Work

There are many efforts in the field of requirements engineering on creating domain models (class diagrams) from textual requirements [11], [12]. While we also have the need to interpret NL, our approach is based on conversation, while the model we create is domain-specific.

Domain-specific modelling using NL or voice is a novel approach that is recently receiving a lot of attention. For example, in [13], the authors propose an approach called ModelByVoice, which 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. VoiceToModel [14] 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. Instead, our focus is to synthesize conversational syntaxes for DSLs that become as natural as possible, by using NL instead of commands.

We have seen that modelling using conversation benefits from a more flexible approach to modelling, which tolerates inconsistencies. Approaches to flexible modelling, based on the parsing of drawings, benefit from techniques for inferring types as well [15], just like we do. Some requirements for flexible modelling approaches were proposed in [16]. For example, the need for a configurable conformance relation, and modelling processes guiding in the transition from informal to formal models. However, these requirements targeted traditional modelling tools, while here we use modelling using conversation.

VII. Conclusions

In this paper, we have argued that modelling using chatbots would profit from adding some flexibility to modelling. In particular, we have proposed a meta-model relaxation process to give users more freedom when building models in NL, e.g., allowing the creation of abstract objects or the assignment of an arbitrary number of values to features. Then, a correction process converts the relaxed model into an instance of the original meta-model, reporting detected errors. These ideas have been implemented in our conversational modelling platform.

We are currently investigating further aspects which may bring flexibility to modelling through NL. For instance, our modelling chatbots are currently limited to error reporting, but we plan to extend them so that they can suggest to the user possible fixes in NL. We also foresee the possibility to customise the modelling process depending on the domain.

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] M. dos Santos Soares, J. Vrancken, and A. Verbraeck, "User requirements modeling and analysis of software-intensive systems," *Journal of Systems and Software*, vol. 84, no. 2, pp. 328 – 339, 2011.

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

[4] P. Markopoulos, J. Nichols, F. Paternò, and V. Pipek, "Editorial: End-user development for the internet of things," *ACM Trans. Comput.-Hum. Interact.*, vol. 24, no. 2, pp. 9:1–9:3, 2017.

[5] A. Xu, Z. Liu, Y. Guo, V. Sinha, and R. Akkiraju, "A new chatbot for customer service on social media," in *Proc. CHI*. ACM, 2017, pp. 3506–3510.

[6] N. Piyush, T. Choudhury, and P. Kumar, "Conversational commerce a new era of e-business," in *2016 International Conference System Modeling & Advancement in Research Trends (SMART)*. IEEE, 2016, pp. 322–327.

[7] J. Singh, M. H. Joesph, and K. B. A. Jabbar, "Rule-based chabot for student enquiries," in *Journal of Physics: Conference Series*, vol. 1228, no. 1. IOP Publishing, 2019, p. 012060.

[8] F. Clarizia, F. Colace, M. Lombardi, F. Pascale, and D. Santaniello, "Chatbot: An education support system for student," in *International Symposium on Cyberspace Safety and Security*. Springer, 2018, pp. 291–302.

[9] S. Perez-Soler, M. Gonzalez-Jimenez, E. Guerra, and J. de Lara, "Towards conversational syntax for domain-specific languages using chatbots," *Journal of Object Technology (proceedings ECMFA)*, vol. 18, no. 2, 2019.

[10] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[11] F. Dalpiaz, A. Ferrari, X. Franch, and C. Palomares, "Natural language processing for requirements engineering: The best is yet to come," *IEEE Software*, vol. 35, no. 5, pp. 115–119, 2018.

[12] C. Arora, M. Sabetzadeh, L. C. Briand, and F. Zimmer, "Extracting domain models from natural-language requirements: Approach and industrial evaluation," in *Proc. MODELS*. ACM, 2016, pp. 250–260.

[13] J. Lopes, J. Cambeiro, and V. Amaral, "Modelbyvoice - towards a general purpose model editor for blind people," in *Proc. MODELS 2018 Workshops*, ser. CEUR Workshop Proceedings, vol. 2245. CEUR-WS.org, 2018, pp. 762–769.

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

[15] A. Zolotas, N. Matragkas, S. Devlin, D. S. Kolovos, and R. F. Paige, "Type inference in flexible model-driven engineering using classification algorithms," *Software and System Modeling*, vol. 18, no. 1, pp. 345–366, 2019.

[16] E. Guerra and J. de Lara, "On the quest for flexible modelling," in *Proc. MODELS*. ACM, 2018, pp. 23–33.