Chatbots for Modelling, Modelling of Chatbots

UAM Universidad Autónoma
de Madrid

Sara Pérez Soler

Supervisors: Esther Guerra, Ph.D
Juan de Lara, Ph.D

Department of Computer Science
Autonomous University of Madrid

This dissertation is submitted for the degree of
Doctorado en Ingeniería Informática y Telecomunicaciones

January 2023

Abstract

Model-Driven Engineering (MDE) uses models as the main element of the development to raise the level of abstraction at which developers work. Therefore, model design is crucial for obtaining quality software. Hence, the collaboration between domain experts and modellers to guarantee a good model design is desirable. However, usually domain experts have a low technical background and do not have knowledge of modelling tools or modelling notations.

Social networks are intensively used nowadays for both leisure and work. They have become a natural communication mechanism which helps users in coordinating and collaborating in their daily life activities. Moreover, along with social networks, the use of chatbots is increasing to offer services to users by means of Natural Language (NL) conversation (e.g., booking trips, sending reminders or customer support).

The first part of this thesis aims at taking advantage of social networks and chatbots to enable the collaboration in modelling tasks by groups of users. It proposes three approaches to create conceptual models, and create and query domain-specific models, using chatbots. The first approach is prototyped in a modelling chatbot called Socio, which is able to interpret the users' inputs in NL to incrementally build a (meta-)model. It has been evaluated with students, in an educational setting, obtaining encouraging results. The second and third approaches are prototyped in two different web tools, which automate the generation of chatbots to create and query domain-specific models (respectively) using their meta-models. A case study illustrates their usefulness building and deploying streaming data applications using a conversational interface.

The second part of this thesis targets chatbot construction. Specifically, many frameworks and platforms have recently emerged for chatbot development. Identifying the most appropriate one for building a particular chatbot requires a high investment of time. Moreover, some of them are closed - resulting in customer lock-in - or require deep technical knowledge. To tackle these issues, the second part of this thesis proposes a model-driven approach to chatbot development. It comprises a neutral meta-model and a domain-specific language for chatbot description; code generators, parsers and validators for several chatbot platforms; and a platform recommender. The approach supports forward and reverse engineering, and model-based analysis. We demonstrate its feasibility presenting a prototype tool called Conga and an evaluation over chatbots developed by third parties.

Keywords: Chatbots, Model-Driven Engineering, Collaborative Modelling, Domain-Specific Language, Migration, Social Network, Natural Language Processing

Resumen

La Ingeniería Dirigida por Modelos (MDE de sus siglas en inglés) usa los modelos como elemento principal para elevar el nivel de abstracción al que trabajan los ingenieros. Por ello, el diseño de los modelos es crucial para obtener software de calidad. De ahí que sea deseable la colaboración entre expertos de dominio y de modelado para garantizar un buen diseño de los modelos. Sin embargo, normalmente los expertos de dominio tienen poca formación técnica y carecen de conocimiento en las herramientas de modelado o de las notaciones de modelado.

Las redes sociales se usan mucho hoy en día tanto para el ocio como para el trabajo. Integran nativamente mecanismos de comunicación que permiten a los usuarios coordinarse y colaborar en sus actividades cotidianas. Además, junto con las redes sociales, el uso de chatbots se ha visto incrementado para ofrecer servicios a los usuarios en lenguaje natural (por ejemplo, organizar viajes, enviar recordatorios o soporte al cliente).

La primera parte de esta tesis propone beneficiarse de las redes sociales y los chatbots para facilitar la colaboración en tareas de modelado por un grupo de usuarios. Propone tres enfoques para crear modelos conceptuales, y para crear y consultar modelos de dominio específico, usando chatbots. El primer enfoque ha sido prototipado en un chatbot de modelado llamado Socio, capaz de interpretar los mensajes del usuario en lenguaje natural para incrementalmente construir un (meta-)modelo. Se ha evaluado en el ámbito educativo con estudiantes, con prometedores resultados. El segundo y tercer enfoque se han prototipados en dos aplicaciones web diferentes, que automatizan la generación de chatbots para la creación y consulta de modelos (respectivamente). Un caso de estudio demuestra su utilidad construyendo y desplegando aplicaciones de transmisión de datos usando una interfaz conversacional.

La segunda parte de la tesis se centra en la construcción de chatbots. En particular, recientemente han surgido muchos entornos y plataformas para el desarrollo de chatbots. Identificar la herramienta más apropiada para construir un chatbot en concreto requiere una gran inversión de tiempo. Mas aún, algunas de las herramientas son privadas – lo que se traduce en una dependencia del proveedor – o requieren de un alto conocimiento técnico. Para abordar estos problemas, la segunda parte de la tesis propone usar la ingeniería dirigida por modelos para desarrollar chatbots. El enfoque se compone de un meta-modelo y un lenguaje de dominio especifico neutro para describir los chatbots; generadores de código, parsers y validadores para diferentes herramientas de chatbots; y un recomendador de herramientas. El enfoque soporta ingeniería directa e inversa, y análisis basado en modelos. Se ha demostrado su viabilidad con un prototipo llamado Conga y una evaluación sobre chatbots creados por terceras personas.

Palabras clave: Chatbots, Ingeniería Dirigida por Modelos, Modelado Colaborativo, Lenguajes de Dominio Especifico, Migración, Redes Sociales, Procesamiento de Lenguaje Natural

Table of contents

List of figures

List of tables

Abbreviations

AIML Artificial Intelligence Markup Language. 14

BPMN Business Process Model and Notation. 17

DSL Domain-Specific Language. iv, viii, 1, 3, 6, 7, 9, 17, 20, 30, 41, 56, 60, 62–71, 80, 91, 93, 96, 98

EMF Eclipse Modelling Framework. 21, 25, 31, 35, 42, 49, 51, 53, 61, 80

GPL General-Purpose Language. 17, 20

MDE Model-Driven Engineering. iii–v, 1, 3, 7, 9, 17, 21–23, 40, 41, 60–62, 65, 95, 96, 98

MOF Meta Object Facility. 20, 21

NL Natural Language. iii, vii, viii, xi, 2, 6, 9, 22, 24, 26, 27, 29, 35, 40–48, 50–56, 58, 60–64, 86, 91, 95–97

NLP Natural Language Processing. 10, 11, 13, 14, 22, 62, 91

NLU Natural Language Understanding. 92, 93

OCL Object Constraint Language. 19, 53, 56

OMG Object Management Group. xi, 20, 21

SUS System Usability Scale. 58, 59

UML Unified Modelling Language. 17, 19, 62

XMI XML Metadata Interchange. 21, 81

XML Extensible Markup Language. 14

Chapter 1

Introduction

The aim of this chapter is to provide a general overview of the thesis. Section 1.1 presents the motivation of this work and the main goals. Next, Section 1.2 summarizes the publications and technical contribution derived from this work. Section 1.3 contains a short description of the rest of the chapters.

1.1 Motivation

Model-Driven Engineering (MDE) is a software engineering paradigm that raises the abstraction level with regard to programming languages. Hence, models are used to automate many activities, like code generation, system simulation or testing. Moreover, models are used in all stages of the software development life cycle [16]

Models in MDE are built using modelling languages, frequently domain-specific ones. Domain-Specific Languages (DSLs) 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).

In MDE, a great part of the development time is spent on model creation and development since models are crucial in MDE to produce high-quality software. Indeed, to ensure a correct domain model, the collaboration between domain and modelling experts is common. Collaborative modelling is a challenge by itself, since it involves tools to synchronize the model, communication channels, and consensus methods. Moreover, collaborative modelling with domain experts represents a major challenge. Domain experts typically lack a strong technological background and are unfamiliar with developer or modeller tools and concepts. Determining strategies that make the collaborative process easier is crucial.

Our daily digital lives are increasingly based on social networks, which we use to communicate with friends and plan leisure activities. Additionally, improvements in NL processing have facilitated the emergence of chatbots, which operate on social networks and provide services with NL, imitating human conversation.

Given the widespread use of social networks and the expansion of chatbots, the goal of this thesis is to exploit them for collaborative modelling. For this purpose, we propose collaborative modelling through social networks assisted by *modelling bots* that interpret the messages of users in order to construct a model or meta-model or even to query them.

The main motivation for this approach is being able to benefit from the collaborative and ubiquitous nature of social networks – applications that people use on a daily basis – to perform assisted lightweight modelling. To this aim, the dissertation propose social networks as front-ends for the modelling activity, where dedicated chatbots interpret certain user messages to assist in modelling tasks. This way, the assisted modelling process seamlessly integrates with the normal use of social networks for discussion.

This approach enhances flexibility in modelling because it can be used in mobility and does not require installing new applications, but users can rely on apps they are already familiar with. When working on mobile devices, interacting via short messages can be easier and faster than using a diagramming tool, and can serve to quickly prototype models. Moreover, people with little or no background in computer science or modelling may be able to actively participate in modelling sessions. This may foster the collaboration of domain experts with teams of engineers. By recording the messages processed by the chatbot, the approach can trace information of the design decisions (who made what), so that every decision can be justified or rolled back.

On the other hand, the success of chatbots has led to the emergence of a plethora of technologies for their creation. Not only big software companies have made available chatbot creation tools, like Google's Dialogflow [31], IBM's Watson Assistant [102], Microsoft's bot framework [55] or Amazon's Lex [47], but many other proposals exist, like Rasa [82], FlowXO [35] and Pandorabots [66]. Among them, we find a variety of approaches. For example, Dialogflow and Watson offer low-code cloud development platforms that support the creation and deployment of bots, while Rasa is a framework that requires Python programming for bot development.

Overall, these chatbot creation tools are indisputably powerful (e.g., some provide NL processing, speech recognition, etc.). However, since there are so many options, choosing the most appropriate one to develop a chatbot with certain features is not easy. There may also be operational factors to consider in the decision, as for example, some options may imply vendor lock-in, and migrating chatbots between tools is not generally supported. Last but not least, some approaches have a steep learning curve and require expert knowledge.

To overcome these problems, this work proposes a MDE approach to chatbot development. This relies on a meta-model with core primitives for chatbot design, and a DSL to define bots independently of the implementation technology. Chatbots defined with the DSL can be analysed for defects, and a ranked list of appropriate bot creation tools is recommended based on the chatbot definition and other requirements. The DSL can be used for forward engineering, to produce the chatbot implementation from its specification; and for reverse engineering, to produce a model out of a chatbot implementation, which can then be analysed, re-factored and migrated to other platforms. Currently, the approach provides code generators and parsers from/to Dialogflow and Rasa, but the proposed architecture is extensible.

1.2 Contributions

This section contains the contributions of the dissertation. Section 1.2.1 presents the publications resulting from this work. Section 1.2.2 describes the technical contribution of the thesis.

1.2.1 Publications

This thesis is a compendium of the following publications. These publications have been organized into four groups: journals, conferences, workshops and journals in progress and under review.

1.2.1.1 Journals

4. Ranci Ren, Sara Pérez-Soler, John W. Castro, Oscar Dieste, Silvia T. Acuña. Using the SOCIO Chatbot for UML Modeling: A Second Family of Experiments on Usability in Academic Settings. *IEEE Access*, 10:130542–130562, 2022, doi: 10.1109/ACCESS.2022.3228772, ISSN: 2169-3536 (JCR impact factor in 2021: 3.476, **Q2 Computer Science, Information Systems**). Related with **Chapter 3**.

3. Sara Pérez-Soler, Sandra Juárez-Puerta, Esther Guerra, and Juan de Lara. Choosing a chatbot development tool. *IEEE Software*, 38(4):94–103, 2021,

doi: 10.1109/MS.2020.3030198, ISSN: 0740-7459 (JCR impact factor in 2021: 3.000, **Q2 Software Engineering**). Related with **Chapter 2**.

2. Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, and Juan de Lara. Towards conversational syntax for domain-specific languages using chatbots. *Journal of Object Technology*, 18(2):5:1–21, July 2019, doi: 10.5381/jot.2019.18.2.a5, ISSN: 1660-1769. Related with **Chapter 3**.

1. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software*, 35(6):48–54, November 2018, doi: 10.1109/MS.2018.290101511, ISSN: 0740-7459 (JCR impact factor in 2018: 2.945, **Q1 Software Engineering**). Related with **Chapter 3**.

1.2.1.2 Conferences

9. José María López-Morales, Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Asymob: a platform for measuring and clustering chatbots. In *IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)*, 2022, pages 16-20, 2022, doi: 10.1109/ICSE-Companion55297.2022.9793784 **(Conference Core A* in 2021)**. Related with **Chapter 4**.

8. Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Automating the measurement of heterogeneous chatbot designs. In *37th ACM Symposium on Applied Computing*, 2022, pages 1491-1498, 2022, doi: 10.1145/3477314.3507255 **(Conference Core B in 2021)**. Related with **Chapter 4**.

7. Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. Automating the synthesis of recommender systems for modelling languages. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 22-35, 2021, doi: 10.1145/3486608.3486905 **(Conference Core B in 2021)**. Related with **Chapter 3**.

6. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Creating and migrating chatbots with conga. In *IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)*, 2021, pages 37–40, 2021, doi: 10.1109/ICSE-Companion52605.2021.00030 **(Conference Core A* in 2021)**. Related with **Chapter 4**.

5. Sara Pérez-Soler, Gwendal Daniel, Jordi Cabot, Esther Guerra, and Juan de Lara. Towards automating the synthesis of chatbots for conversational model query. In *EMMSAD'2020 (CAISE)*, LNBIP 387, pages 257–265, 2020, doi: 10.1007/978-3-030-49418-6_17 **(Conference Core C in 2020)**. Related with **Chapter 3**.

4. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Model driven chatbot development. In *39th International Conference on Conceptual Modelling (ER'2020)*, Springer, pages 207–222, 2020, doi: 10.1007/978-3-030-62522-1_15 **(Conference Core A in 2020)**. Related with **Chapter 3**.

3. Ranci Ren, John Wilmar Castro, Adrian Santos, Silvia Teresita Acuña, Sara Pérez-Soler, and Juan de Lara. Collaborative modelling: chatbots or on-line tools? an experimental study. In *EASE'2019*, pages 260–269. ACM, 2020, doi: 10.1145/3383219.3383246 **(Conference Core A in 2020)**. Related with **Chapter 3**.

2. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Assisted modelling over social networks with SOCIO. In *Proceedings of MODELS 2017 Satellite Event co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)*, Austin, TX, USA, September, 17, 2017, pages 561–565, 2017, **(Conference Core B in 2017)**. Related with **Chapter 3**.

1. Sara Pérez-Soler, Esther Guerra, Juan de Lara, and Francisco Jurado. The rise of the (modelling) bots: towards assisted modelling via social networks. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, Urbana, IL, USA, October 30 - November 03, 2017, pages 723–728, 2017, doi: 10.1109/ASE.2017.8115683 **(Conference Core A in 2017, acceptance rate 22%)**. Related with **Chapter 3**.

1.2.1.3 Workshops

1. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Flexible modelling using conversational agents. In *5th Flexible MDE Workshop (satellite event of ACM/IEEE MODELS'19)*, pages 478-482. IEEE Computer Society, 2019, doi: 10.1109/MODELS-C.2019.00076. Related with **Chapter 3**.

1.2.1.4 Journals in progress and under review

2. Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Conga: A model-based solution for automated chatbot (re-)engineering. 2022. In progress. Related with **Chapter 4**.

1. Pablo C. Cañizares, José María López-Morales, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Measuring and clustering heterogeneous chatbot designs. *IEEE Transactions on Software Engineering*, pages 1–21, 2022, (JCR impact factor in 2021: 9.322, **Q1 Software Engineering**). Under review (2$^{\text{nd}}$ round). Related with **Chapter 4**.

1.2.2 Technical contributions

In addition to the mentioned publications, this thesis also makes the following technical contributions:

1. The development of a chatbot called SOCIO to create conceptual models (meta-models) in NL over social networks. SOCIO has an extensible architecture that allows adding support for its deployment in new social networks. The source code is available in https://saraperezsoler.github.io/SOCIO/

2. The development of a web tool to generate chatbots to create domain-specific models conformant to a given meta-model. The source code is available in https://saraperezsoler.github.io/SOCIO/

3. The development of a web tool to generate chatbots to query domain-specific models conformant to a given meta-model. The source code is available in https://github.com/SaraPerezSoler/QueryBot

4. The development of a web tool called Conga that integrates a DSL to define chatbots independently of the chatbot construction tool; code generators, parsers and validators for specific chatbot construction tools; and a recommender of chatbot construction tools, based on the chatbot definition and a questionnaire about the technical requirement of the chatbot. The source code is available in https://saraperezsoler.github.io/CONGA/

5. Collaboration on the development of Asymob, a framework to analyse and assess conversational agents or chatbots built in different tools (https://github.com/ASYM0B/tool).

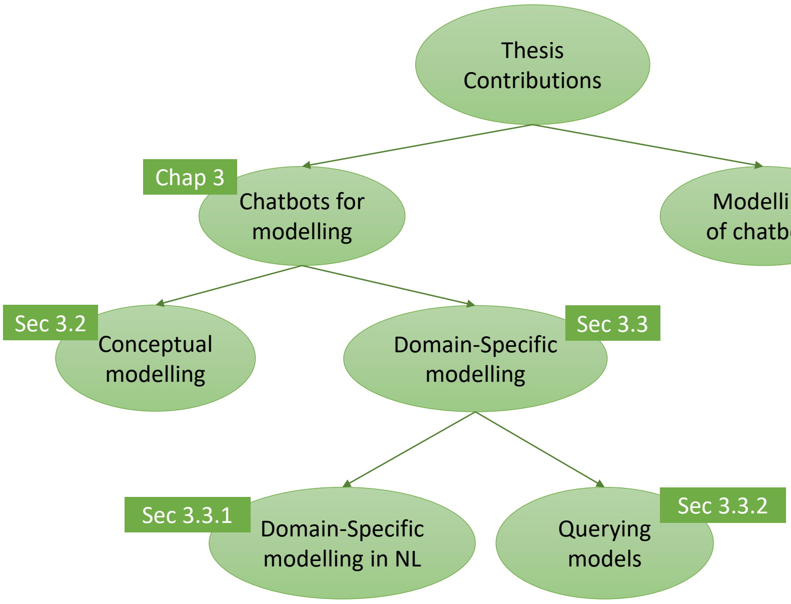


Figure 1.1 Scheme of the thesis organization.

1.3 Organization

The following is a brief summary of what each of the 4 chapters of this thesis describes. Figure 1.1 displays the organization of the contributions of this thesis.

- **Chapter 2** presents the context and background of the dissertation. It explains an overview of chatbots and their creation tools and an introduction of MDE concepts.
- **Chapter 3** details the proposed three approaches to use chatbots as a front-end for MDE activities. The first approach is for creating conceptual models (meta-models, Section 3.2) using chatbots. Next, Section 3.3 explains the contributed approaches to automate the generation of chatbots to create (Section 3.3.1) and to query (Section 3.3.2) domain-specific models from its meta-model.
- **Chapter 4** describes the proposed MDE solution to create chatbots. This solution includes a textual DSL (Section 4.3) to define chatbots; code generators, parsers and validators (Section 4.4) for specific chatbot creation tools; and a recommender (Section 4.5) that suggests the most suitable chatbot tool.
- **Chapter 5** concludes the thesis and proposes some future work.

Chapter 2

Background

This chapter describes the background elements behind this work. On one hand, Section 2.1 explains the chatbots, a general overview about their operation (Section 2.1.1) and the properties of fourteen chatbot creation tools (Section 2.1.2). This section is based on the paper [78].

On the other hand, Section 2.2 provides a general perspective on MDE and its concepts, such as model, meta-model, DSL, modelling language and code generator.

Finally Section 2.3 contains a summary and conclusions of the background.

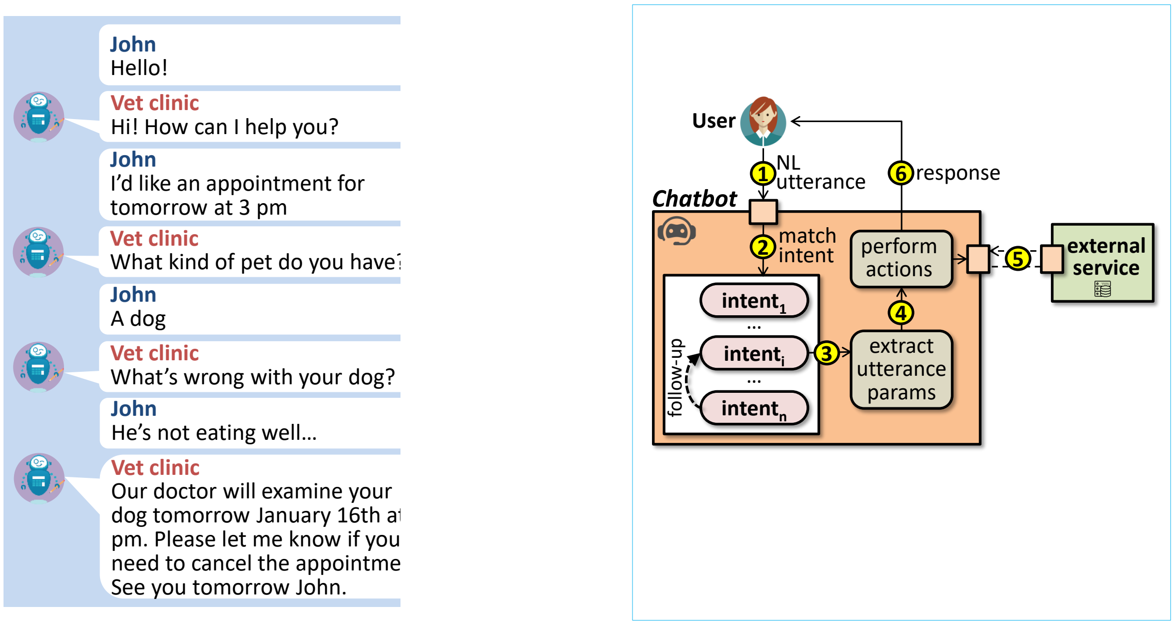
2.1 Chatbots

A chatbot is a program with conversational user interface that is usually accessible through the web, social networks or intelligent speakers like Google Home or Alexa. They can be classified into *open-domain*, if they can converse on any topic with users, or *task-specific*, if they assist in a concrete task (e.g., bookings flights or shopping). Our work targets the latter kind of bots.

In the following sections, we will explain how chatbots work (section 2.1.1) and review the features of the most prominent tools for their construction (section 2.1.2)

2.1.1 Chatbot Concepts

Chatbots are software programs that interact with users via Natural Language (NL) conversation. As an example, assume that a vet clinic has an information system with a database storing information about veterinarians and appointments, and decides to bring its services closer to customers by means of a chatbot to which customers can ask about opening hours and make appointments. This chatbot would allow the clinic



John
Hello!

Vet clinic
Hi! How can I help you?

John
I'd like an appointment for tomorrow at 3 pm

Vet clinic
What kind of pet do you have

John
A dog

Vet clinic
What's wrong with your dog?

John
He's not eating well...

Vet clinic
Our doctor will examine your dog tomorrow January 16th a pm. Please let me know if you need to cancel the appointme See you tomorrow John.

(a) Example of user interaction taken from [78].

(b) General working scheme of a chatbot taken from [76].

Figure 2.1 A chatbot conversation (a) and chatbot working scheme (b)

to offer 24/7 service, reduce costs (e.g., decreasing customer telephone calls) and widen the potential customers.

Figure 2.1a shows an example of a user interacting with the envisioned chatbot. Usually, the users start the interaction with the chatbot, and the chatbot asks questions to guide the conversation. In our example, the user starts the conversation greeting, and the chatbot responds to the greeting. Then the user requests an appointment with a date for his pet. As the chatbot does not know the kind of pet, it asks it, together with the pet problem. Finally, the chatbot confirms the appointment as scheduled.

Figure 2.1b shows the typical working scheme of task-specific chatbots. They are designed around a set of *intents* that represent possible users' intentions and permit accessing the offered services. These intents typically reflect the use cases of the chatbot. As an example, the chatbot for the clinic would define two intents: one to inform about opening hours, and another for making appointments. Given a user utterance (e.g., *"I'd like an appointment for tomorrow at 3 pm"*, label 1 in the figure), the chatbot tries to identify the corresponding intent (label 2). There are two different approaches to this. First, defining patterns or regular expressions upon which the utterance is matched. The second approach is based on Natural Language Processing (NLP) techniques using machine learning, which requires declaring training phrases for each intent and training models. If the chatbot does not find any matching intent, some approaches allow having a default fallback intent.

After matching an intent, the chatbot extracts the parameters of interest from the utterance (e.g., date and time of the appointment, label 3). Parameters may be typed by entities, which can be either predefined (e.g., date, number) or specific to a chatbot (e.g., pet type). If the utterance lacks some expected parameters (e.g., pet type), the chatbot can be configured to ask for them.

Besides intents, the conversation flow can be structured into expected sequences of intents (relation follow-up in the figure) to accomplish a task. For example, after the user requests an appointment in Figure 2.1a, the chatbot asks the pet problem, and only then the appointment is fixed. For this purpose, the chatbot needs to store the dialogue state to carry the information of previous input phrases through the stages of the conversation.

As a last step, the chatbot can perform different actions depending on the intent, such as calling an external service (e.g., the clinic database if the intent is setting an appointment, label 5) or replying to the user (label 6). The simplest response format is text, but some chatbot deployment platforms (e.g., Telegram, Twitter) also support images, URLs, videos or buttons.

2.1.2 Chatbot Creation Tools

The growing popularity of chatbots has caused the emergence of many tools for their construction. Table 2.1 compares the main available software options for chatbot construction. It includes proposals of both large companies (Dialogflow by Google [31], Watson by IBM [102], Lex by Amazon [47], Bot Framework [55] and LUIS [51] by Microsoft) and younger chatbot specialized companies (FlowXO [35], Landbot.io [45], Chatfuel [25], Rasa [82], SmartLoop [92], Xenioo [104], Botkit [15] which has been recently acquired by Microsoft, ChatterBot [26] and Pandorabots [66]). All are domain-independent but Chatfuel, which targets marketing applications.

The features analysed in the table stem from a thorough analysis of each tool. We distinguish between technical features (e.g., input processing) and managerial features (e.g., pricing model). The first row in the table indicates whether the software is a library, a framework, a platform or a service. While platforms and frameworks offer support for the whole bot creation life-cycle, services and libraries support only some steps, typically related to NLP. Frameworks provide sets of classes that need to be complemented with custom code for each created chatbot, and hence chatbots are built via programming. Most platforms are cloud-based, low-code development

Table 2.1 Comparison of chatbot libraries, frameworks, platforms and services taken from [78].

			Dialogflow (Google) [v2]	Watson (IBM) [v2]	Lex (Amazon) [07/06/2020]	Bot Framework + LUIS (Microsoft) [v4]	FlowXO [07/06/2020]	Landbot.io [07/06/2020]	Chatfuel [07/06/2020]	Rasa [10.1.2]	SmartLoop [07/06/2020]	Xenioo [07/06/2020]	Botkit (part of Bot Framework) [4.9.0]	LUIS [05/19/2020]	ChatterBot [1.0.5]	Pandorabots [07/06/2020]
Technical Factors		1. Kind (Library, Framework, Platform, Service)	**P**	**P**	**P**	**F**	**P**	**P**	**P**	**F**	**P**	**P**	**F**	**S**	**L**	**P**
	Input processing	2. Regular expressions/patterns	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
		3. NLP for phrase match	✓	✓	✓	✓			✓	✓	✓	✓		✓	✓	
		4. Text processing to obtain phrase parameters	✓	✓	✓	✓				✓	✓	✓		✓		✓
		5. Number of languages: very high (≥50), high (≥10), some (<10), 1 (represented with flag)	**h**	**h**	🇺🇸	**h**	**h**		**h**	**v**		**s**		**h**	**v**	
		6. Sentiment analysis	✓	✓	✓	✓								✓		
		7. Speech recognition	✓	✓	✓	✓						✓		✓		
	Dialogue	8. Storage of phrase parameters: volatile, persistent, both	**b**	**b**	**b**	**b**	**b**	**v**	**v**	**b**	**v**	**v**		**v**		**v**
		9. Support for intents	✓	✓	✓	✓	✓			✓	✓			✓		✓
		10. Support for entities: predefined, user-def, both	**b**	**b**	**b**	**b**	**p**	**p**		**b**	**b**	**b**		**b**		
		11. Dialogue structure: tree, followup intents, dsl	**f**	**f**	**f**	**f**	**t**	**t**	**t**	**t**	**f**	**t**			**f**	**d**
		12. Utterances to reengage users					✓		✓		✓	✓				
		13. Specification of chatbot answers	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓
	Deployment	14. Integration with social networks/websites: high (≥10), some (<10), 1 (represented with logo)	**h**	**s**	**s**	**h**	**s**	☎ www	f	**h**	**s**	**s**	**s**			**s**
		15. Interaction support for specific social networks	✓			✓						✓				✓
	Sys. integ.	16. Call to services from chatbot	✓	✓	✓	✓	✓	✓	✓	✓					✓	
		17. Chatbot usage via API	✓	✓	✓						✓	✓		✓		✓
	Development and testing	18. Pre-built components: chatbot templates, intents, small talks, services	**cts**	**c**	**i**	**cs**	**c**	**c**							**c**	**t**
		19. Version control: native, code-based	**n**	**n**	**n**	**c**				**c**			**c**		**c**	
		20. Chat console for testing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓
		21. Debug mechanisms	✓			✓				✓			✓		✓	
		22. Validation support	✓													
	Execution	23. Hosted deployment	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓		✓
		24. Support for analytics	✓	✓		✓	✓	✓			✓	✓				
		25. User message persistence	✓		✓	✓	✓	✓	✓			✓	✓			
	Sec	26. Cloud security	✓	✓	✓	✓								✓		
Managerial Factors	Organization	27. Pricing model: free, pay-as-you-go, quota, advanced feats	**fp**	**fp**	**fp**	**fpa**	**fq**	**fa**	**fa**	**fa**	**fa**	**fq**	**f**	**fq**	**f**	**fqa**
		28. Developer expertise: low, high	**l**	**l**	**l**	**h**	**l**	**l**	**l**	**h**	**l**	**l**	**h**	**h**	**h**	**l**
	Development	29. Code hosting: external, on-premises	**e**	**e**	**e**	**o**	**e**	**e**	**e**	**o**	**e**	**e**	**o**	**o**	**o**	**e**
		30. Group work				✓				✓			✓	✓	✓	
		31. i8n	✓													
		32. Open source								✓			✓		✓	
	Operational	33. New channels								✓			✓	✓	✓	
		34. No vendor lock-in								✓			✓	✓	✓	

environments to define chatbots graphically or via forms, and frequently support hosting the deployed chatbot logic for a channel. In addition, some platforms and frameworks (e.g., Dialogflow, Bot Framework, Rasa) also support the use of their NLP modules via services. Rows 2–26 in the table analyse decisive technical dimensions when selecting a chatbot development tool. These comprise aspects related to the processing of the user input text (rows 2–7), the dialogue support (rows 8–13), the chatbot deployment (rows 14–15), the integration with other systems (rows 16–17), testing and development support (rows 18–22), execution support (rows 23–25) and security aspects (row 26). In addition to technical factors, some managerial factors may influence the selection of a development tool. Rows 27–34 in Table 2.1 classify those factors among organizational (rows 27–28), related to development (rows 29–32), or operational (rows 33–34).

Input processing. Some approaches allow defining the expected input phrases using regular expressions or patterns (row 2), while others permit specifying intents via training phrases and then apply NLP (row 3). In addition, platforms like Landbot.io also support user inputs by means of buttons and widgets. Most approaches based on NLP can identify parameters in the input phrases, with the exception of Chatfuel and ChatterBot (row 4). Another important aspect in NLP is the language support (row 5). All approaches consider some of the most spoken languages (English, Spanish), and some platforms excel for their wide language support (e.g., Dialogflow includes 22). Interestingly, Rasa can use pre-trained language models (e.g., fastText word vectors are available for hundreds of languages [39]) but developers can train their own. Only a few approaches – the NLP service LUIS, Watson, Lex, Bot Framework, and the Enterprise non-free edition of Dialogflow – provide sentence sentiment analysis (row 6), which can be useful in specific domains such as marketing. Finally, in addition to text, several approaches natively support voice-based interaction (row 7). This interaction kind could be added by hand to approaches based on programming languages (e.g., Botkit) or which are open source.

Dialogue. This dimension looks at the capabilities to organize the conversation flow. All platforms and most frameworks automatically store the parameter values extracted from user phrases to allow their reuse in the future, while libraries require programming this facility (row 8). This storage can be volatile (active only during the current user interaction) or persistent. Intents and entities (rows 9 and 10) are common primitives of platforms like Dialogflow, Watson and Lex. Approaches supporting NLP define intents by sets of training phrases. These phrases may be examples of expected user utterances, or to improve the user experience, they may be obtained from existing conversation logs (e.g., when migrating a traditional customer support system into a chatbot). Regarding the dialogue structure (row 11), we find two main definition styles: explicitly by means of a conversation tree where nodes correspond to dialogue steps, or implicitly via dependent contexts and follow-up intents which are activated upon matching their parent intent (e.g., an intent for making appointments which declares a follow-up intent to inform the kind of pet). More differently, Pandorabots uses the Artificial Intelligence Markup Language (AIML [4]), an XML format from the '90s aimed to be a scripting standard for chatbots. Being based on templates, it is in stark contraposition to modern approaches based on NLP. Some platforms also permit defining utterances that the chatbot can use to reengage unresponsive users (row 12). Finally, all approaches but LUIS and Botkit permit specifying the chatbot answers (row 13).

Deployment. While some approaches allow deploying chatbots in many social networks, others target specific ones (row 14). For example, Chatfuel chatbots are specific for Facebook messenger, Landbot.io chatbots can be deployed just on WhatsApp Business and websites, while Dialogflow has 15 channel integrations including websites, services like Skype, intelligent speakers and social networks like Slack, Viber, Twitter and Telegram. Libraries and services lack deployment options, since this is out of their scope. In addition, Dialogflow, Bot Framework, Xenioo and Pandorabots permit including custom interaction mechanisms for the selected channel, like buttons in Telegram (row 15).

System integration. Several approaches enable calling services from the chatbots (row 16). In some cases, like Dialogflow, this is done by associating the URL of the service to an intent, so that matching the intent triggers a POST message to the service. In other cases, it is possible to define programs with custom code. For this purpose, Dialogflow supports Cloud Functions for Firebase, and Lex supports AWS lambdas. Conversely, some approaches offer an API that permits integrating parts of the chatbots with existing applications (row 17). For example, Dialogflow chatbots can be used programmatically to check the most probable matching intent given a user phrase.

Development and testing. Some tools offer pre-built components that can be added into new chatbots (row 18). These include generic chatbot templates (e.g., for a coffee shop or a hotel booking system), predefined intents, predefined small talks (answers to simple phrases and questions), or services (e.g., to build a QA chatbot from a knowledge base). Regarding version control (row 19), all frameworks and libraries rely on code and can be used with any generic version control system, while only some platforms (Dialogflow, Watson and Lex) give native support for versioning though this is simpler than state-of-the-art versioning systems like GitHub. As for testing, most approaches provide a web chat console to test the chatbots manually (row 20). For debugging (row 21), frameworks and libraries can rely on the support of the programming language, while only one platform (Dialogflow) offers debugging facilities to inspect the matched intent and related information. In addition, Dialogflow incorporates checks of the chatbot quality, such as detecting intents with similar training phrases (row 22).

Execution. Once a chatbot is defined, all platforms and most frameworks support its execution on their clouds (row 23). This solution can be optimal for many companies, especially if they already use the cloud services of the platform vendor (e.g., Google, Azure or AWS); however, this may not be always suitable. In some cases, like Watson, there is a special pricing plan to deploy the chatbot on third-party clouds. Finally, some approaches permit obtaining analytics about the chatbot usage (row 24) or persisting the user phrases (row 25). Developers might find the latter feature useful to adjust the accuracy of the intent recognition and improve the user experience [40]. Approaches like Watson automate this task, while others like Dialogflow require uploading the conversation logs and retraining.

Security. Chatbots may need to incorporate security aspects, especially if they work with private user data. While in general, implementing any security capability is the developers' responsibility, some tools can provide a security layer atop the cloud where the chatbot is deployed (row 26). Hence, approaches without deployment services do not offer this possibility natively. Instead, Dialogflow, Watson, Lex and Azure (Microsoft cloud for the Bot Framework and LUIS) provide a layer with features like firewalls; authentication and authorization when used via API; and secure connections (e.g., SSL or HTTPS/TLS). In addition, social networks like Whatsapp or Telegram support message encryption and user authentication.

Organizational factors. A critical selection factor is the pricing model of the approach (row 27). Most offer a free version suitable for small businesses or for experimentation (e.g., Dialogflow provides five free assistants and Watson supports 10,000 API calls). In addition, they provide other pricing models, typically collecting small fees for every interaction with the chatbot (the pay-as-you-go option of Dialogflow), limiting the number of interactions or active chatbots (the different plans of FlowXO), or supplying advanced features (e.g., webhooks in Landbot.io are not free). The expertise of the development team on chatbot-related technology is also important (row 28). Development platforms allow creating simple chatbots with no need for coding and require less expertise than approaches based on programming, though these latter are less constrained.

Development related factors. Like any kind of software, chatbot construction should follow proper engineering processes. In this respect, using a platform may be problematic if the chatbot development has to be harmonized with the company development culture and processes. For example, platforms host the chatbot specifications on their clouds, while the backend needs to reside in a different place; instead, chatbots developed with libraries, frameworks and services can run on-premises (row 29). Likewise, some code facilities such as versioning or debugging are standard for frameworks and libraries but may be unavailable for some platforms. The same applies to group work (row 30): platforms currently do not support synchronous collaborative development, so working on different parts of a chatbot cannot be parallelised among developers. Depending on the domain or the company strategy, the need to support several languages (i8n) can be necessary (row 31). Rather than developing a chatbot for each language, platforms like Dialogflow offer multi-language support by enabling the specification of different training phrases for each language over the same intent. Interestingly, among the reviewed approaches, only the community edition of Rasa, Botkit and ChatterBot are open source (row 32). No platform is open source, which may result in vendor lock-in, but it is possible to make public the chatbot specifications built with any platform.

Operational factors. Once a chatbot is in operation, the need to deploy it in novel channels or new versions of existing ones may arise (row 33). If the chatbot was developed using a platform, the available deployment options might be limited (e.g., Watson does not provide out-of-the-box deployment in Telegram). Libraries and (extensible) frameworks like Rasa, Botkit, LUIS and ChatterBot are more flexible, as they allow the manual implementation of the required deployment. Finally, platform-based approaches imply vendor lock-in as there are currently no migration tools using neutral exchange formats between platforms (row 34); however, an advantage of platforms is the ability to use the services of the provider (IBM, Google). Instead, libraries and frameworks require coding the chatbot logic in a programming language (like Python in case of Rasa), which brings more independence and safety with respect to possible policy changes of the platform owner company. This independence is stronger in open-source systems (row 32) since they could even be personalized to the developer needs.

Overall, this variety of chatbots creation tools makes it difficult to ascertain which tool is suitable to build a specific chatbot. Moreover, the conceptual model of the chatbot might be difficult to attain, as the chatbot definition frequently includes tool-specific accidental details. As a consequence, reasoning, understanding, validating and testing chatbots independently from the implementation technology becomes challenging.

2.2 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software engineering paradigm that uses models as the main elements in the software development process. Its goal is to raise the abstraction level, reducing the distance between the way developers think of solutions and the way they have to express them. Moreover, it increases the automation to obtain better products reducing the cost. In traditional development paradigms, models are only used in the design stage of a product, to guide in the implementation, and to document aspects of a software system. On the contrary, in MDE models become a development artefact and developers use them in all stages of the software development life cycle. They are created at the beginning and can be used to specify, design, test and generate code for final applications [16].

MDE models are abstract system representations, with the necessary information to describe the whole system. Abstraction, in this case, means compacting the information, erasing accidental details and focusing on essential aspects. This way the essence of the system is more comprehensible and better known. Also, as the models are parts of the software, their definitions should be formal. To formalize them, modelling languages are required [94]. There are two types of modelling languages: General-Purpose Language (GPL, e.g., UML) and Domain-Specific Language (DSL, e.g., BPMN).

Modelling languages are defined by an abstract syntax, a concrete syntax and semantics. The abstract syntax specifies the language's structure. It is defined by a meta-model, typically a class diagram, which specifies the domain elements and their features and relations [43]. Figure 2.2 shows an example of a meta-model of a Pre-doctoral System. The *Pre-doctoral System* contains a list of *people*. A *Person* has a *name* and a *birth date* and can be *Student* or *Professor*. *Professor* also contains a *department*. Besides *People*, the system contains a list of *Theses*. A *Thesis* has a *title*, a *status*, one or zero *defense date*, a *Student author*, one or two *Professor supervisors* and three to five *Professor committee* members.

Models should conform to their meta-model, and preserve the structure and rules defined in it. Figure 2.3 displays a model conforming to the meta-model of Figure 2.2. The *Pre-doctoral System* contains five *Professor*s (Daniel, Raquel, Sofía, Esther and

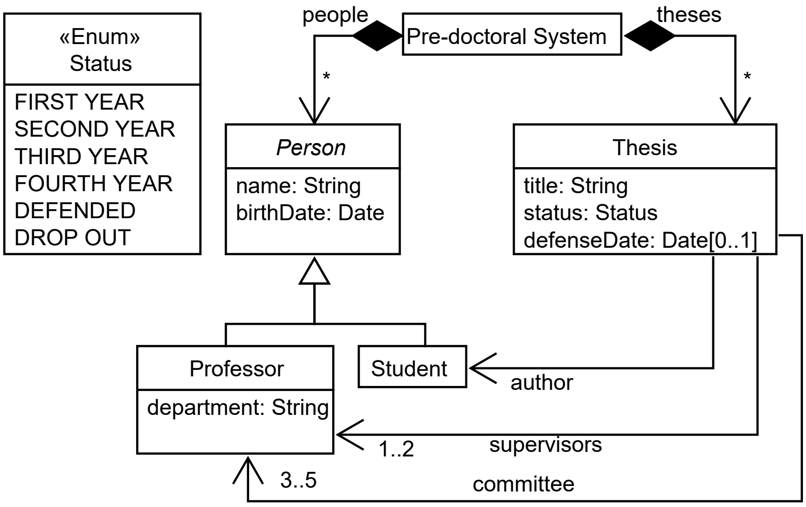


Figure 2.2 A meta-model example of a Pre-Doctoral System.

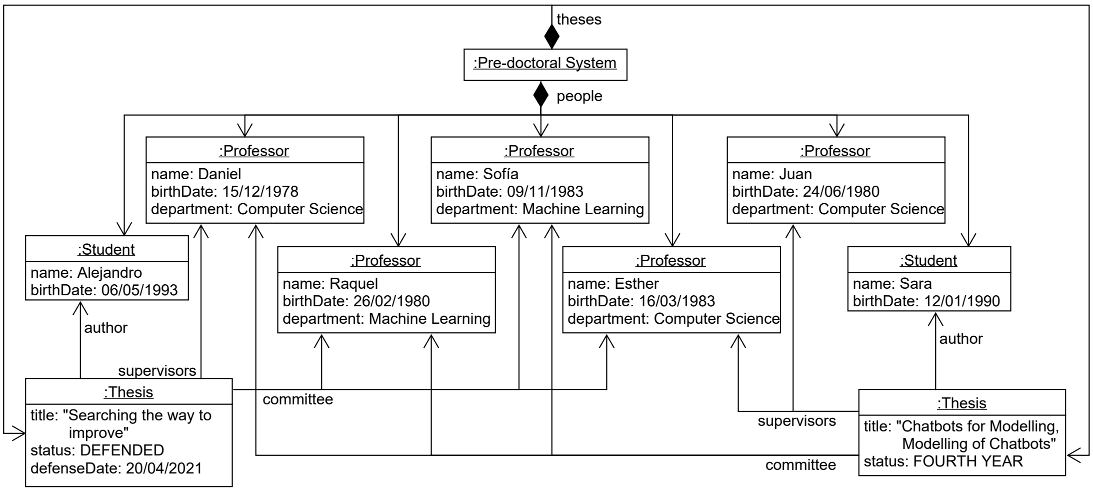


Figure 2.3 A model example instance of a Pre-Doctoral System (Figure 2.2).

Juan) with *birth dates*. Two of them of the "Machine Learning" *department*, and the rest of them of the "Computer Science" *department*. It also contains two *Students* (Alejandro and Sara) and two *Theses*. As an example, the first *Thesis title* is "Searching the way to improve", its *status* is *Defended*, the *defense date* is on 20/04/2021, the *author* is Alejandro, the *supervisor* is Daniel and the *committee* members are Raquel, Sofía and Esther.

Sometimes, class diagrams alone are not expressive enough to specify detailed aspects of a system. The Pre-doctoral System meta-model, for example, does not capture the following constraints: in a *Thesis*, a *committee* member can not be a *supervisor*, and if the Thesis *status* is *Defended*, it should contain a *defense date*. The Object Constraint Language (OCL) complements meta-models for this purpose. It is a formal language used to describe expressions in models. These expressions typically specify conditions (invariants) that must hold for the model to be considered valid [64]. Listing 1 shows the two previous constraints in OCL (between lines 1 and 3 the first one, and between lines 5 and 7 the second one). The OCL restrictions have a context (after the keyword *context*, both of them are *Thesis*), and a name defined after the keyword *inv* (supervisorsDifferentToCommittee and defenseDateInformed). The keyword *self* references the object of the context, and after a dot, the features of the object can be accessed (e.g., supervisors, committee, status or defenseDate). After an arrow, there are some OCL functions. The function *excludesAll* can be used with collections or lists, like supervisors and committee. It ensures that elements between the parenthesis are not in the collection. The function *oclIsUndefined* returns true if the value of the element is undefined. OCL expressions can also include relational (e.g., =, >, <, <>) and logical (e.g., and, or, not, implies) operators.

```
1 context Thesis
2 inv supervisorsDifferentToCommittee:
3   self.committee->excludesAll(self.supervisors)
4
5 context Thesis
6 inv defenseDateInformed:
7  self.status = Status.DEFENDED implies not self.defenseDate->oclIsUndefined()
```

Listing 1 OCL constraints for Pre-doctoral meta-model.

The concrete syntax defines how models are represented. It can be graphical (e.g., a UML diagram) or textual. Listing 2 shows the model of Figure 2.3 with a textual concrete syntax. The keyword *System* represents the Pre-doctoral System object. The *People* keyword references the people list, that contains *Professor*s (with their name, birth date and department) and *Student*s (with their name and birth date). The keyword *Theses* references the theses list. Each thesis is declared with a hyphen, followed by the *title*, *author*, *supervisors*, *committee*, *status* and *defense date* where applicable. Here we have a textual notation but an abstract syntax can be expressed with several notations (e.g., a graphical and a textual syntax).

```
System:
  People:
    Professor Daniel (15/12/1978, "Computer Science")
    Professor Raquel (26/02/1980, "Machine Learning")
    Professor Sofía (09/11/1983, "Machine Learning")
    Professor Esther (16/03/1983, "Computer Science")
    Professor Juan (24/06/1980, "Computer Science")

    Student Alejandro (06/05/1993)
    Student Sara (12/01/1990)

  Theses:
    – "Searching the way to improve"
      author: Alejandro
      supervisors: Daniel
      committee: Raquel, Sofía, Esther
      status: Defended
      defense date: 20/04/2021

    – "Chatbots for Modelling, Modelling of Chatbots"
      author: Sara
      supervisors: Esther, Juan
      committee: Daniel, Raquel, Sofía
      status: Fourth Year
```

Listing 2 An example of the textual concrete syntax for the Pre-doctoral system model of the Figure 2.3.

Finally, the meaning of models is defined by their semantics, typically, determined by transformations of the models into another domain. There are three main types of transformations: model-to-model (M2M), model-to-text (M2T) and text-to-model (T2M). Given a model a M2M transformation creates another model, which can be conformant to the same or different meta-model. This transformation specifies the mapping between elements in the source meta-model and the target meta-model. M2T transformations transform the model elements into text, typically, source code in a programming language, documentation or configuration files. When a M2T transformation generates source code, it is also called a code generator. T2M transformations are usually used to do reverse engineering, and transform source code into a model [97].

Figure 2.4 shows the 4-layer infrastructure proposed by the OMG [65], which comprises the description of languages to represent meta-models, down to the real system. Since a meta-model is also a model, a modelling language in needed to describe it, which is called a meta-meta-model (layer 3). The Meta Object Facility (MOF [59]) is the meta-meta-model proposed by the OMG [94]. There is no level above the MOF because it defines itself. The layer number two corresponds to the meta-models created with the MOF. This layer includes DSL meta-models and GPL meta-models. Layer one is where we can find the models that are instances of the meta-models. Finally, the Real System or the application data is an instance of the model (layer 0).

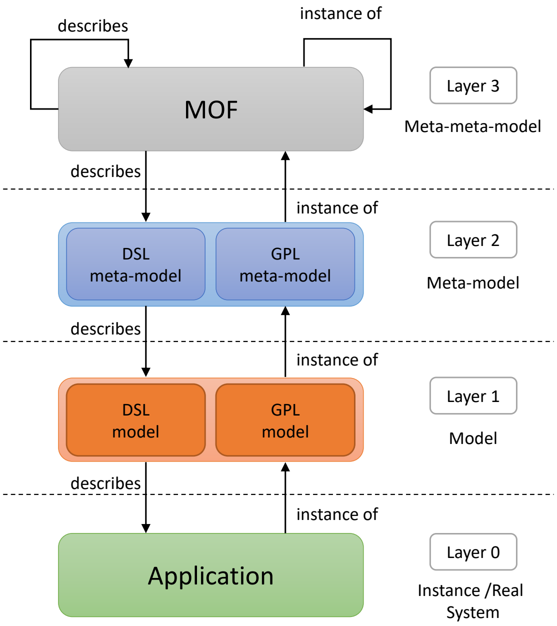


Figure 2.4 OMG 4-layer architecture.

As models are essential in MDE to create quality software, a great part of the development effort goes into model design and development. Indeed, to ensure a correct domain model, the collaboration between domain and modelling experts is common. If collaborative modelling is a challenge by itself because it requires communication channels, tools to synchronise the model and consensus mechanisms, collaborative modelling with domain experts represents a major challenge. Usually, domain experts have a low technological background, and they are not familiarized with developer or modeller tools and concepts. Therefore, is important to define approaches that facilitate the collaboration task.

There are different tools to give support to MDE, but the most widespread environment is Eclipse Modelling Framework (EMF) [95]. EMF uses Ecore as the language to describe meta-models, which is very similar to the MOF proposed by the OMG, and XMI format for the model and meta-model persistence.

Xtext [13] is the main framework to create textual syntaxes in EMF. Finally, Acceleo [3] and Xtend [13] are languages for code generation based on templates integrated in EMF.

2.3 Summary and Conclusion

Chatbots are programs that interact with users using NL conversations. They have a great potential for reducing costs (they replace customer telephone calls) and for improving the customer experience (they offer the service 24/7 in NL). Using NLP, they can match the user interaction with a predefined intent, extract information from the interaction and, depending on the intent and information, perform actions such as sending messages or requests to an information system.

Because of the increasing popularity of chatbots, many tools for their creation have emerged. These tools have different characteristics and properties, which we have analysed in Section 2.1.2

This variety of chatbot tools makes it difficult to choose the most appropriate one to develop a chatbot with certain features. Moreover, the chatbot definition frequently includes tool-specific accidental details. Designing chatbots independently of the particular tool to enable early reasoning and analysis, before the implementation, may be challenging. Last but not least, with a few exceptions, most chatbot tools are closed, proprietary software with no support for migration between tools, e.g., to benefit from the pricing plans of a competitor. This leads to vendor lock-in. This Thesis will address this problem in Chapter 4.

On the other hand, MDE is a software engineering paradigm that uses models in all stages of the software development life cycle. It increases automation, reducing the cost of development. Such models are the main element of software development, their design and creation are an important part of the process. Usually, domain model design involves domain and modelling experts. Collaborative modelling requires conversation channels, tools to synchronise the model and consensus mechanisms. Furthermore, domain experts usually are not familiarized with modelling tools or concepts, so the collaboration may represent a challenge. This Thesis will address this problem in Chapter 3.

Chapter 3

Chatbots for Modelling

This chapter will explore the use of chatbots and Social Networks in collaborative modelling tasks. On one hand, social networks are very popular in society, and people are familiar with them. Also, they have proven their effectiveness as a coordination and communication mechanism. On the other hand, chatbots have proliferated in recent years as front-end for all sorts of services. Therefore, this work aims at profiting from the benefits and potential of social networks and chatbots to promote the collaboration between modelling and domain (usually with low technical knowledge) experts.

Section 3.1 explains the motivation behind this work. Then, Section 3.2 describes the approach and tool support for a chatbot for conceptual modelling. After that, Section 3.3 details the approach for chatbots for domain-specific modelling (creation and query). Section 3.4 reports on evaluations of our approaches. Finally, Sections 3.5 and 3.6 compare with the related work and conclude the chapter respectively.

This chapter is based on publications [5, 83, 70–75]

3.1 Introduction and Motivation

Many software development activities are not individual but require collaboration among teams of stakeholders. Modelling is no exception to this rule since the initial stages of development generally involve heterogeneous partners with diverse backgrounds and likely distributed. Among them, the participation of domain experts is essential for building successful domain models. Usually, domain experts have a low technological background, and they are not familiarized with developer or modeller tools and concepts. However, one of the limiting factors in MDE today is poor tool support for collaboration [85].

Franzago et al. [36] have identified the main aspects that a collaborative modelling environment should contain:

- **Model management** as a distributed model environment for managing the life cycle of the models, with a repository to manage models persistence and a modelling tool where stakeholders can edit models. The authors suggest multi-device access to models, which facilitates collaborations in mobility.

- **Communication mechanisms** to allow involved stakeholders to be aware of the other stakeholders and exchange messages and information to coordinate themselves as a team. This mechanism must guarantee traceability between the designed decision debated and the modelling artifact.

- **Collaborative mechanisms** to allow involved stakeholders to participate in the modelling task in collaboration.

In addition, social networks are becoming an important part of our daily digital lives, where we use them to keep in touch with friends and organize leisure activities. Not only general-purpose networks like Twitter [100], Facebook [33], Whatsapp [103] or Telegram [98] have boosted, but specialized work-team nets like Slack [91], Microsoft Teams [57] or Yammer [106] have spread in enterprises. The reason of this success is their agility, simplicity of use, and the possibility to use them everywhere and in mobility, covering the need to stay connected while being familiar to people. The use of social media has also disrupted how software engineers work, changing the way developers communicate with colleagues e.g., supporting the formation of ecosystems around particular concepts and technologies; participating in online development communities; and disseminating technologies and crowdsource content [96].

Moreover, advances in NL processing have enabled the proliferation of chatbots which run on social networks and offer services to users upon NL requests, thereby mimicking human responses. Developers use bots, e.g., to automate deployment tasks, schedule tasks like sending reminders, integrate communication channels, or customer support [48]. They have also been proposed to access API documentation [99] and analyse software projects [12]. 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.

Given the widespread use of social networks and the expansion of chatbots, the goal of this chapter is to exploit them for collaborative modelling. Hence, the concept *modelling bot* is introduced, able to interpret natural language (NL), assist users in creating models, and which integrates with minimum disruption into the natural communication mechanism that micro-blogging based social networks – like Twitter or Telegram – offer. This approach enhances flexibility in modelling because it can be used in mobility and does not require installing new applications, but users can rely on apps they are already familiar with. When working on mobile devices, interacting via short messages can be easier and faster than using a diagramming tool, and can serve to quickly prototype models. Moreover, people with little or no background in computer science or modelling may be able to actively participate in modelling sessions. This may foster the collaboration of domain experts with teams of engineers. By recording the messages processed by the bot, the approach can trace information of the design decisions (who made what), so that every decision can be justified or rolled back.

This approach can be useful in several scenarios. First, to allow engineers quickly prototyping models when and where needed (e.g., in working meetings, but also when travelling home). Second, to assist teams of engineers collaborating with domain experts (who may lack a computer science background) to create domain models or meta-models. Third, in the educational domain, to enable groups of students the collaborative resolution of modelling exercises. In this scenario, bots could be configured for gamification activities or blended learning. Finally, being based on social networks, the modelling process can involve a large number of people, enabling, for example, crowdsourcing modelling decisions.

In order to give support to these scenarios, modelling chatbots must achieve the following requirements in the envisioned approach:

- Interaction through NL (to permit use by domain experts) and commands (more suitable for modelling experts). Anyhow, commands should have a flexible and natural syntax to minimize mistakes and user frustration.

- Traceability mechanisms able to justify design decisions and find their provenance.

- Integration of multiple social networks, so that users can use their preferred one.

- Support for both modelling (domain-specific modelling) and meta-modelling (conceptual modelling).

- Customizable collaboration protocols, e.g., supporting voting or user roles.

- Interoperability with accepted modelling frameworks, like the Eclipse Modelling Framework (EMF) [95].

The next sections detail the steps towards realizing this vision.

3.2 Chatbots for Conceptual Modelling

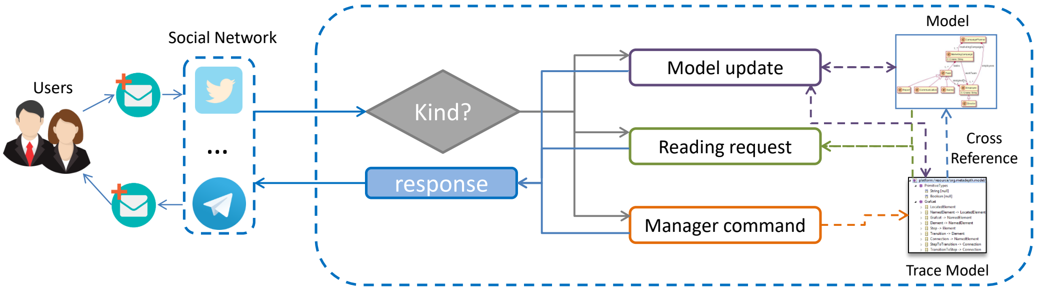
The goal of this approach is to provide meta-modelling assistance integrated within social networks. For that purpose, Section 3.2.1 presents a chatbot with support for modelling in NL. It also contains collaboration mechanisms (Section 3.2.2) such as voting. Section 3.2.3 realizes these ideas in a prototype tool called Socio (from assisted modelling through social networks,).

3.2.1 Conceptual Modelling in NL

Figure 3.1 sketches the proposed approach to modelling via social networks. Users can interact within the network of choice by sending messages directed either to the other partners or to the modelling bot. The former messages permit discussing and coordinating, and they are handled by the network normally (i.e., they are regular text messages).

Messages directed to the bot are received by the users of the social network, just like any other message, but in addition, they are processed by the bot. There are three kinds of such messages: management commands, reading requests and model update messages. The former allow performing project management tasks, like creating or deleting a model. The bot processes such messages and sends the result as a text to the social network. Reading requests allows obtaining information of the model, like recovering the history of changes performed in a model, validating or downloading a model, and obtaining statistics. Finally, model update messages can be either commands or descriptions. The former are imperative actions to directly manipulate a model, e.g., to add a class or feature, change its type, or delete an element. The latter are descriptive statements of the domain concepts, like "houses have windows".

Both commands and descriptions are expressed in NL. Figure 3.2 shows a scheme of the processing of NL messages for model update. When the bot receives an update message,it uses a NL parser to process it and produce a parse tree of the sentences in the message. Supporting commands in NL provides flexibility because there is no need to adhere to a strict syntax. The system has an extensible library of NL processing rules, able to handle different kinds of NL phrases. From the current model state and the information inferred from the message, the chatbot synthesizes a number of model update actions. As the approach is incremental, sometimes it is necessary to refer to



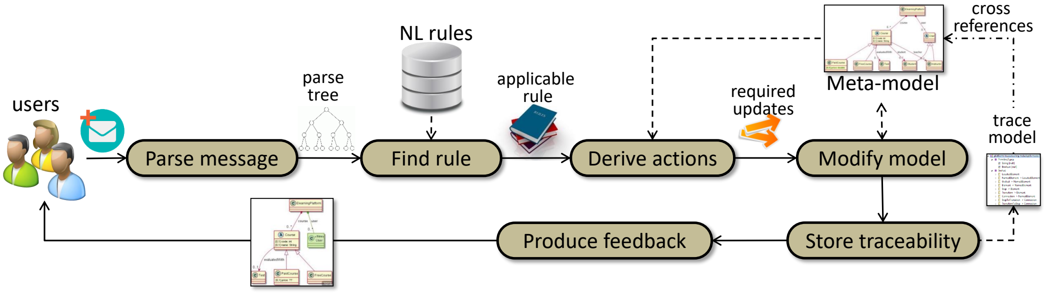


Figure 3.2 Processing a NL message for model update

existing model elements. In order to provide flexibility and avoid redundancies, the chatbot allows for synonyms which are sought using WordNet [58], a lexical database for the English language. The NL processing approach is described in Section 3.2.1.1, and the extraction of model update actions in Section 3.2.1.2.

The extracted actions are applied to the current version of the model to make it evolve. Moreover, the system maintains a traceability model to keep track of *why* the model was updated, and by *who*. Section 3.2.1.3 will explain model building and traceability. Finally, the bot emits a feedback message, which is received through the social network. The feedback in response to model update messages consists of an image of the updated model where the impacted elements are highlighted in a different colour. Section 3.2.1.4 will illustrate some steps in a model construction session, and the feedback obtained.

3.2.1.1 Natural Language Processing

The modelling chatbot uses the Stanford NL parser [53] to process model update messages, both commands and descriptions. The parser creates a parse tree with the grammatical relations of the words in the message. This tree identifies the noun phrases (NP) that may provide information about the model, and the syntactical role of each part of the phrase. For example, for message “houses have windows”, “houses” and “windows” are tagged as common plural nouns (NNS), while “have” is tagged as a verb in present tense which is non-3rd person singular (VBP). The complete list of tags is available in [67].

The parsed messages are interpreted according to a number of rules, which are based on the work of Arora and collaborators [6]. Each rule specifies the combination of word classes that activate the rule, usually based on the presence of certain verbs, as well as the model update actions that should be performed when the rule is matched. The following rules are considered:

- **Verb to be**: When the main verb of the phrase is “to be”, it can indicate either an inheritance relation between two classes (e.g., “Kitchen is a room”, “Service may be premium service or normal service”), the type of a feature (e.g., “Name is string”, “The bank of the customer is BLUX”) or the abstractness of a class (e.g., “course is abstract”). If the phrase contains an expression of the form “A of B” or a genitive “B’s A”, then the rule infers that A is an attribute or reference of B.

- **Verb to have**: When the main verb is “to have”, or synonyms of it like “characterized by” and “identified by”, this rule infers the subject of the sentence is a class, which has a feature. Deciding whether the feature is a reference or an attribute depends on the information there is in the meta-model about the feature. If there is not enough information, the feature is assigned an “open” type which may be refined by subsequent messages. As an example, the message “Bulky packages are characterized by their width, length and height” triggers the creation of features *width*, *length* and *height* with open type in class *BulkyPackage*.

- **Contain**: Verbs like “contain”, “be made of”, “include” and “be composed of” imply a composition relation between two classes. For example, the phrase “A delivery is made of packages” creates a composition relation between the classes *Delivery* and *Package*, and also creates the classes if they do not exist yet.

- **Transitive verb**: This rule handles all verbs with a subject and a direct object. It creates classes for the subject and direct object, and a reference whose name is the verb. For example, the phrase “The simulator shall send log messages” triggers the creation of classes *Simulator* and *LogMessage*, and the reference *send* from the former to the latter.

- **Add**: This rule handles imperative sentences (with implicit subject) whose verb is “add”, “create”, “make”, etc. These are interpreted as commands with a flexible syntax, resulting in the creation of classes, attributes or references. For example, “add house” will create the class *House*, while “create room in house” adds a feature *room* to class *House*.

- **Remove**: This rule is similar to the previous one but for deletion. It considers imperative verbs synonyms of “remove”, like “delete” and “erase”.

Model update messages can include several sentences and more than one verb, like in “Add house and remove windows”. Moreover, the processing of one message may trigger several NL rules, in which case, the chatbot applies the rule with higher priority. In particular, rules seeking for specific verbs have higher priority than the more general rule seeking transitive verbs.

3.2.1.2 Model Update Actions

As previously mentioned, each NL rule specifies the model update actions to be applied when the rule is selected. The possible actions are the following:

- **Add class**: This action is issued when the rule finds a common name that should be a class. The class is not created if one exists with the same or synonym name. Supporting synonyms provides flexibility and avoids redundancy. The chatbot applies accepted modelling styles for class names (i.e., singular, camel case).
- **Make class abstract/concrete**: Classes can be set to abstract or concrete using their name or a synonym. If the class does not exist, then an *add class* action is issued as well.
- **Set parent class**: This action sets an inheritance relation between two classes, creating the classes if they do not exist.
- **Remove parent**: This action removes an inheritance relation. If the class does not exist, the action will make no changes.
- **Add attribute**: This action is issued by the “verb to have" rule (e.g., “packages have weight”) and in case of genitive cases (e.g., “package’s name”). The attribute is added to the given class or to a synonymous one, creating a new class if it does not exist. If the class already owns a reference with same name, it is replaced by an attribute. The attribute’s upper cardinality is set to 1 if the attribute name is singular, or to * if it is plural. At this point, the attribute type is left open, so that it can be refined later.

- **Add reference**: This action is issued by the "transitive verb" and "contain" rules, and it works similarly to the previous action. If the owner class already defines an attribute with the same name, it is replaced by a reference.

- **Modify feature type**: The chatbot supports primitive data types like int, float, String, boolean and Date for attributes, while the type of references must be a class. The feature is created if it does not exist.

- **Remove class**: It removes a class and its features.

- **Remove feature**: It removes one or several features.

Any action can be undone and redone through commands.

3.2.1.3 Model Update, Recommendation and Traceability

The actions derived from the messages are applied to the current model version. In some cases, these actions may lack some information. For instance, the action that adds an attribute may miss the specific type of the attribute, in which case, its type is left "open" so that it can be refined later. Similarly, as removing a class would let the references pointing to the class dangling, the chatbot adds a provisional "ghost" class as target of these references. The modelling bot is extended with a recommender system specified with Droid [5] to recommend new elements to the model and also allow fixing incomplete elements.

Droid [5] is a tool to automate the generation of recommender systems for modelling languages. It has a DSL to configure every aspect of a recommender system. The generated system can be deployed as a service. This way, Socio integrates a recommender system created for class diagrams. When the user requests a recommendation, she must specify the class subject of the recommendation. The recommender system will recommend attributes with their types, and also superclasses for that class. When the user selects the item of interest, the modelling bot adds it automatically to the class.

Socio also maintains traceability information of each message sent to the bot, including the sender and the model update actions it triggers. It uses a model-based approach to record the traceability data, building a traceability model conformant to the meta-model in Figure 3.3 for each model being constructed. Class *User* keeps track of the participant users and the social network (*channel*) they employed to send the messages. The system stores all messages directed to the chatbot, and distinguishes the message used to create the model. *Actions* point to the model elements affected by the action using reference *element*, whose type is *EObject* as this is the base class in EMF,

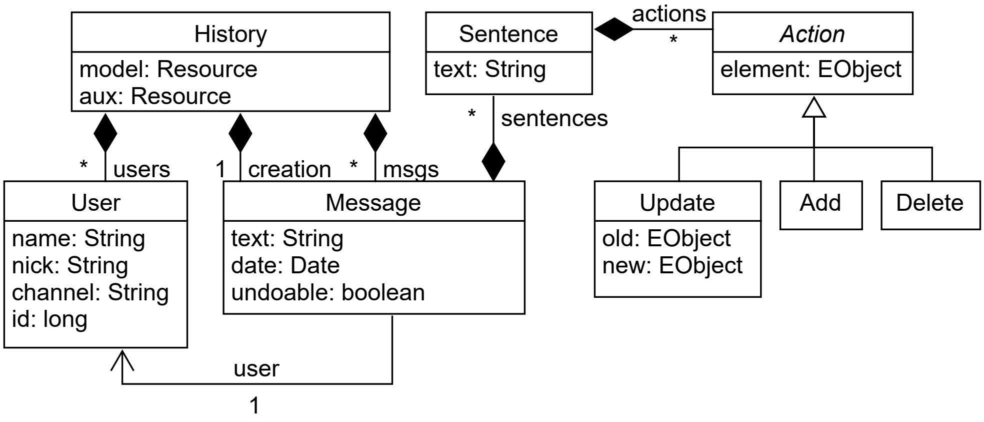


Figure 3.3 Traceability meta-model

the implementation technology used. To keep track of the removed elements that are no longer in the model, the system stores them in an auxiliary model. Finally, *Update* actions point to the *old* and *new* versions of the updated element in the auxiliary model, and to the current version of the *element* in the model.

An image of the updated model diagram is sent to the collaborator users. The modified model elements are marked in the diagram. This information is read from the traceability model. The trace can also be queried using a reading request, which sends a text with the traceability information to the users.

3.2.1.4 Example

Figure 3.4 illustrates a typical modelling session. The rectangles labelled 1–4 contain NL messages that a user sends, while the diagrams are the feedback provided by the modelling bot.

The first message is handled by the “transitive verb” rule. This creates classes for “good transport company” and “delivery”, and a reference for “handle”. The cardinality of the reference is many as “deliveries” is in plural. The created classes have singular, camel case names. The newly created elements are shown in green. For space constraints, the figure omits the explanation of changes which the bot also produces as notes in the diagram.

The second message is handled by the “verb to have” rule, which adds an integer attribute to class Delivery. The bot assigns an upper cardinality of 1, as there is no plural.

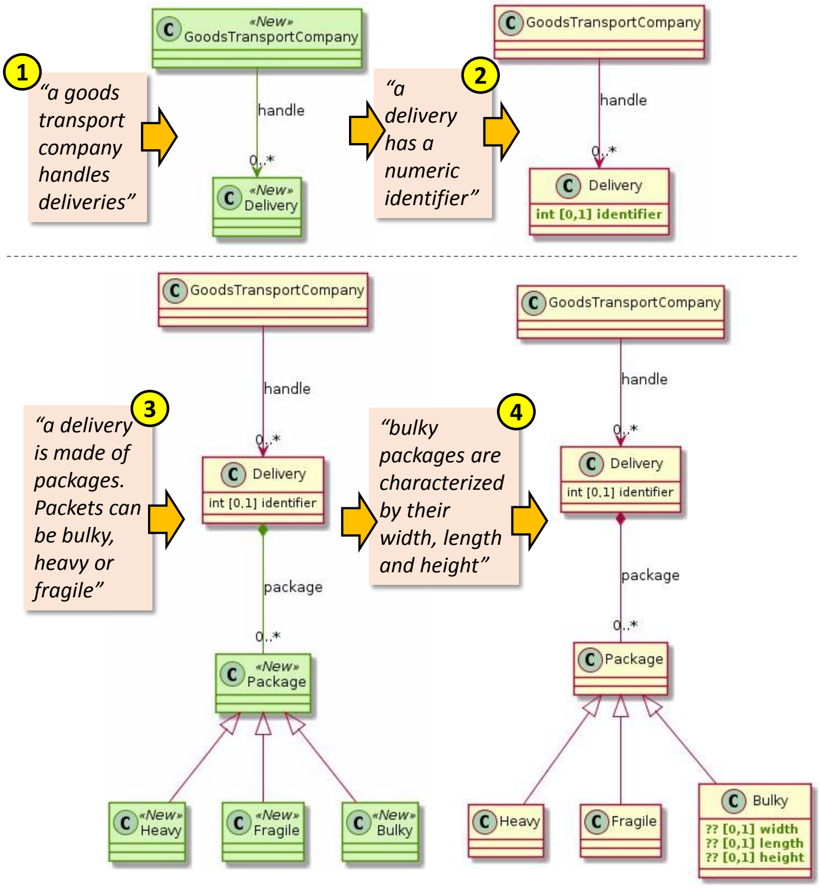


Figure 3.4 Some steps in a model construction session

The third message contains two phrases. The first one is handled by the "contain" rule, which creates class Package and reference package. The second one is handled by the "verb to be" rule, which creates the inheritance hierarchy. This phrase makes use of the word "packet", which is as a synonym of "package", and hence, no new class is added for it.

The fourth message, processed by the "verb to have" rule, creates three features in class Bulky. At this point, there is no information on whether they should be attributes or references, and hence, this is left open (shown with "??").

3.2.2 Collaboration Support: Soft Consensus for Group Decision-Making

The participants in a collaborative session may require exploring several solutions to a modelling problem, and eventually, they will have to opt for one of them. If collaboration is distributed or involves many participants, assistance to facilitate consensus is essential for agile coordination.

Let us assume that, after a modelling session for Marketing Campaigns, the modelling team obtains the model in Figure 3.5a. A *Campaign Planner* contains several *Employees* (with their names) and *Marketing Campaigns* (with their names). *Marketing Campaign* contains a list of *Task*s *assigned to* an *employee*, which can be *Report*, *Communication* or *Survey*. *Marketing Campaign*, also, has a *director Employee* and a *work team* of *employees*.

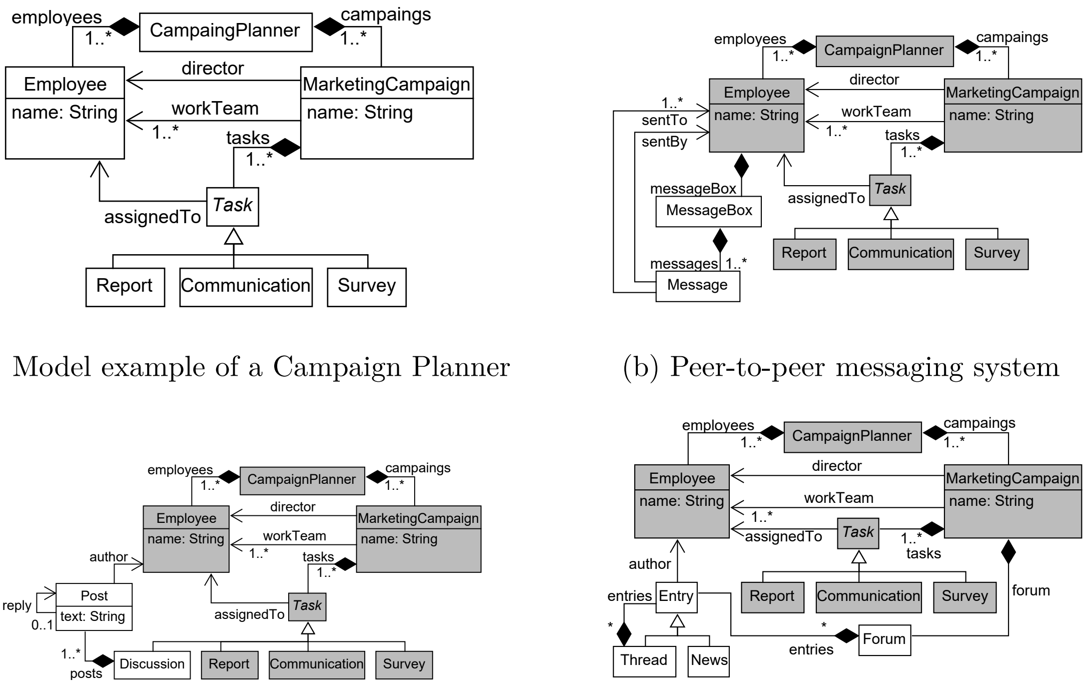
The participants in the example have expressed the need for a communication channel between the team members of a marketing campaign. The following options are being considered:

- A message box per employee, not necessarily working in the same campaign. This would be like an e-mail or peer-to-peer messaging system.

- A special type of work task for discussion, where any employee can post comments and reply to other comments.

- A forum associated to each marketing campaign, where work-team members can contribute news organized into threads, like in bulletin boards.

To study different possibilities, the approach supports creating branches to explore alternative modelling solutions. These collect the alternatives and discussions to model an aspect of a system as different branches of the current model. In this example, this



(a) Model example of a Campaign Planner

(b) Peer-to-peer messaging system

(c) Work task for discussion

(d) Forum associated to each marketing campaign

Figure 3.5 A model example of a Campaign Planner (a) and three branches: a peer-to-peer messaging system (b), a work task for discussion system (c) and a forum associated to each marketing campaign (d).

facility is used to create three branches, one for each considered solution: p2pMessages (Figure 3.5b), DiscussionTasks (Figure 3.5c), and BulletinBoard (Figure 3.5d).

After outlining the alternative solutions in branches, the participants need to agree on the most suitable one. For this, the modelling chatbot incorporates a soft consensus mechanism for multi-person decision-making [41] that assists in choosing the option that is acceptable to all participants. Participants can express their favourite solution in several ways, like ordering the alternatives from better to worse, or giving a score to each option (e.g., from 1 to 10). Then, an iterative consensus process identifies the preferred alternative based on the expressed preferences. The chatbot uses “soft” consensus because unanimous agreement can be difficult to achieve, especially in numerous groups or with experts with dissimilar backgrounds. Soft consensus models [41] permit measuring the degree of consensus in a group, provide feedback to each participant on the current consensus, and iterate to improve the consensus and converge towards a shared consensus threshold.

When all voters have indicated their preferences or when a predefined deadline is reached, the system aggregates all answers into a collective preference vector, and computes a global ranking for the alternatives, and a consensus measure ranging from 0 to 1. If the consensus is below a threshold (0.75 by default), another iteration is performed. When the consensus reaches the threshold the most preferred branch is integrated with the main model trunk, and the other branches closed. The branches and voting results can be consulted in the project history.

3.2.3 Tool Support

The presented approach has been realized in a chatbot called Socio. It works on Telegram [98] and Twitter [100], though it is designed to be extensible with further social networks and NL rules. The bot stores meta-models in Ecore format, and the traceability data as EMF [95] models. The pictures of meta-models are generated with PlantUML [80]. A video showcasing its use and some examples are available at https://saraperezsoler.github.io/SOCIO/.

Socio has an extensible architecture based on web services. This architecture allows the extension to different social networks without changes in the source code or stopping the main service. Also, it allows the use of different programming languages for each social network. The main part of Socio is a REST service, which is implemented in Java using the Jersey framework [42]. Chatbots for the different social networks communicate with the main service using HTTP requests. In addition to Telegram and Twitter, four final degree projects directed by myself [63, 34, 86, 19] have extended Socio to other social networks like Skype [90] or Slack [91].

Figure 3.6 shows some screenshots in the interaction with Socio to build and validate a meta-model for e-learning systems. In the first place (not shown), a Telegram group that includes the participants and the bot needs to be created. In Figure 3.6a, the bot shows all available commands, and then, one participant creates a modelling project using the /newproject command.

In Figure 3.6b, a participant sends a NL message to the chatbot using the command /talk. The chatbot interprets the message to deduce domain requirements, updates the current meta-model version accordingly, and returns a picture of the updated meta-model with the created and updated elements highlighted in green. Figure 3.6c shows a similar interaction using Twitter. In this case, the bot username (@ModellingBot) and the project name (learningplatform) need to be mentioned. The created attribute (code) is shown in green. When models are too large to be seen comfortably in an image, Socio instead of the picture of the complete meta-model, returns a picture of the updated elements and its context. After some interactions, one participant validates the model (Figure 3.6d), and the bot reports an error because the type of attribute PaidCourse.price is missing. At any moment, the meta-model can be downloaded in Ecore format using command /get (Figure 3.7a). The downloaded meta-model can then be used within Eclipse (Figure 3.7b).

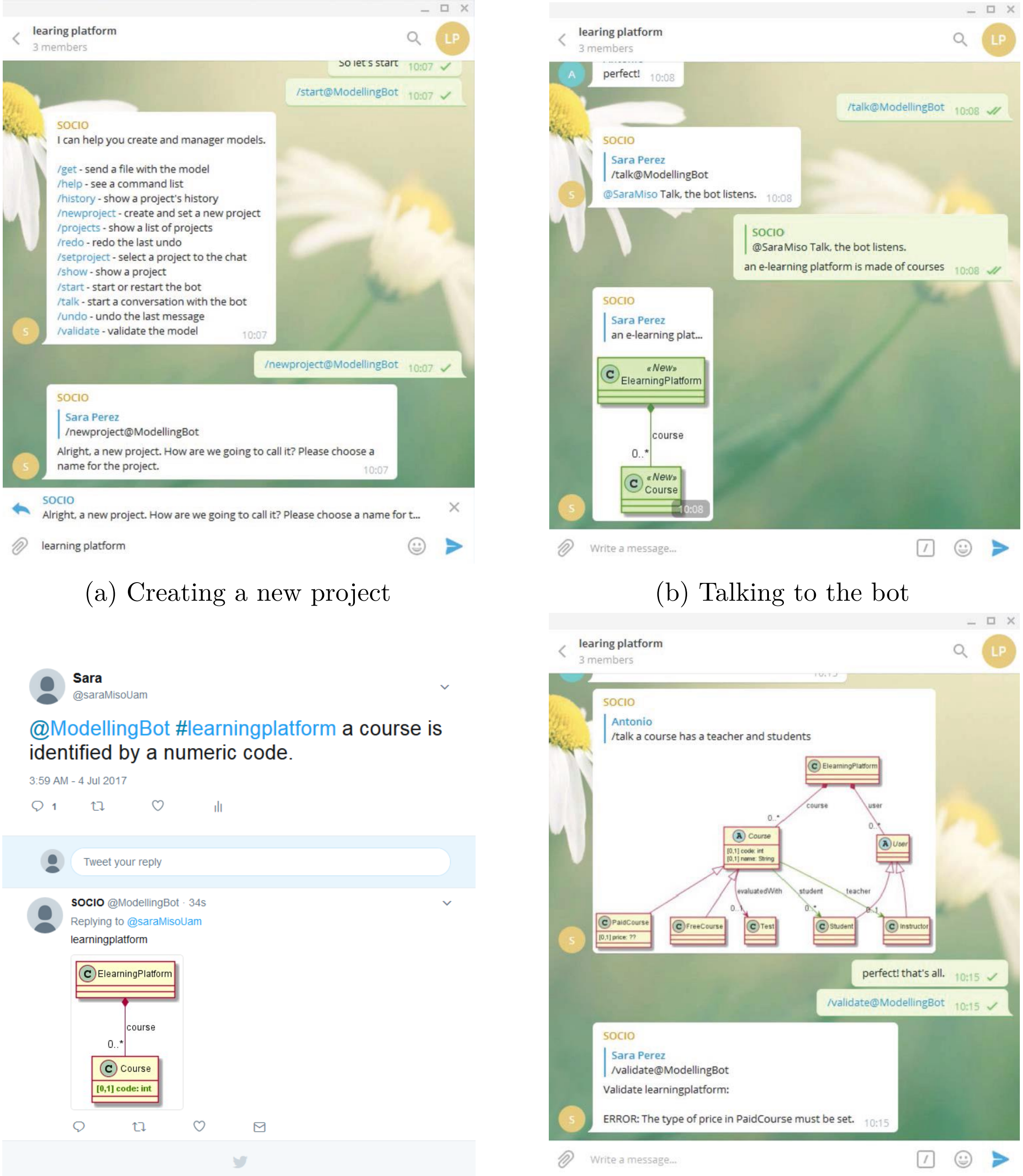
The interaction with SOCIO is recorded in a traceability model. This can be used to understand the rationale of every decision and analyse user contributions. In particular, SOCIO offers the following statistics: messages sent by one or all users, meta-model update actions done by one or all users, and percentage of meta-model authorship. They are available through the /history command (see Figure 3.8a). As an example, Figure 3.8b shows the number of messages directed to the bot from all users along time, while Figure 3.8c shows the percentage of authorship. In addition, it is also possible to obtain a more detailed history of the messages sent by each user and their consequences.

Figure 3.9a shows the usage of the /recommender command in Telegram. The modelling session is about a *School* that contains a list of *Student*s and *Teacher*s. Teachers have a name and surname. When a user types the command (label 1), SOCIO displays the current model and prompts the user to select a class (label 2). Once the user selects a class (label 3), SOCIO asks the kind of items to be recommended (label 4). The user can choose the recommendation of attributes and superclasses.

Figure 3.9b illustrates the recommendations provided by Droid. It shows the recommended superclasses (label 1) and attributes (label 2) for the class *Teacher*. When the user presses the button with the recommendation *Person*, SOCIO creates a new class because it does not exist, and adds it as a superclass of *Teacher*. When the user presses the button with the recommendation *name*, SOCIO detects that *Teacher* already defines this attribute and only updates its type. This way, recommendations not only add new elements to the model, but sometimes also allow fixing incomplete elements

3.3 Chatbots for Domain-Specific Modelling

This section explores the use of chatbots in domain-specific tasks. More specifically, it proposes two approaches: the use of chatbots for domain-specific modelling tasks (Section 3.3.1) and the use of chatbots for querying models (Section 3.3.2).



learing platform
3 members

So let's start 10:07

/start@ModellingBot 10:07

SOCIO
I can help you create and manager models.

/get - send a file with the model
/help - see a command list
/history - show a project's history
/newproject - create and set a new project
/projects - show a list of projects
/redo - redo the last undo
/setproject - select a project to the chat
/show - show a project
/start - start or restart the bot
/talk - start a conversation with the bot
/undo - undo the last message
/validate - validate the model 10:07

/newproject@ModellingBot 10:07

SOCIO
Sara Perez
/newproject@ModellingBot
Alright, a new project. How are we going to call it? Please choose a name for the project. 10:07

SOCIO
Alright, a new project. How are we going to call it? Please choose a name for t...

learning platform

(a) Creating a new project

learing platform
3 members

perfect! 10:08

/talk@ModellingBot 10:08

SOCIO
Sara Perez
/talk@ModellingBot
@SaraMiso Talk, the bot listens. 10:08

SOCIO
@SaraMiso Talk, the bot listens.
an e-learning platform is made of courses 10:08

SOCIO
Sara Perez
an e-learning plat...

Write a message...

(b) Talking to the bot

Sara
@saraMisoUam

@ModellingBot #learningplatform a course is identified by a numeric code.

3:59 AM - 4 Jul 2017

Tweet your reply

SOCIO @ModellingBot · 34s
Replying to @saraMisoUam
learningplatform

(c) Interaction through Twitter

learing platform
3 members

SOCIO
Antonio
/talk a course has a teacher and students

perfect! that's all. 10:15

/validate@ModellingBot 10:15

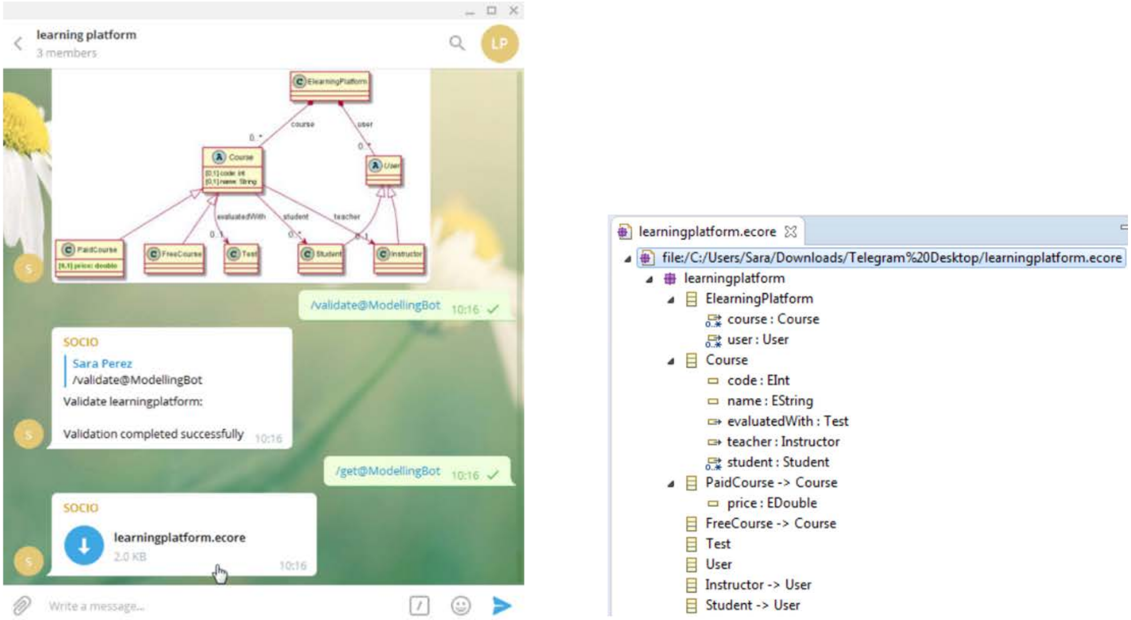
SOCIO
Sara Perez
/validate@ModellingBot
Validate learningplatform:

ERROR: The type of price in PaidCourse must be set. 10:15

Write a message...

(d) Meta-model validation

Figure 3.6 Interaction with Socio



learning platform
3 members

/validate@ModellingBot 10:16

SOCIO
Sara Perez
/validate@ModellingBot
Validate learningplatform:

Validation completed successfully 10:16

/get@ModellingBot 10:16

SOCIO
learningplatform.ecore
2.0 KB 10:16

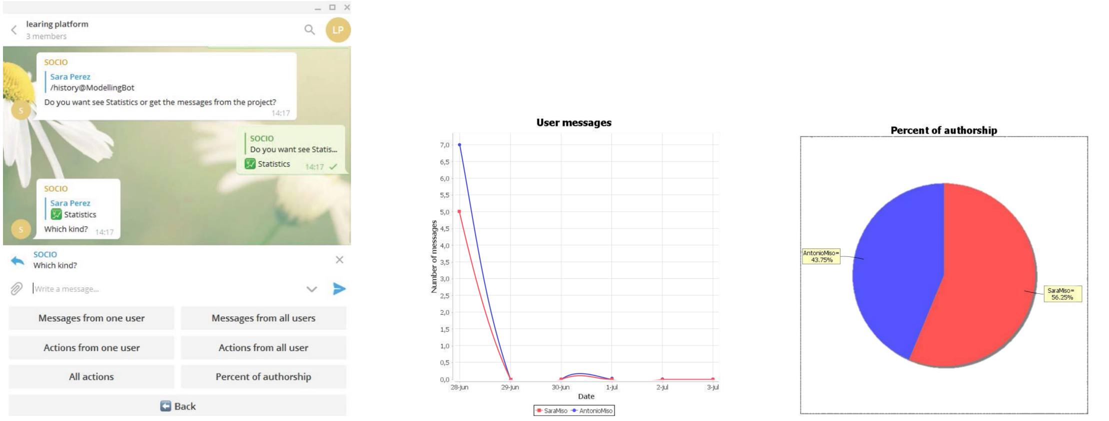
Write a message...

learningplatform.ecore
- file:/C:/Users/Sara/Downloads/Telegram%20Desktop/learningplatform.ecore
  - learningplatform
    - ElearningPlatform
      - course : Course
      - user : User
    - Course
      - code : EInt
      - name : EString
      - evaluatedWith : Test
      - teacher : Instructor
      - student : Student
    - PaidCourse -> Course
      - price : EDouble
    - FreeCourse -> Course
    - Test
    - User
    - Instructor -> User
    - Student -> User

(a) Downloading Ecore meta-model

(b) Ecore meta-model in Eclipse

Figure 3.7 Obtaining the final meta-model



learing platform
3 members

SOCIO
Sara Perez
/history@ModellingBot
Do you want see Statistics or get the messages from the project?
14:17

SOCIO
Do you want see Statis...
Statistics 14:17

SOCIO
Sara Perez
Statistics
Which kind? 14:17

SOCIO
Which kind?

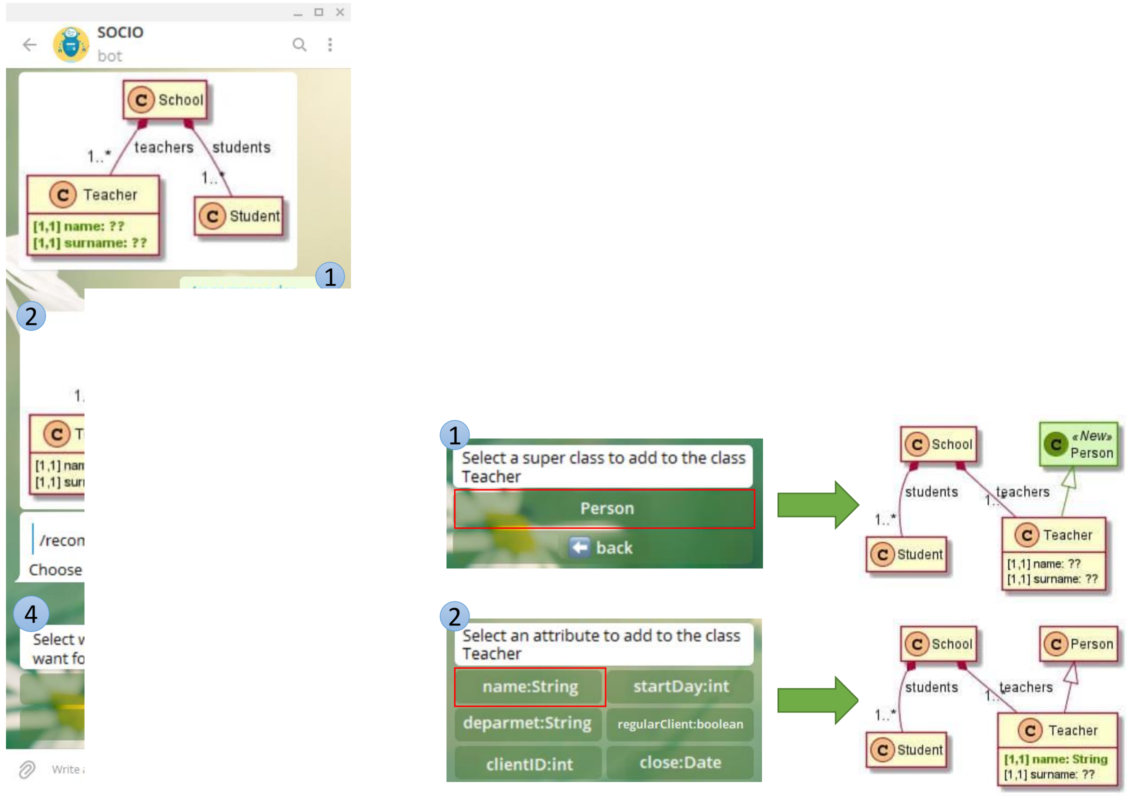
Write a message...

(a) Selecting statistics

(b) Messages sent by users

(c) Percentage of authorship

Figure 3.8 Some process statistics



Select a super class to add to the class Teacher

Select an attribute to add to the class Teacher

(a) Recommender command

(b) Droid recommendations in SOCIO

Figure 3.9 Obtaining recommendations in SOCIO with Droid

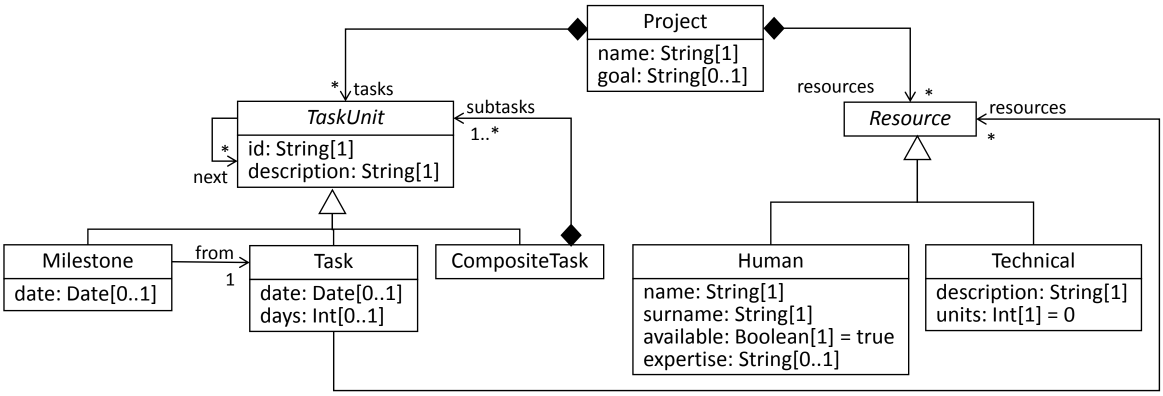


Figure 3.10 Running example: A domain meta-model to describe project plans.

3.3.1 Domain-Specific Modelling in NL

The objective of the work presented in this section 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, 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) [1]. Finally, this approach can be used to automate the generation of chatbot interfaces for existing information systems.

The previous section proposed a chatbot called Socio to assist in the creation of meta-models (i.e., class diagrams) via conversation. This section proposes a methodology and prototype tool support to create NL concrete syntaxes for arbitrary meta-models (i.e., not limited to class diagrams).

As a running example, the section will illustrate how to build a chatbot to define project plans conformant to the meta-model shown in Figure 3.10. *Project*s have a *name* and optionally a *goal*. They comprise a number of *TaskUnit*s that can be organised in sequences through reference *next*, and have an *id*. There are three kinds of task units: *Task*s, which may have a start date and end after a number days; *Milestone*s, which may have a start date but no duration, and are related to exactly one task; and *CompositeTask*s 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.

[1] A term coined by Satya Nadella, CEO of Microsoft.

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.

The aim of this thesis 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 human resources in the example).

3.3.1.1 Conversational Syntax for DSLs

To simplify the creation of modelling chatbots, Figure 3.11 shows an automated process.

As usual in MDE, a domain meta-model is used to describe the abstract syntax of the DSL. With regards to its concrete syntax, the conversational syntax is defined based on a meta-model, similarly to when it is graphical or textual. To facilitate this definition, first, a default configuration of the NL syntax is derived from the domain meta-model. This configuration declares how to refer to objects and features of the instantiable classes. 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.

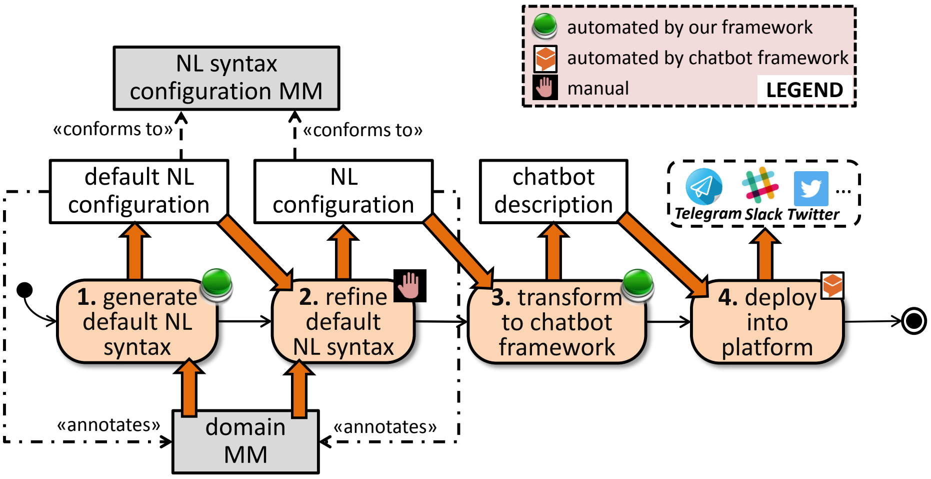

automated by our framework
automated by chatbot framework
manual
LEGEND
NL syntax configuration MM
«conforms to»
«conforms to»
default NL configuration
NL configuration
chatbot description
Telegram Slack Twitter ...
1. generate default NL syntax
2. refine default NL syntax
3. transform to chatbot framework
4. deploy into platform
«annotates»
domain MM
«annotates»

Figure 3.11 Steps for creating a modelling chatbot with our approach.

Once the conversational syntax is ready, a framework synthesizes a chatbot description from it, and generates source files for some chatbot tool (e.g., Dialogflow, Rasa or Pandorabots). Currently, the chatbot generates files for Dialogflow, but the approach can be adapted to work with other chatbot frameworks that provide similar concepts. As Section 3.3.1.5 will show, the deployed chatbot interacts with a modelling service, created to handle the model modifications at the abstract syntax level (e.g., object creation and deletion).

In the following, Section 3.3.1.2 presents the meta-model to describe the NL concrete syntax, Section 3.3.1.3 shows the generations of the chatbot description, and Section 3.3.1.4 presents a flexible modelling approach which allows saving incomplete or incorrect information in a model, waiting for its later refinement.

3.3.1.2 Configuring the NL Concrete Syntax

Figure 3.12 shows the 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 EMF [95] is used as meta-modelling technology, the classes in the NL syntax configuration meta-model refer to the *EPackage*s, *EClass*es, *EAttribute*s and *EReference*s in the domain meta-model.

NLModel is the root class. It contains one *NLClass* for each domain meta-model 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 *Identifier*s that will be used to refer to the objects of the class. An object identifier may consist of one or more attributes of its

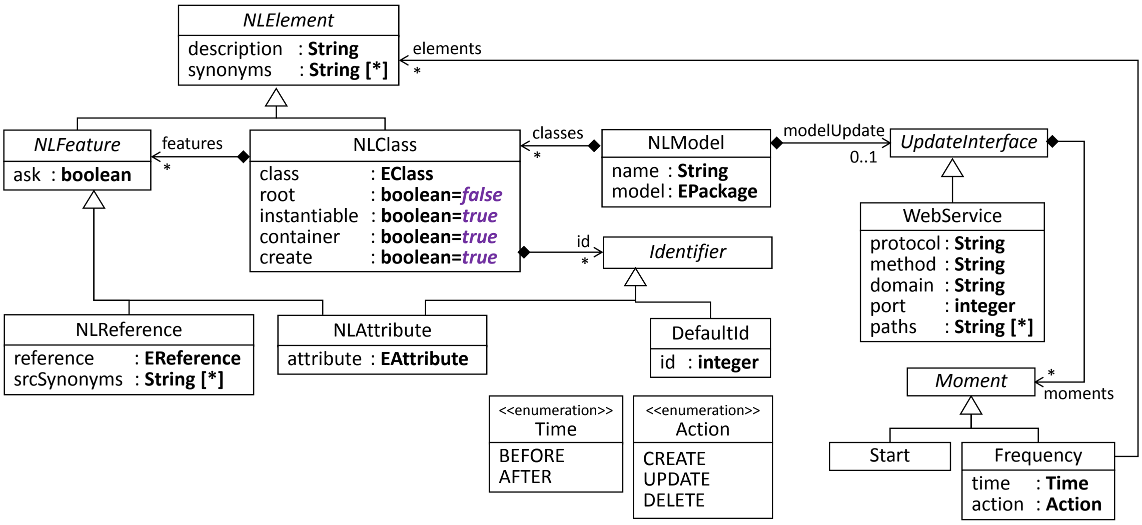


Figure 3.12 The meta-model for configuring the NL syntax.

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 to 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, the retrieval of external objects through *WebService*s are also supported . For this purpose, it is necessary to specify the *protocol*, *domain*, *port*, *method* and *paths* of the web service; and to configure the *Moment*s 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, a default NL configuration model is automatically produced. This contains one *NLClass* for each domain class, and one

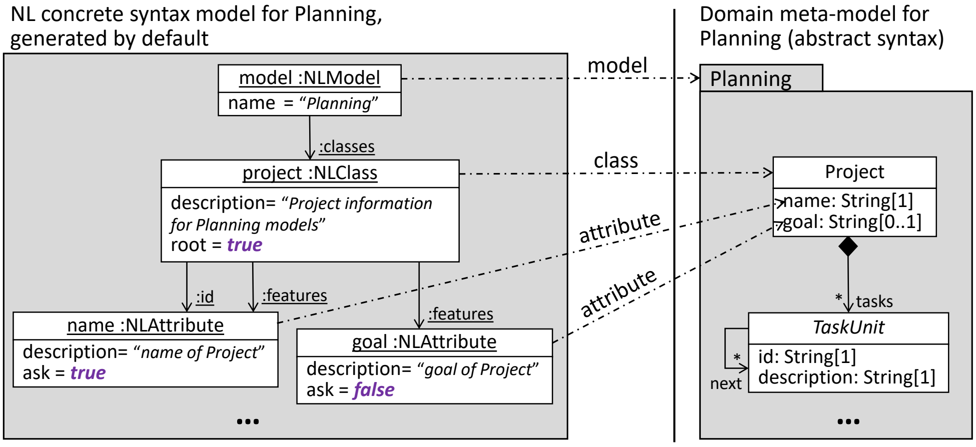


Figure 3.13 Excerpt of default NL concrete syntax model for the running example.

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*. All classes are marked 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.

Figure 3.13 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. Object *model* with type *NLModel* represents the model, and points to the *EPackage* containing the domain meta-model. 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.

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 needs 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).

In the 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*).

3.3.1.3 Generation of Chatbot Definition

Starting from the refined NL syntax configuration model, a chatbot description model for Dialogflow is generated.

Next, the intents for a generated chatbot are explained. The chatbots rely on an external modelling service to perform the model modifications at the abstract syntax level.

Welcome intent. Each chatbot contains a *welcome* intent that is trained with typical greeting phrases (e.g., *"hello"*, *"hi"*, *"hey"*, *"hi there"...*). 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 text, meaning that it can receive anything, and it invokes the REST web service 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 a followup intent called *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.

Specifically, two intents for each instantiable class are created, 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 3 shows the template used to synthesize training phrases for creating objects of a class and initialize 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 keywords 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, it comes the optional assignment of 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). The meta-model integrity constraints are not taken into account for generating these sample values, as they are not required to train the NL processor. Instead, correctness of values is checked at runtime at the abstract syntax level by the modelling service.

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

Listing 3 Template to synthesize training NL phrases for creating objects.

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. String is the predefined entity for text (*sys.any* in Dialogflow), Integer/Long and Double/Float is the predefined entity for numbers (*sys.number-integer* and *sys.number* in Dialogflow), and Date is the predefined entity for date (*sys.date-time* in Dialogflow). In addition, a custom-made entity is created 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"*...). This is because, when asking a value for boolean parameters, the answers typically have this form.

The object creation intents send all data collected to the external modelling service to create the object.

Similar to object creation intents, other intents are created to delete objects, modify features (attributes and references) and for the creation of non-instantiable classes with instantiable children. They are explained in detail in [73].

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"*. In feature modification intents, there are training phrases like: *"the units of technical pcs are 4"*, *"Peter's surname is Parker"*, *"set date of t2 to May 24th"*, *"Peter participates in task t2"*, and *"task t2 follows task t1"*.

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

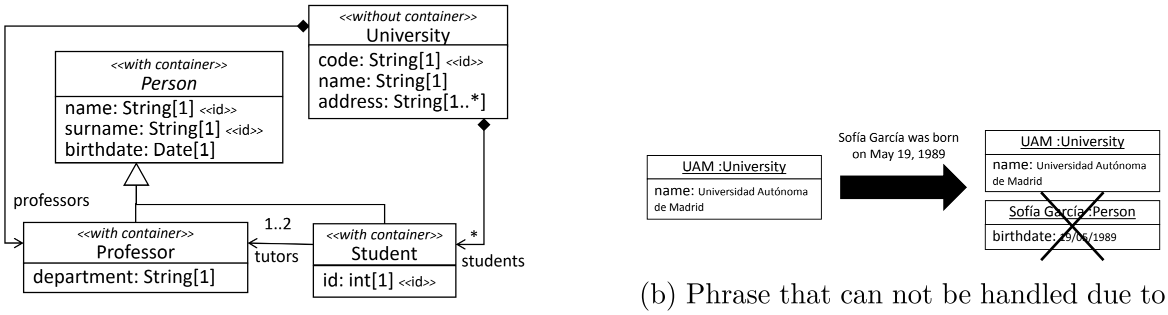
3.3.1.4 Flexible Modelling

When using NL, people normally do not provide all the required information in their phrases. Moreover, it is natural to let users express their ideas in a more free way, which can be refined later. For this reason, a flexible modelling approach was devised to enhance the conversation-based modelling presented in the previous section. This approach allows saving incomplete or incorrect information in a model, which can be refined later.

Figure 3.14a shows a meta-model and the configuration provided by the designer of the chatbot. This example is a meta-model of a University. The root class is *University* and all classes are marked as *instatiable* and *create*. The other elements of the configuration are represented with stereotypes, that is, they are between the symbols « and ». 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*t 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, as explained in the previous section.

When the agent processes the phrase "Sofía García was born on May 19, 1989" (Figure 3.14b), it infers that there is an object of type *Person* with *name* "Sofía",



(a) University meta-model.

(b) Phrase that can not be handled due to the rigidity of the meta-model.

Figure 3.14 An example of a configured meta-model (a) and a phrase that can not be handled for the generated chatbot (b)

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.

To allow the agent to save partial or incorrect information in the model, the meta-model with which the user will work is relaxed. This relaxation takes place in the modelling service.

Figure 3.15 displays the steps followed 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 their 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.

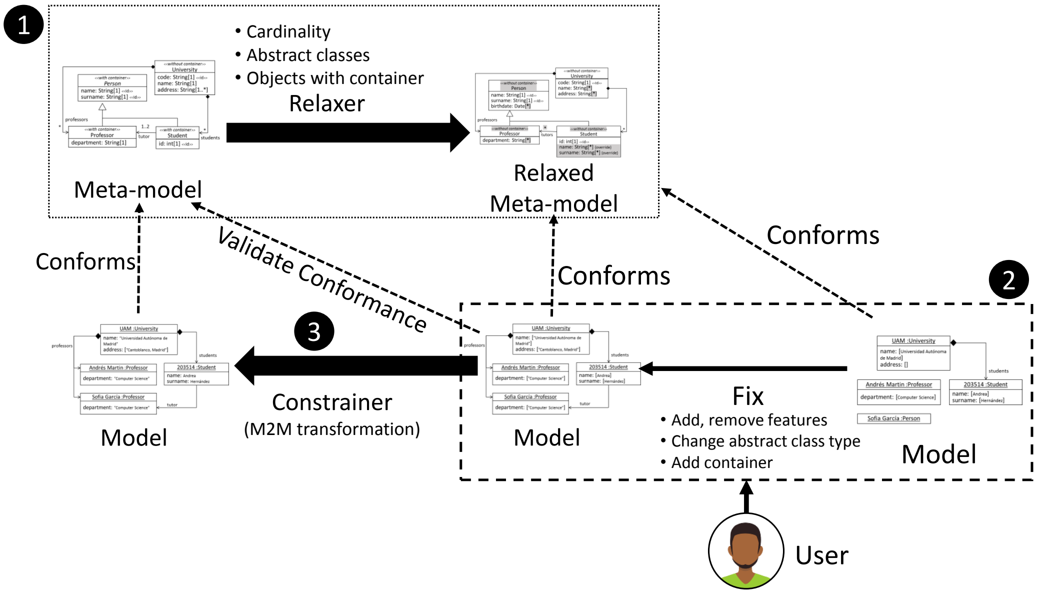
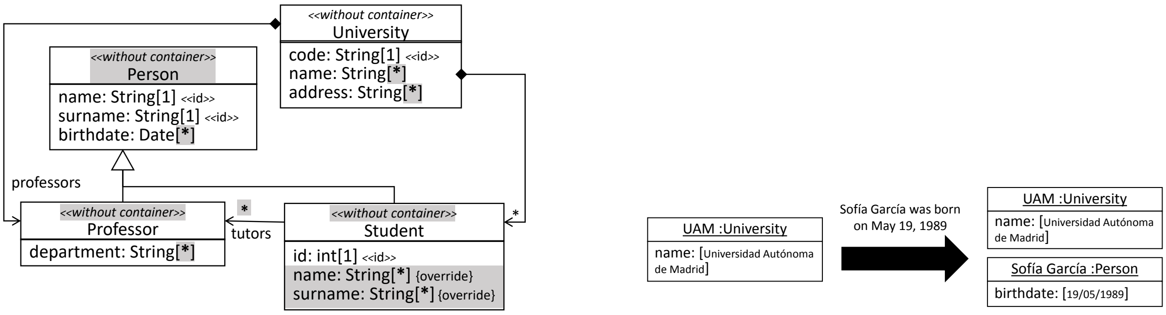


Figure 3.15 Steps in flexible modelling. (1) Meta-model relaxation (2) User interaction to create a valid model. (3) Model constrainer

- **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 EMF [95], the technology used to modelling. 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 3.16a shows the relaxed meta-model from Figure 3.14a 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



(a) Meta-model relaxation example.

(b) Example of model creation in NL.

Figure 3.16 An example of meta-model relaxation (a) and model creation with the relaxation (b)

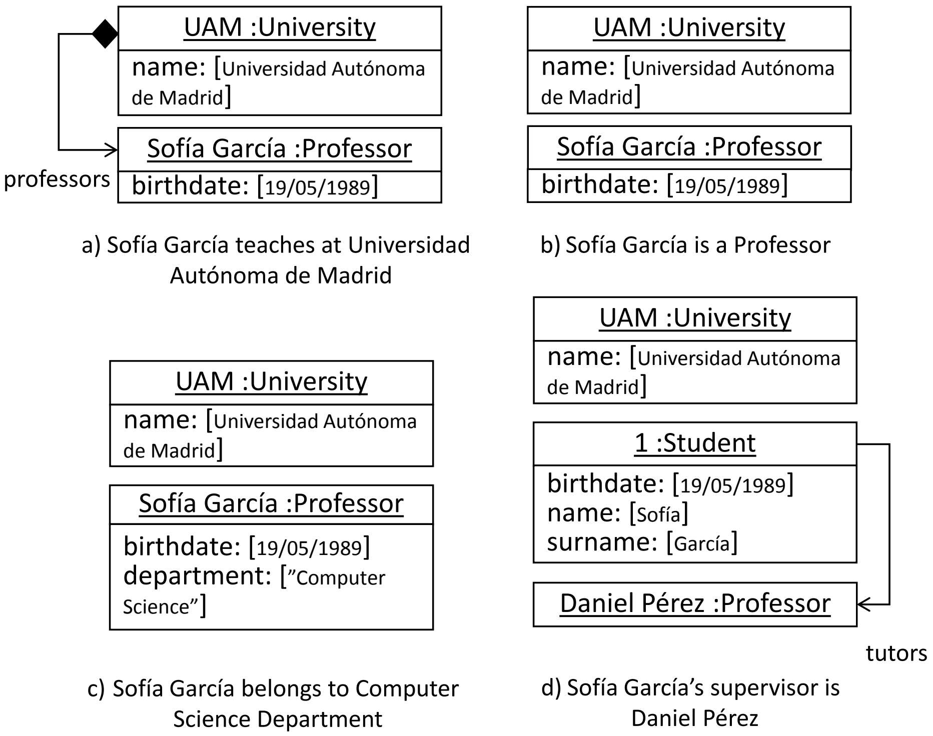
all properties that are not class identifiers are set cardinality [0..*]. The *Student* features *name* and *surname* must be overriden to increase the cardinality, because in *Person* and in *Professor* their cardinality must be [1..1], since they are identifiers.

Figure 3.16b shows an example of a message in NL and how it is interpreted to generate the model according to the meta-model of Figure 3.16a. 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 the proposed flexible modelling approach the agent creates a *Person* object to save all the information provided by the user, and waits for the rest.

Figure 3.17 displays several ways to make the object type concrete. The most direct one is that the user says the type explicitly (Figure 3.17b) 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 3.17a), when the user sets feature *department*, which belongs to *Professor* (Figure 3.17c) or *tutors*, which belongs to *Student* (Figure 3.17d). Moreover, the phrase *"Sofía García's supervisor is Daniel Pérez"* creates Daniel Pérez as *Professor* because only *professors* can be supervisors of students.

3.3.1.5 Tool Support

The previous approach has been implemented in a tool that automates the creation of modelling chatbots. The solution includes an EMF implementation of the meta-model in Figure 3.12 for configuring the NL syntax, an Eclipse plugin that instantiates this meta-model for a given domain meta-model, and a transformer into Dialogflow.



a) Sofía García teaches at Universidad Autónoma de Madrid

b) Sofía García is a Professor

c) Sofía García belongs to Computer Science Department

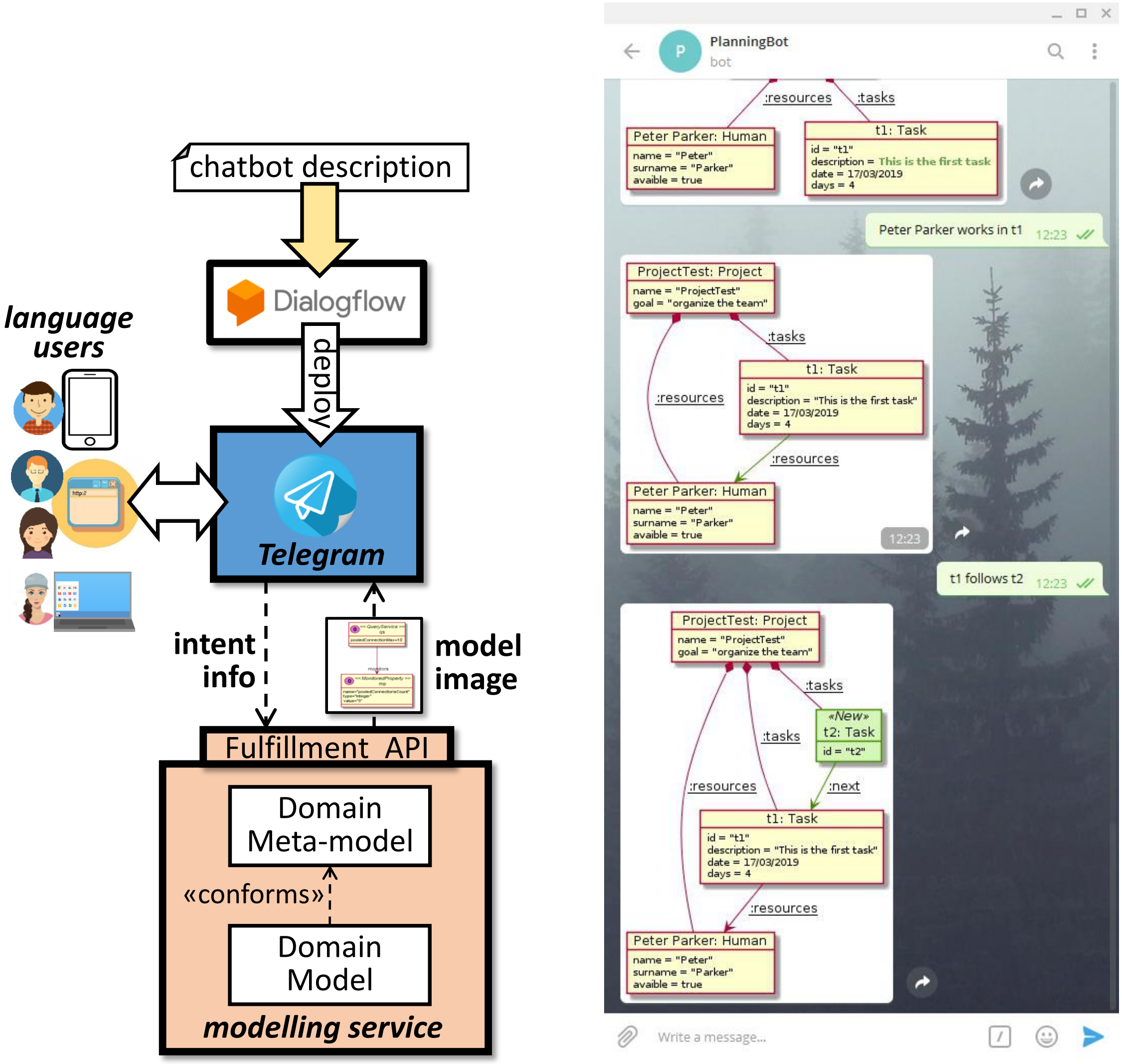
d) Sofía García's supervisor is Daniel Pérez

Figure 3.17 Four examples to type a Person object

Figure 3.18a shows the runtime architecture of the generated modelling 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 [80]. The image highlights the elements that have been modified in green. The */validate* command shows possible inconsistencies in the model, which then the users can correct.

Figure 3.18b 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*



Peter Parker works in t1 12:23

t1 follows t2 12:23

(a) Run-time architecture.

(b) Interaction in Telegram.

Figure 3.18 Modelling chatbots.

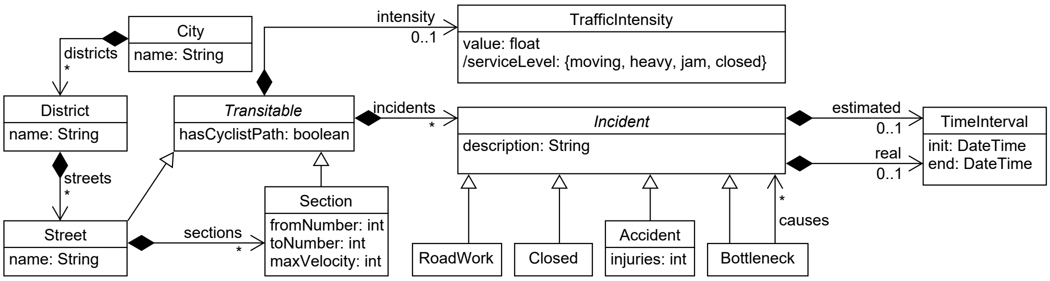


Figure 3.19 A meta-model for real-time traffic information.

Parker (*name* and *surname*) and the *Task* object with identifier *t1* (*id*). Then, the user inputs the sentence *"t1 follows t"*, 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/SOCIO/.

3.3.2 Chatbots for Querying Models

The present section extends the use of chatbots for domain modelling (Section 3.3.1), to support NL conversational queries over the models. This is a more accessible and user-friendly way to query models than the use of technical languages like OCL [64]. Moreover, we avoid the manual programming of the model query chatbots by their automatic synthesis. For this purpose, the solution devised is based on (i) the availability of a meta-model describing the structure of the models, (ii) its configuration with NL information (class name synonyms, names for reverse associations, etc.), and (iii) the automatic generation of a chatbot supporting queries over instances of the given meta-model. This approach is implemented on top of the Xatkit model-based chatbot development platform [30], which interprets the generated chatbot model and interacts with an EMF [95] backend.

As a motivating example, assume a city hall would like to provide open access to its real-time traffic information system. Given the growth of the open data movement, this is a common scenario in many cities, like Barcelona [10] or Madrid [52].

The data provided includes a static part made of the different districts and their streets, with information on the speed limits. In addition, a dynamic part updated in real-time decorates the streets and their segments with traffic intensity values and incidents (road works, street closings, accidents or bottlenecks). Figure 3.19 shows a meta-model capturing the structure of the provided information.

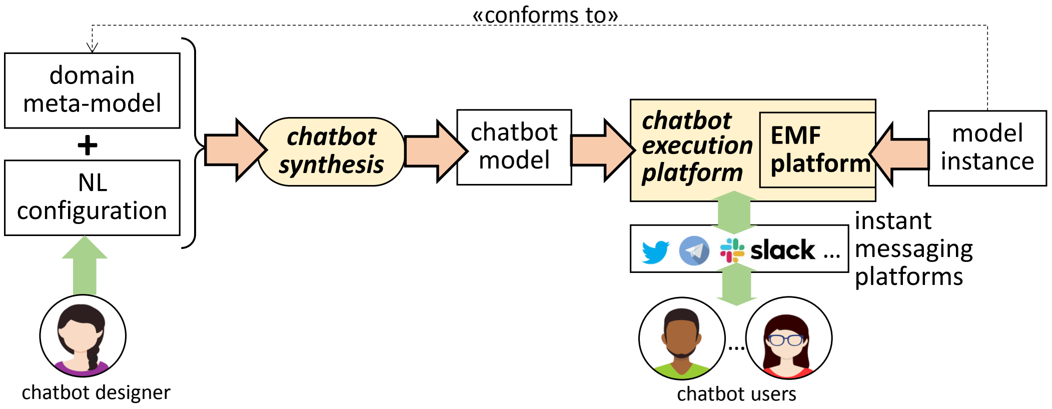


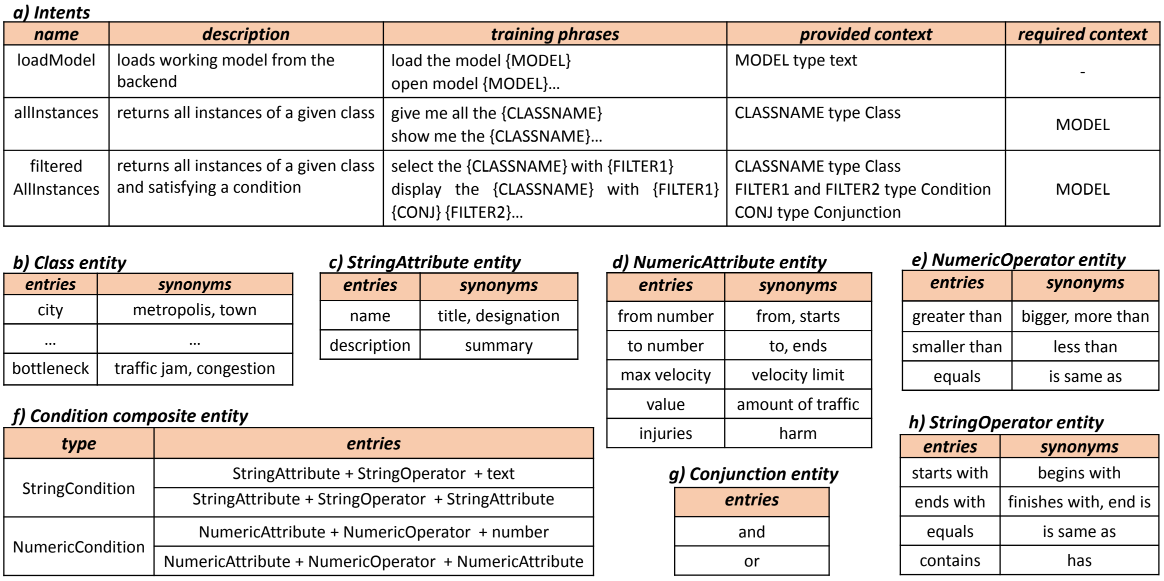
Figure 3.20 Scheme of the proposed approach to generate chatbots for querying models.

In this scenario, citizens would benefit from user-friendly ways to query those traffic models. However, instead of relying on the construction of dedicated front-ends with fixed queries, or on the use of complex model query languages like OCL, the thesis proposes the use of conversational queries based on NL via chatbots. Chatbots can be used from widely used social networks, like Telegram or Twitter, facilitating their use by citizens. Hence, citizens would be able to issue simple queries like *"give me all accidents with more than one injury"*; and also conversational queries like *"what are the incidents in Castellana Street now?"*, and upon the chatbot reply, focus on a subset of the results with *"select those that are accidents"*. Finally, for the case of dynamic models, reactive queries like *"ping me when Castellana Street closes"* would be possible.

The proposal generates a dedicated query chatbot given the domain meta-model. Figure 3.20 shows its scheme. First, the chatbot designer needs to provide a domain meta-model (like the one in Figure 3.19) defining the structure of the models to be queried, and complemented with NL hints on how to refer to its classes and features (synonyms). From this information, an executable chatbot model that can be used to query model instances is generated. The next subsections explain these two steps.

3.3.2.1 Chatbot Generation: Intents and Entities Model

The chatbot designer has to provide a domain meta-model and optionally, a NL configuration model. The latter is used to optionally annotate classes, attributes and features with synonyms, and the source of references with a name to refer to its backward navigation. From this information, the chatbot intents and entities are generated.



a) Intents

name	description	training phrases	provided context	required context
loadModel	loads working model from the backend	load the model {MODEL} open model {MODEL}…	MODEL type text	-
allInstances	returns all instances of a given class	give me all the {CLASSNAME} show me the {CLASSNAME}…	CLASSNAME type Class	MODEL
filtered AllInstances	returns all instances of a given class and satisfying a condition	select the {CLASSNAME} with {FILTER1} display the {CLASSNAME} with {FILTER1} {CONJ} {FILTER2}…	CLASSNAME type Class FILTER1 and FILTER2 type Condition CONJ type Conjunction	MODEL

b) Class entity

entries	synonyms
city	metropolis, town
…	…
bottleneck	traffic jam, congestion

c) StringAttribute entity

entries	synonyms
name	title, designation
description	summary

d) NumericAttribute entity

entries	synonyms
from number	from, starts
to number	to, ends
max velocity	velocity limit
value	amount of traffic
injuries	harm

e) NumericOperator entity

entries	synonyms
greater than	bigger, more than
smaller than	less than
equals	is same as

f) Condition composite entity

type	entries
StringCondition	StringAttribute + StringOperator + text
	StringAttribute + StringOperator + StringAttribute
NumericCondition	NumericAttribute + NumericOperator + number
	NumericAttribute + NumericOperator + NumericAttribute

g) Conjunction entity

entries
and
or

h) StringOperator entity

entries	synonyms
starts with	begins with
ends with	finishes with, end is
equals	is same as
contains	has

Figure 3.21 Intents and entities generated for the running example chatbot.

Table(a) of Figure 3.21 captures the generation of intents. An intent is created per query type, plus an additional intent called *loadModel* to select the model to be queried. The second row of the table shows the intent *allInstances*, which returns all objects of a given class. The intent is populated with training phrases that contain the class name as parameter. The possible class names are defined via an entity *Class* (see Table(b)). This intent would be selected on user utterances such as *"give me all cities"* or *"show every incident"*. The intent requires having a loaded model, which the table indicates as the intent requiring a model as context.

In the same table, intent *filteredAllInstances* returns all instances that satisfy a given condition. The intent is populated with training phrases that combine a class name and a condition made of one or more filters joined via logical connectives. An entity *Condition* for the filters is provided, as explained below. This intent would be selected upon receiving phrases like *"give me all accidents with more than one injury"* (please note the singular variation w.r.t. the attribute name *injuries*).

In addition to intents, several entities are created based on the domain meta-model and the NL configuration. Specifically, an entity named *Class* (Table(b)) is created with an entry for each meta-model class name. These entries may have synonyms, as provided by the NL configuration, to refer to the classes in a more flexible way. Likewise, an entity is created for each attribute name attending to their type: *String* (Table(c)), *Numeric* (Table(d)), *Boolean* and *Date* (omitted for brevity). For example, the *StringAttribute* entity (Table(c)) has an entry for all *String* attributes called *name*. Just like classes, these entries may have synonyms if provided in the NL configuration.

The *Condition* entity (Table(f)) is a composite one, i.e., its entries are made of one or more entities. This entity permits defining filter conditions in queries, such as *"name starts with Ma"* or *"injuries greater than one"*.

Regarding the complexity of the chatbot, the number of intents is fixed, and it depends on the primitives of the underlying query language that the chatbot exposes. Figure 3.21 exposes two primitives of OCL: *allInstances*, and *allInstances()→select(cond)*. Other query types can be added similarly, which would require defining further intents. The number of generated entities is also fixed, while the number of entries in each entity depends on the meta-model size and the synonyms defined in the NL configuration.

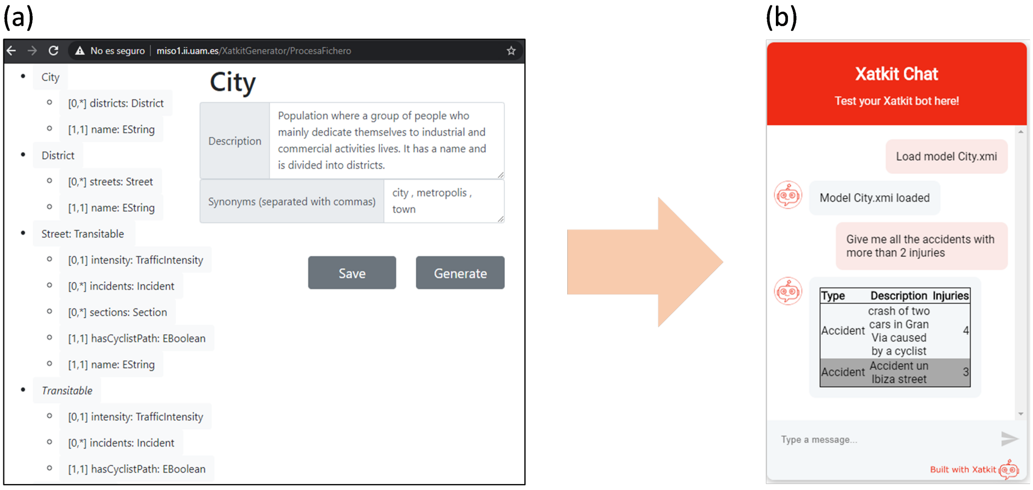
3.3.2.2 Chatbot Generation: Execution Model

The generated chatbot also contains actions, required to perform the query on a modelling backend, which we call the *execution model*. This execution model contains a set of *execution rules* that bind user intentions to response actions as part of the chatbot behaviour definition. For each intent in the *Intent* model, the corresponding execution rule in the execution model is generated using an event-based language that receives as input the recognized intent together with the set of parameter values matched by the NL engine during the analysis and classification of the user utterance.

All the execution rules follow the same process: the matched intent and the parameters are used to build an OCL-like query to collect the set of objects the user wants to retrieve. The intent determines the type of query to perform (e.g., *allInstances*, *select*, etc.), while the parameters identify the query parameters, predicates, and their composition. The query computation is delegated to the underlying modelling platform (see next section), and the returned model elements are processed to build a human-readable message that is finally posted to the user by the bot engine.

3.3.2.3 Proof of Concept

As a proof of concept, a prototype was created that produces Xatkit-based chatbots [30], following the two phases depicted in Figure 3.20. Xatkit is a model-driven solution to define and execute chatbots, which offers DSLs to define the bot intents, entities and actions. The execution of such chatbots relies on the Xatkit *runtime* engine. At its core, the engine is a Java library that implements all the execution logic available in the chatbot DSLs. Besides, a connector with Google's Dialogflow engine [31] takes



No es seguro | miso1.ii.uam.es/XatkitGenerator/ProcesaFichero

- City
  - [0,*] districts: District
  - [1,1] name: EString
- District
  - [0,*] streets: Street
  - [1,1] name: EString
- Street: Transitable
  - [0,1] intensity: TrafficIntensity
  - [0,*] incidents: Incident
  - [0,*] sections: Section
  - [1,1] hasCyclistPath: EBoolean
  - [1,1] name: EString
- *Transitable*
  - [0,1] intensity: TrafficIntensity
  - [0,*] incidents: Incident
  - [1,1] hasCyclistPath: EBoolean

City

Description: Population where a group of people who mainly dedicate themselves to industrial and commercial activities lives. It has a name and is divided into districts.

Synonyms (separated with commas): city , metropolis , town

Xatkit Chat

Test your Xatkit bot here!

Load model City.xmi

Model City.xmi loaded

Give me all the accidents with more than 2 injuries

Type	Description	Injuries
Accident	crash of two cars in Gran Via caused by a cyclist	4
Accident	Accident un Ibiza street	3

Type a message...

Built with Xatkit

Figure 3.22 (a) Web application to configure the chatbot. (b) A query in the generated chatbot.

care of matching the user utterances, and a number of platform components enable the communication between Xatkit and other external services.

For this thesis, a web application was developed, where domain meta-models (in *.ecore* format) can be uploaded, and then (optionally) configured with synonyms. Once the configuration is finished, the application synthesizes a Xatkit chatbot model, which then can be executed using the Xatkit runtime engine.

Figure 3.22(a) shows the web application on the left, where the running example meta-model (cf. Figure 3.19) is being configured. Figure 3.22(b) shows a moment in the execution of the generated Xatkit chatbot, and the result returned by the bot when processing the example utterance *"show all accidents with more than one injury"*.

3.4 Evaluation

This section reports on two user studies performed over Socio's modelling capabilities (Sections 3.4.1 and 3.4.2, details in [71] and [83]), another one over Socio's soft consensus mechanisms (Section 3.4.1, full details in [72]) and one use case evaluating the approach of chatbots for domain-specific modelling over a pre-existing information system (Section 3.4.3, details in [73]).

3.4.1 Preliminary User Studies of Socio

To assess the suitability of the proposed a modelling chatbot, we conducted a preliminary evaluation with 10 participants organized in 4 Telegram groups: 2 groups of 2 people, and 2 groups of 3 people. All participants had a computer science background (postgraduate or last year degree students) and were non-native English speakers. They were asked to create a meta-model for ecommerce in 15 minutes but with no other restriction, and then complete a questionnaire with 3 parts: two with Likertscale questions, and a last one with free text questions. More details can be found in [71].

The first part of the questionnaire consisted of the 10 questions of the System Usability Scale (SUS) [17], a de-facto standard to measure system usability. Socio obtained 74% which indicates good usability. The second part of the questionnaire comprised 8 questions evaluating four aspects: (1) suitability of NL to build models w.r.t. using an editor, (2) precision of the bot to interpret NL, (3) enough functionality in the command set, and (4) whether participants liked embedding a modelling tool in a social network, or they would prefer a separate collaborative tool. Aspects 1 and 4 obtained around 75%, indicating that participants considered NL as a suitable interaction mechanism, and they appreciated the idea of collaborating through social networks, while aspects 2 and 3 were rated with 62,5% and 60%, which is reasonable but improvable.

The study is preliminary, with several threats, like the low number of participants, the limited group size, the similar participant background, the fact that participants were non-native English speakers, and the lack of a precise modelling goal, which permitted evaluating the produced artefacts. However, the positive results encouraged further research on this approach.

To ease decision-making by a potentially large heterogeneous group, a soft-consensus mechanism was incorporated to measure the degree of agreement based on the group preferences, and avoid the bias that a human moderator may introduce [41] (Section 3.2.2). To assess this hypothesis, a small-scale evaluation was performed with 8 participants recruited from the Master and Doctoral programs of the Department of Computer Science of the Universidad Autónoma de Madrid [72]. 6 participants were computer scientists, 1 engineer in telecommunication, and 1 physicist. After attending a 10-minutes tutorial about Socio, they used it to select the best solution among three possibilities for two different projects, first without consensus mechanism, and then using it. Interestingly, without the consensus mechanism, they ended up organizing a public poll within Telegram, but discrepancies among participants remained until the end of the experiment. They also answered a five-point Likert scale survey on the consensus mechanism, which was considered especially useful for large groups (average 4.7/5), and with an outcome that reflected the opinion of the majority (4.8/5) and was deemed objective because of the private voting (4.3/5). More details are available in [71, 72].

3.4.2 User Study and Comparison of Socio and Creately

After the preliminary study described in Section 3.4.1 to assess the suitability of modelling bots, we conducted a major experimental study [83] with 54 participants, to evaluate the usability of Socio, comparing it with the on-line modelling tool Creately [28], which uses a traditional GUI, accessible through a web browser. The experiment was structured as 2 sequences and 2 periods within-subjects cross-over design.

The 54 participants were split into two groups of 27 participants each. The participants in each group were further divided into 9 teams. The teams were randomly created. A total of 18 teams participated in the experiment (9 per group). Each group applies the treatments in a different order (AB/BA). The treatments are two tools for creating class diagrams: the chatbot Socio and the web application Creately. Group 1 first applies Socio and then Creately. Conversely, group 2 first applies Creately and then Socio. Both groups implement the tasks in the same order (task 1 and task 2). Each task consists of the creation of a class diagram.

All the participants first received a brief tutorial about the tool they had to use. Then, they were required to perform the first task with the tool in a maximum of 30 minutes. At the end of the experimental session the subjects filled in a modified and validated satisfaction questionnaire SUS [17] associated with the tool. Once the questionnaire was completed, participants received a tutorial of the second tool. Then, they performed the second task with the tool in a maximum of 30 minutes. At the end of the allowed time the participants filled in another modified SUS satisfaction questionnaire, with questions about the tool. In this last questionnaire, the participants were asked if they preferred Socio or Creately.

Usability was determined by attributes of efficiency, effectiveness, satisfaction and quality of the results. The study reports that Socio saved time and reduced communication effort over Creately. Socio satisfied users to a greater extent than Creately, while in effectiveness results were similar. With respect to diagram quality, Socio outperformed Creately in terms of precision, while solutions with Creately had better recall and perceived success. However, in terms of accuracy and error scores, both tools were similar. More details can be found at [83].

After this user study, Socio was improved to provide more help when it does not understand users (Socio V1), and the help page was changed to provide more help to users. We conducted a family of three experiments to compare the usability of Socio V1 and Creately with 87 students. Students appeared to be more satisfied with Socio V1, and Socio V1 scored better on completeness. There were no significant differences between the two tools regarding efficiency and quality. More details are available in [84].

3.4.3 Use Case: Datalyzer

To ascertain the feasibility of chatbots for domain-specific modelling, the approach to develop a conversational front-end [73] was used for the pre-existing system Datalyzer [38]. Datalyzer 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 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?

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 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.

The approach (Section 3.3.1) was used to automatically generate a default NL concrete syntax model from the Datalyzer meta-model. Next, the NL model was manually refined to add synonyms and modified some configuration features (e.g., to change the data source class to non-instantiable). Starting from the modified concrete syntax model, a Dialogflow chatbot was produced.

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, a middleware was created to provide model management support (e.g., uploading Datalyzer models) and to support all commands available in the browser. The middleware is connected to the chatbot as a REST API, and implements model transformations, to transform the chatbots models in EMF to JSON models compatible with Datalyzer, and back. Also, the chatbot sends requests to the middleware to obtain the data source types, as this is a non-instantiable class. The middleware retrieves the data source types by sending a request to the Datalyzer REST API.

Next, we answer the research questions.

RQ1: Feasibility. This question can be answered positively: using our approach, a NL interface was easily added 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, the middleware was implemented to 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 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.

More details of this case study are available in [73].

3.5 Related Work

This section revises related works on the usage of NL processing techniques within MDE, generation of chatbots, and collaborative modelling.

NL processing in MDE. This thesis proposes using NL to conceptual and domain-specific model creations and model query. NL processing techniques have been used within Software Engineering to derive UML diagrams/conceptual models from text [6, 46]. Our contribution in this context is to use an interactive, incremental approach, and the use of social networks to embed assistance.

The ModelByVoice [49] 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 [93] 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 use a more flexible NLP approach to create conceptual models, which not only support commands but descriptive sentences of the system are allowed and support synonyms. Moreover, the domain-specific modelling approaches generate a flexible NL syntax adapted to the DSL, also supports synonyms, the conversation flow is configurable, and does not assume a diagrammatic model. Lastly, none of those two approaches support queries.

In the vision paper [18], 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. Droid recommenders can be integrated within Socio, as an external service, for that purpose, as we saw in Section 3.2.1.3.

Finally, DoMoBOT [87] is also a modelling chatbot that creates domain models from their descriptions in NL. It was published after Socio in 2022.

Generation of chatbots. Generating chatbots from domain models is largely unexplored. Almost no chatbot platform supports automatic chatbot generation from external data sources. A relevant exception is Microsoft QnA Maker [56], which generates bots for the Azure platform from FAQs and other well-structured textual information.

In [81], 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, this research is more concerned with the usage of NL as front-end of models, modelling services and information systems, and it provides tools support.

Castaldo and collaborators [21] propose generating chatbots for data exploration in relational databases, but requiring an annotated schema as starting point, while in our case providing synonyms is an optional step. Similarly, [89] integrates chatbots to service systems by annotating and linking the chatbot definition to the service models. In both cases, annotations and links must be manually created by the chatbot designer to generate the conversational elements. In contrast, domain-specific approaches generates automatically the configuration. In [101], chatbots are generated from OpenAPI specifications but the goal of such chatbots is helping the user in identifying the right API Endpoint.

Altogether, to our knowledge, there are no automatic approaches to the generation of flexible chatbots with model creation and query capabilities. Therefore, applying classical concepts from CRUD-like generators to the chatbot domain is a highly novel solution to add a conversational interface to any modelling language.

Collaborative modelling. Collaborative modelling has been used for model construction [37] and collaborative creation of DSLs [20]. However, these works do not use social networks or NL processing, but they rely on collaborative graphical model editors [37] or ad-hoc tools [20] without assistant support. Instead, our approach integrates a modelling bot within a social network, so that users do not need to switch between discussion, coordination and modelling tools.

Altogether, from the analysis of the state of the art, it can be concluded that the usage of NL and chatbots as front-end for modelling services, as proposed in this thesis, is highly novel.

3.6 Summary and Conclusions

This chapter explored the use of chatbots as front-ends for modelling services. First, Section 3.2 proposed an approach to collaborative conceptual modelling via social networks, with assistant chatbots able to process NL messages. Using NL as modelling interface has the advantage of lowering the entry barrier to modelling, and does not interrupt the group discussion flow because messages for discussion and modelling are intertwined. Moreover, any element included in the model is immediately justified by the NL message used for its creation, hence documenting its provenance. To enrich traceability and rationale of modelling decisions, we also produce a history model tracking user contributions to model elements. To ease decision-making by a potentially large heterogeneous group, we incorporate a soft-consensus mechanism to measure the degree of agreement based on the group preferences, and avoid the bias that a human moderator may introduce. Section 3.2.3 showed prototype tool support called Socio. We performed an initial evaluation of Socio concerning modelling in NL and collaboration based on soft-consensus mechanism (Section 3.4.1) with encouraging results. Then we performed a study with 54 participants (Section 3.4.2) to evaluate the usability of Socio with positive results.

Section 3.3 proposed approaches for domain-specific modelling in social networks using NL. On one hand, Section 3.3.1 presented an 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. A case study (Section 3.4.3) illustrated the functionality added by the chatbot, which includes support for collaboration in NL, multi-platform, mobility, and traceability. On the other hand, Section 3.3.2 proposed the automatic synthesis of chatbots able to query the instances of a domain meta-model.

Once we have presented the approach to use chatbots for modelling, the next chapter details the proposal for modelling of chatbots.

Chapter 4

Modelling of Chatbots

This chapter will explore the use of MDE techniques to help in the chatbot development process. The proliferation of chatbot has made emerge many tools for their construction with different features and possibilities. However, since there are so many options, choosing the most appropriate one to develop a chatbot with certain features is not easy. There may also be operational factors to consider in the decision, as for example, some options may imply vendor lock-in, and migrating chatbots between tools is not generally supported. Last but not least, some approaches have a steep learning curve and require expert knowledge.

To overcome these problems, Section 4.2 proposes a MDE approach to chatbot development. This relies on a meta-model with core primitives for chatbot design (Section 4.3.1), and a DSL to define bots independently of the implementation technology (Section 4.3.2). Section 4.4 presents DSL validators, which analyse the model for defects, generators for forward engineering, to produce the chatbot implementation from its specification, and parsers for reverse engineering, to produce a model out of a chatbot implementation, which can then be analysed, refactored and migrated to other platforms. Section 4.5 explains a creation tools recommender based on the chatbot definition and other requirements. A prototype tool has been implemented for the approach (Section 4.6). Section 4.7 reports on evaluations of the approach. Finally, Sections 4.9 and 4.10 compare with the related work and conclude the chapter respectively.

This chapter is based on publications [77, 76, 23, 22, 50]

4.1 Introduction and Motivation

The growing interest in developing chatbots has triggered the emergence of many platforms, tools and libraries for their construction [78]. Several of them were explained in Section 2.1.2. This rich plethora of tools brings chatbot developers a wide range of possibilities, but also complicates the decision of which would be the best option for a given scenario.

The proliferation of tools and approaches may cause the underlying conceptual design of a chatbot to become difficult to grasp, as it may get obscured by the accidental complexity imposed by the platform of choice (e.g., in the different configuration files and Python code in a Rasa chatbot; or in the different forms and scripts in Dialogflow). This fact may also hinder comparison of chatbots across tools. Finally, closed chatbot development tools may lead to vendor lock-in, and hence the developed chatbots may become difficult to migrate to other platforms. While current research and practice has focussed on creating chatbot development tools, there is scarce support for automating the migration across tools, or proposals for representing chatbot designs in a neutral way.

In order to attack these issues, this chapter presents a model-driven solution [16] for chatbot design and re-engineering. Our solution is based on a neutral chatbot design language called Conga. We developed this DSL based on the analysis of the concepts supported by fourteen chatbot development approaches (Section 2.1.2). Our solution does not involve a chatbot execution engine. Instead, a recommender system helps the user to select the most suitable implementation tool for the chatbot design, and code generators are able to synthesize the chatbot implementation for the tool of choice. To support migration, our solution also includes parsers from specific chatbot development tools into Conga. This way, the chatbot design can be reverse engineered from the implementation, improved if needed, and then migrated to a different tool using the code generators.

4.2 Overview of the Approach

This chapter proposes modelling chatbots independently of the chatbot development tool, and then enable forward and backward engineering of chatbot code from the designed chatbot models. This enables reasoning and validation of chatbot models prior to their implementation, keeping the chatbot design independent of a specific technology. Moreover, it facilitates coping with a variety of scenarios, including forward

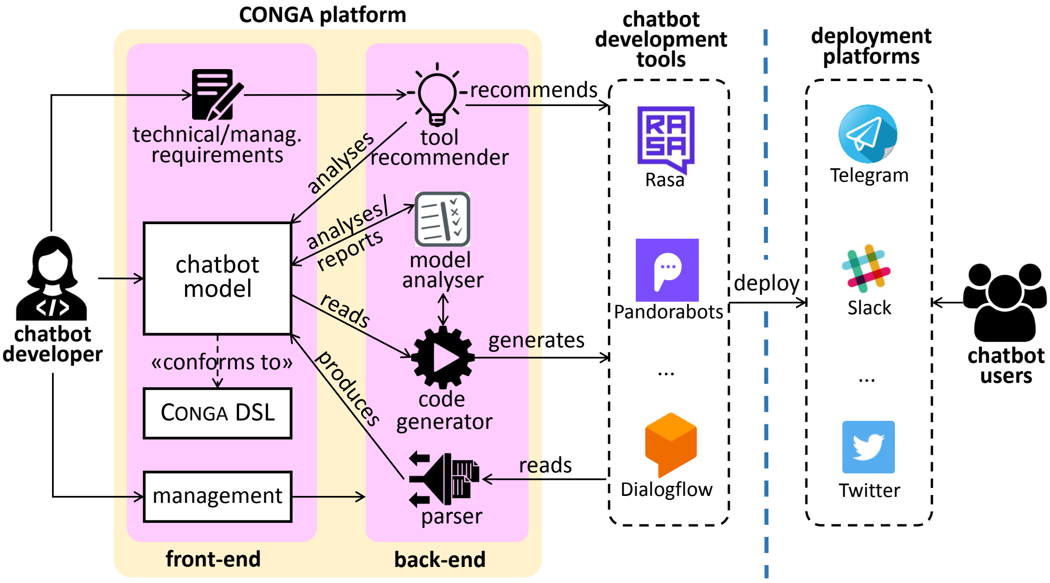


Figure 4.1 Schema of CONGA tool approach.

engineering of chatbots, chatbot evolution, chatbot maintenance, and chatbot migration between different technologies and between different versions of a same technology.

Figure 4.1 shows a scheme of our proposal. The main component is a web platform where chatbot developers can design chatbots using a technology-agnostic DSL called CONGA (ChatbOt modelliNg lanGuAge). The DSL is built based on a neutral meta-model resulting from an analysis of fourteen of the most prominent chatbot development tools used nowadays [78]. The DSL permits defining chatbot models independently of any development platform, and will be described in Section 4.3.

The DSL is complemented with code generators that synthesize code from the chatbot models into specific development tools (e.g., JSON files in the case of Dialogflow, or Python, markdown and YAML files in the case of Rasa). The generated chatbots can be deployed in a variety of channels (e.g., Telegram, Slack or Twitter) to make them available to chatbot users. The DSL also facilitates chatbot migration by the provision of parsers from several development tools into the DSL. In addition, chatbot models can be validated, checking general chatbot quality criteria and well-formedness rules. The architecture enables also to incorporate specific validations for particular platforms. Section 4.4 explains the parsing, generation and validation services, detailing the algorithms to generate, validate and parse code for Dialogflow and Rasa

To facilitate the task of selecting a development tool for implementing a given chatbot model, the CONGA platform incorporates an extensible recommender that analyses the chatbot model and other technical requirements to provide a ranked list of suitable tools. Section 4.5 explains the recommender system and its extensible architecture.

Finally, the CONGA platform incorporates a management module, by which chatbot developers can extend the platform with new code generators and parsers from/to other chatbot development tools, or new versions of tools already supported. Likewise, the platform permits registering new model analysers to perform validations specific to a development tool. Section 4.6 will explain the architectural decisions to enable this extensibility.

4.3 The Conga DSL

This section first introduces the abstract syntax of CONGA by explaining its meta-model (Section 4.3.1) and then describes its textual concrete syntax through an example (Section 4.3.2).

4.3.1 Abstract Syntax

Section 2.1 analysed fourteen chatbot tools and proposed an initial neutral meta-model to define chatbots. The main elements are *Intents* or the user intention in the communication, *Entities* which correspond with the domain elements, *Actions* that the chatbot can perform to answer the user, and the conversation *Flows*. Figure 4.2 shows the meta-model [76], divided into four different packages.

The *intents* package contains the intents definition. A *Chatbot* has a *name*, a list of supported *languages* (which allows defining multi-language chatbots), and a list of *intents*. Intents have a *name*, can be *fallback* and may define a set of *TrainingPhrases* per each supported language. Training phrases are examples of how a user can express an intention. For example, to express the intention to order a pizza, a training phrase could be "*I want a medium thin crust pizza*". Intents may also contain *Parameter*s, pieces of information that the chatbot needs to extract and store. In the previous example, the chatbot needs to save *medium* as the pizza size and *thin crust* as the dough type to allow the pizza store to manage the order. Parameters have a *name*, a *type*, can be multivalued (a *list*), can be *required* and may define a list of *prompts* to ask for a value when the parameter is required but the user utterance does not include its value. Parameter types can be predefined (enumeration *PredefEntity* with values *text*, *date*, *number*, *float* and *time*) or domain-specific ones using *Entities*.

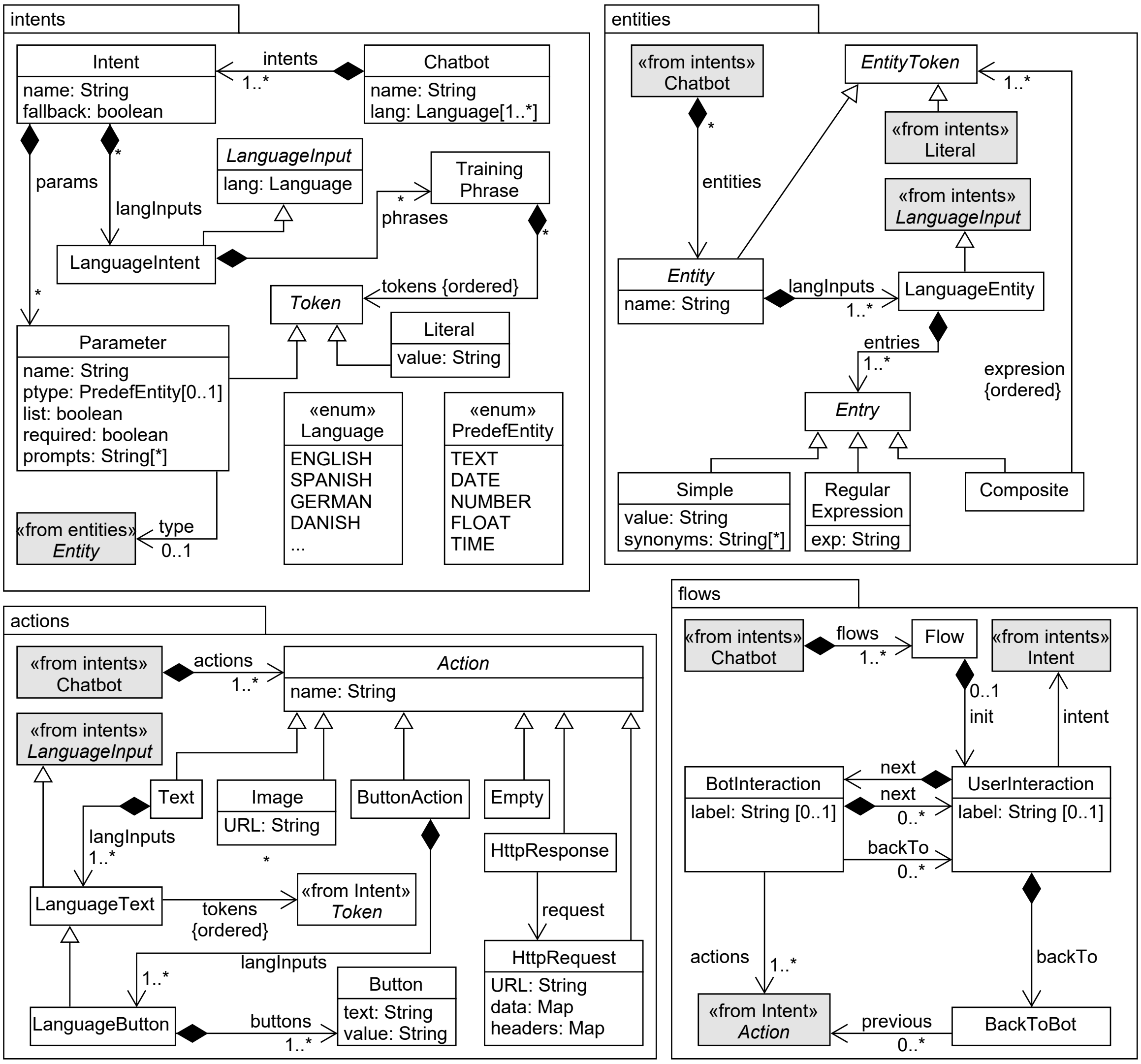


Figure 4.2 Meta-model for chatbot design.

Entities are defined in the *entities* package. They have a *name* and a set of *Entries* per each language. Entries can be *Simple*, consisting of a word and its synonyms, *Composite*, made of other entities and text, or *Regular Expression*. For example, in our pizza store chatbot, we may define simple entities for pizza size (small, medium, large) and the kind of pizza (Veggie, Cheese, Pepperoni...), a composite entity for products (⟨*pizzaSize*⟩ ⟨*pizzaKind*⟩ pizza) and a regular expression for the zip code in the delivery address.

The *actions* package defines six types of actions that the chatbot can perform. Sending a *Text* response, which may refer to parameters (e.g., "*Small cheese pizza order confirmed*"). Sending an *Image* using its *URL*. Sending a text with *Buttons* (e.g., to present each pizza ingredient as a button, facilitating the client's selection). Performing *HttpRequest* to a given *URL* and sending to the user an *HttpResponse* for a previous http request (e.g., to place an order to the pizza store and send a confirmation message). The last action type, *Empty*, is a wildcard for other platform-specific actions.

Finally, the *flows* package supports the definition of the conversation structure. The conversation alternates between the user and the chatbot turn (classes *UserInteraction* and *BotInteraction*). The user turns reference an intent, and the chatbot turns refer to a list of actions that the chatbot has to perform. In addition, we have provided support for loops. This way, *UserInteraction* and *BotInteraction* objects may have a *label*. In the DSL, these are useful to identify the target step of a loop. From a *BotInteraction* it is possible to go *back to* previous *UserInteractions*. Also, it is possible to go *back to* a *BotInteraction* from a *UserInteraction*, using the class *BackToBot*, which may have a list of *actions* to perform before going back.

4.3.2 Concrete Syntax

The CONGA DSL has a textual concrete syntax based on the meta-model described in the previous section. Listing 4 shows an example of a Pizza Store chatbot defined in CONGA. The definition starts with the name of the chatbot and the list of supported languages, in this case english (line 1).

The keyword *intents* (line 2) begins the intent definition. Every intent starts with a name and defines a number of training phrases. In case of multilanguage chatbots, each intent should provide training phrases for each language considered by the chatbot. The listing defines four intents: *StartOrder* in line 3 (to start ordering a pizza), *Toppings* in line 11 (to add a topping), *EndOrder* in line 18 (to finish an order) and *ToppingsInfo* in line 23 (to obtain the available toppings).

```
Chatbot “Pizza Store” language: en
intents:
  StartOrder:{
      “I want to order a pizza”,
      (“Big”)[size] “pizza”,
      “One” (“small”)[size] “pizza, please”
    }
    parameters:
      size: entity PizzaSize, required, prompts [“What size of pizza do you want?”];
      address: entity text, required, prompts [“What is your address?”];
  Toppings:{
      “extra” (“cheese”)[toppings],
      “with” (“ham”)[toppings],
      (“bacon”)[toppings]
    }
    parameters:
      toppings: entity Ingredients, required, prompts [“What topping do you want?”];
  EndOrder:{
      “That’s all”,
      “Nothing else”,
      “No more toppings, thank you”
    }
  ToppingsInfo: {
      “What toppings do you have?”,
      “Which are the toppings?”,
      “What toppings can I choose?”
    }
entities:
  Simple entity PizzaSize:{
      small synonyms tiny, little
      medium synonyms regular, intermediate
      big synonyms huge, large
  }
  Simple entity Ingredients:{
      cheese ham pepperoni bacon mushrooms pepper olives corn chicken
  }
actions:
  Button response askingForToppings: {
      text: “What toppings do you want?”
      buttons:
      – value: “cheese”      – value: “ham”
      – value: “pepperoni”   – value: “bacon”
      – value: “mushrooms”   – value: “pepper”
      – value: “olives”      – value: “corn”
      – value: “chicken”     – value: “That’s all”
  }
  Request post orderPizza:
      URL: “https://mypizzaStore.com”;
      data: “size” : [“StartOrder.size”],
            “address” : [“StartOrder.address”];
      dataType: JSON;
  Request post noteTopping:
      URL: “https://mypizzaStore.com”;
      data: “address” : [“StartOrder.address”],
          “toppings”: [“Toppings.toppings”];
      dataType: JSON;
  Text response info:{
      “We have cheese, ham, pepperoni, bacon, mushrooms, pepper, olives, corn, onion and chicken”
  }
flows:
  – user ToppingsInfo => chatbot info;
  – user StartOrder => ask: chatbot askingForToppings{
      => user Toppings => chatbot noteTopping back to ask;
      => user EndOrder => chatbot orderPizza;
  };
```

Listing 4 Defining a chatbot for a pizzeria with Conga.

Intents may also contain parameters, for example, *address* or *size* for intent *StartOrder* (lines from 8 to 10) and *toppings* in intent *Toppings* (line 17). Parameter definitions start with the parameter name, followed by the entity type. This can be a predefined entity like *text* for *address*, or user-defined entities, like *PizzaSize* for *size* and *Ingredients* for the parameter *toppings*. Optionally, we can indicate if the parameter is required, or a list, and prompts that the chatbot asks the user in case of missing values. Training phrases can allude to parameters of the same intent. For example, the sentence "One small pizza, please" in intent *StartOrder* references the parameter *size* with the word small.

The *entities* keyword opens the entity declaration (line 28). In our example, there are two simple entities: *PizzaSize* (line 29) and *Ingredients* (line 34). Similar to intents, entities should provide inputs for every language considered by the chatbot. The *PizzaSize* entity has three entries: small, medium and big (lines 30–32), each of them with two synonyms. The *Ingredients* entity has nine entries with no synonyms.

The *actions* keyword (line 37) declares the possible chatbot actions. Our chatbot has four actions: a *Button response* called *askingForToppings* (line 38), two HTTP *Requests* named *orderPizza* (line 47) and *noteToppings* (line 52) and a text response info (line 57). Button and text responses should also declare inputs for all languages considered by the chatbot, providing an answer text (lines 39 and 58). In addition, button responses should define a list of values, which are the button options (line 41 to 45). Request actions need the URL of the endpoint (lines 48 and 53), may have data (lines 49 and 54) to send in the request, and may specify the type of data being sent (lines 51 and 56).

Finally, the keyword *flows* (line 60) marks the section for defining the conversation flows. The example contains two. The first one (line 61) is a simple flow with one user interaction, the intent *ToppingsInfo*, followed by the chatbot textual answer with the information. Overall, this conversation flow enables a user to obtain information on available pizza toppings. The second flow (from line 62 to 64) starts with the user intention *StartOrder*, then, the chatbot asks for the pizza toppings by means of buttons, each containing a possible ingredient. At this point, the flow bifurcates, so that the user may choose an ingredient (intent *Toppings*), or declare that no more toppings are required (intent *EndOrder*). If a topping is selected the chatbot sends the *noteTopping* request and goes back to ask for more toppings (label *ask*). This mechanism defines a loop of asking and storing the pizza ingredients, which ends when the user triggers the *EndOrder* intent. At that point, the chatbot sends an *orderPizza* request and ends the conversation.

4.4 Generating, Parsing and Validating Chatbots

In this section, we describe three of the chatbot services that Conga supports: generators and parser to/from specific chatbot platforms in Section 4.4.1, and chatbot validation services in Section 4.4.2. Platform recommendation services will be explained in Section 4.5.

4.4.1 Chatbot Generators and Parsers

Conga models are not executable, but they can be compiled into code for a particular development tool. For this purpose, our approach is extensible with chatbot compilers – generating the necessary artefacts from Conga models to specific chatbots tools – and parsers – generating Conga models out of chatbot platform artefacts. This approach makes it possible the migration of chatbots defined in a specific tool to another one, by using Conga as an intermediate representation.

There is a wide variety of tools to build and execute chatbots, as we discussed in Section 2.1.2, each one of them offering different features. To facilitate the implementation and integration of generators and parsers, Conga provides an extension mechanism based on web services, enabling the external addition of generators and parsers.

Figure 4.3 shows the meta-model our extension mechanism relies on. The *Conga* platform contains a list of *User*s, with a *name* and a *password*. Users can define *Service*s for a particular chatbot *Tool* (identified by a *name* and *version*). *Web Service*s are a kind of *Service* that can be used to provide *Generator*s, *Parser*s and *Validator*s. Such services are defined by a *URL*, have a *status* (*ON*, *OFF* or *Error*) and a date of *last access*. Technically, they also may contain a list of key-value *headers* and a *basic authentication*.

Generator web services receive a Conga model and return a *zip* file with the different artefacts required by the tool. Conversely, parser web services receive a zip of the tool files and return a Conga model.

Currently, there are compilers and parsers targeting Dialogflow and Rasa. Table 4.1 shows a mapping between Dialogflow and Rasa files into Conga elements. They are detailed in [23].

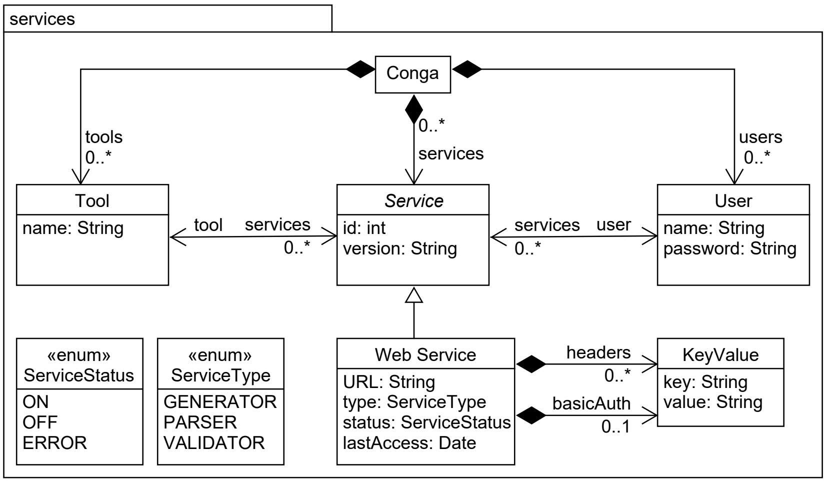


Figure 4.3 CONGA services meta-model.

Table 4.1 Mapping between Dialogflow and Rasa with CONGA.

Dialogflow	Conga	Rasa
Agent.json file contains the configuration information and the language definition	Chatbot	Config.yml file contains the configuration information and the language definition
There is one JSON file per intent with configurations	Intent	Domain.yml contains a list with all the intents
There is a JSON file per language and intent, the language is defined in the name	Language Intent	Only supports one language, the language defined in the config file
Language intent file contains the training phrases, one JSON object per phrase	Training Phrase	nlu.md file contains a list of training phrases per intent in markdown
Intent configuration file contains the list of parameters in a JSON array	Parameter	Parameters can be inferred from training phrases
There is one JSON file per Entity with configuration information, like if it is composite or regular expression	Entity	nlu.md may contain a list of options, synonyms of words and regular expression for parameters
There is a JSON file per language and entity, the language is defined in the name	Language Entity	Only supports one language, the language defined in the config file
The language entity file contains one JSON object per entry	Entry	-
The intent configuration file contains the intent responses like text, images or buttons responses	Text, Image and Button Action	File Domain.yml contains the intent responses like text, images or buttons responses
Agent.json defines the weebhook information, with the URL, headers and basic authentication	HTTP Request Action	The HTTP request could be defined in the action.py file in Python or in the endpoint.yml file
The intent configuration file contains inputs and outputs contexts, which define the conversation flow	Flow	File stories.md defines in markdown conversations flows

Table 4.2 Current validation rules in Conga (BP=Best Practice, EF=Emulated Feature, H=Heuristic, ME=Missing Element, MF=Malformed element, NS=Not Supported Feature, UE=Unused Element).

Id	Validation rule	Severity	Conga element
General			
G1	Names of intents, actions, entities, and parameters should be unique	error	Intent, Action, Entity, Parameter
G2	The language of the element (intent, action, entity) must be among the chatbot languages	error	LanguageInput
G3	There can not be two LanguageInputs with the same language in the same element (intent, action, entity, ...)	error	LanguageInput
G4	Different flow paths cannot start with same intent	error	Flow
G5	There should be a referencing HTTPRequest before the HTTPResponse	error	Flow
G6	Reference “back to” must point to an element on the same path, at a previous position	error	Flow
G7	Parameters in the training phrase must be defined in the same intent	error	Parameter
G8	All entries of the entity should be of the same type	error	Entity
G9	There should be one LanguageInput per chatbot language	warning (ME)	LanguageInput
G10	Regex syntax must be well-formed	warning (MF)	RegularExpression
G11	The loop has no terminating branch	warning (H)	Flow
G12	Defined entities should be used in some parameter	warning (UE)	Entity
G13	Intents should be used in some flow	warning (UE)	Intent
G14	Mandatory parameters should be used in some training phrase	warning (UE)	Parameter
G15	The language intent must contain at least three training phrases	warning (BP)	LanguageIntent
G16	Two training phrases should not be equals in different intent	warning (BP)	TrainingPhrase
G17	Two training phrases should not be equals in the same intent	warning (BP)	TrainingPhrase
G18	If the phrase has a text parameter, it should have more content	warning (BP)	TrainingPhrase
G19	The chatbot should have a fallback intent	warning (BP)	Chatbot
G20	Mandatory parameters should have prompts	warning (BP)	Parameter
Dialogflow			
D1	The data of an HttpRequest will be ignored, Dialogflow sends its JSON format with all the information	warning (NS)	HttpRequest
D2	Dialogflow does not support two HttpRequests in same action, only the first is taken into account	warning (NS)	HttpRequest
Rasa			
R1	Rasa does not support multi-language chatbots, the generator creates one chatbot per language	warning (EF)	Chatbot
R2	Rasa generator does not create a loop, only five repetitions of the conversation	warning (EF)	Flow

4.4.2 Chatbot Validation Services

The Conga language includes model validation rules, which every Conga model should satisfy. In addition, validation services can be added, to check specific well-formedness constraints for particular platforms (cf. Figure 4.3). These validations are typically checked before code is synthesized for the platform, to ensure a proper generation is achieved.

Table 4.2 summarizes the validation rules currently supported by Conga. We distinguish two kinds of rules. The first ones are general model invariants, and are included in the *general* section of the table. These rules ensure well-formedness of chatbot models, and we have further divided them into errors and warnings.

Errors may be caused by equally named elements (e.g., two Actions with the same name), which is checked by G1; problems with the declared languages of the element (e.g., an intent provides phrases in a language not supported by the chatbot), which is checked by G2 and G3; problematic conversation flows (G4–G6); parameters (G7) and entities (G8). Flow problems include issues like having two flows starting with the same intent, which would make the chatbot unable to distinguish which flow should be followed (checked by G4); HTTP responses that are not preceded by an HTTP request (G5); and malformed loops (G6).

Warnings can be classified according to whether they check syntactical issues (missing, malformed or unused elements, expected heuristics), and best practices that ensure a proper chatbot definition. Within the first kind, we can find elements (e.g., an intent) that miss phrases in a language defined by the chatbot (G9); malformed regular expressions (G10); loops that are not terminating (G11); and unused elements like entities (G12); intents (G13); and parameters (G14).

Regarding best practices, the validations perform a static analysis of the chatbot definition to assess whether it adheres to best practices for chatbot design. In particular, G15 checks that each intent that is not fallback is defined by a bare minimum number of 3 phrases (otherwise it results in an imprecise intent); G16 checks that the same training phrase does not belong to two intents (since this would potentially confuse the chatbot); G17 checks that text parameters are not the only parameter of a phrase (since otherwise the parameter would take the whole phrase as its value); G18 checks that the bot has a fallback intent (to inform the user if no other intent is recognized); G19 checks that mandatory parameters have prompt phrases (so that the chatbot can asks the users if no value is given); and G20 checks that the buttons' text are actually phrases accepted by some intent (otherwise the chatbot would not understand the button interaction).

In addition, our approach supports specific validation rules for particular chatbot generators. For Dialogflow, we inform of two features that are not fully supported by the platform. Hence, if *HttpRequest*s are used in the chatbot, D1 reports that the data will be ignored; and D2 notifies about multiple *HttpRequests* in the same action. Rasa does not support multi-language chatbots, but Conga supports them. This way, R1 produces a warning explaining that one chatbot per language will be created. Finally, R2 checks if a chatbot has loops. In that case, it informs that the current Rasa generator emulates loops by unrolling them a maximum of 5 times.

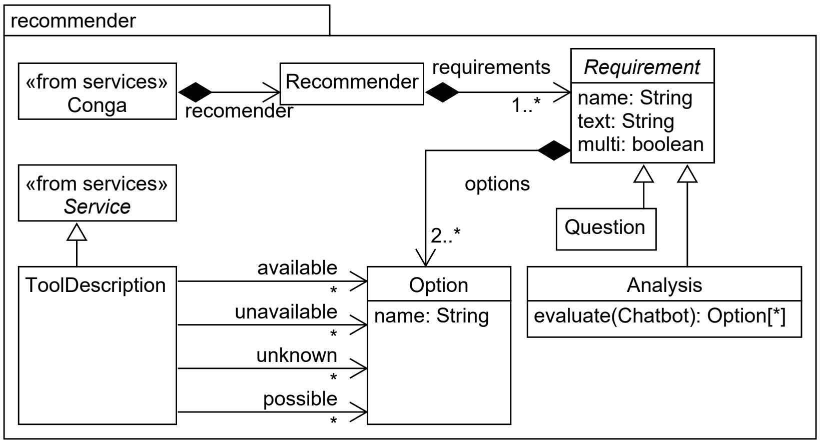


Figure 4.4 Recommender meta-model.

4.5 Recommending a Chatbot Creation Tool

Due to the large amount of tools and approaches for chatbot creation, selecting the best option to build a particular chatbot becomes complex. To assist in this task, we provide a recommender that receives a chatbot model specified with Conga and the answers to a questionnaire relative to other aspects of the chatbot not captured by the model (e.g., technical, organizational or managerial requirements), and from this information, it recommends an appropriate tool to implement the chatbot. The recommender builds on a model-based extensible architecture that enables the addition of new chatbot creation tools and the customization of the questions and model features the recommendation builds on.

Figure 4.4 shows the meta-model our *Recommender* relies on. To make a recommendation, it considers a list of chatbot *Requirements*, whose value can be retrieved either by means of a *Question* to the developer, or automatically via an *Analysis* of the chatbot model. Both kinds of requirements have a *name*, a *text*, a list of admissible *Option*s, and can be *multi-response* or not. In addition, *Analysis* requirements define an *evaluate* method, which analyses the chatbot model and returns a list of *Option*s . This latter class must extend the built-in abstract class *Evaluator* and implement its abstract method *evaluate*, which receives a chatbot model and returns the *Option*s that this model fulfils. The recommendation consists of a list of *Tools* (cf. Figure 4.3). For each tool, the recommender stores the requirement options that are *available*, *unavailable*, *unknown* or are ultimately *possible* (i.e., not natively supported but achievable using a workaround).

The recommender currently follows the criteria described in Section 2.1.2, which is summarized in Table 4.3, but new ones can be added if needed. The table also shows the coverage of these requirements by two chatbot creation tools: Dialogflow and Rasa. Regarding analysis requirements, we check whether the chatbot model is multi-language (like in Listing 4), the targeted languages[1], and whether it uses predefined or chatbot-specific entities, calls to external services, parameters, training phrases or regular expressions. Rasa does not support multi-language bots, but a workaround is generating one bot per language, hence the value *possible* in the table.

Questions are chatbot requirements explicitly asked to the developer as they cannot be inferred from the chatbot model. The first seven questions in Table 4.3 deal with technical aspects. Specifically, we ask for the following issues: the social network the chatbot is to be deployed in (Dialogflow supports 16, and Rasa 8); the hosting server of the chatbot, since some platforms (e.g., Dialogflow) can host the chatbot themselves, but others (e.g., Rasa) require an external server; the level of support for version control, which is built-in in platforms like Dialogflow, while programming-based approaches like Rasa need to use an external version control system like *github*; the need to monitor the chatbot performance (e.g., Dialogflow provides some chatbot analytics); the persistence of utterances for their subsequent analysis; and the need to support speech recognition or sentiment analysis.

The last three questions in Table 4.3 tackle organizational and managerial aspects concerned with open-source and price model requirements, and the level of expertise of the development team. For example, the expertise for using Rasa is higher than for Dialogflow, since the former requires programming.

Since some requirements may be more important than others depending on the project, we assign an importance level to each requirement, which the developer can customize. The supported levels are: *irrelevant*, *relevant*, *double relevant* and *critical*. Irrelevant requirements are not considered for the recommendation, and critical ones are breaking factors (i.e., tools that do not comply with the requirement will not be recommended). For each tool, the recommender computes a score based on the supported requirements and their importance level. *Available* requirements add 1 to the score of a tool, *unavailable* ones add 0, *unknown* ones add 0.5, and *possible* ones add 0.75. In all cases, double relevant requirements score double. Then, the recommender sorts the tools according to their score, and produces a report with the ranking of tools and how each requirement contributes to this ranking.

[1] For brevity, Table 4.3 shows the number of languages supported, not the list of them.

Table 4.3 Requirements that the recommender currently takes into consideration.

Text	Multi-response	Options	Dialogflow	Rasa
Analyses				
Is the chatbot multi-language?	false	Yes	avail.	possib.
		No	avail.	avail.
Which are the chatbot languages?	true	-	21	all
Does the chatbot use new or predefined entities?	true	Predefined	avail.	avail.
		New entities	avail.	avail.
		None	avail.	avail.
Does the chatbot call to external services?	false	One	avail.	avail.
		Multiple	unavail.	avail.
		None	avail.	avail.
Does the chatbot use phrase parameters?	false	Yes	avail.	avail.
		No	avail.	avail.
Does the chatbot need persistent or volatile parameter storage?	true	Persistent	avail.	avail.
		Volatile	avail.	avail.
		None	avail.	avail.
Does your chatbot need natural language processing or pattern matching?	true	NLP	avail.	avail.
		Pattern	avail.	avail.
Has the chatbot conversation loops?	true	Yes	avail.	possib.
		No	avail.	avail.
Questions				
Which social networks do you want to deploy the chatbot in?	true	-	16	8
Do you want to deploy the chatbot on your own host?	false	Tool host	avail.	unavail.
		Own host	unavail.	avail.
Do you want to use a built-in version control system?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require native support for chatbot analytics?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require native support for utterance persistence?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require the chatbot to support speech recognition?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require the chatbot to support sentiment analysis?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require to use an open-source tool?	false	Yes	unavail.	avail.
		No	avail.	avail.
Which price model do you plan to use?	true	Free	avail.	avail.
		Pay as you go	avail.	unavail.
		Quota	unavail.	unavail.
		Pay advanced feats.	unavail.	avail.
What's the level of expertise of the development team?	false	Low	avail.	unavail.
		High	avail.	avail.

Incorporating a new chatbot creation tool (e.g., Watson) into our framework requires: (i) informing the tool options for every requirement in the recommender; (ii) providing a code generator from Conga to the tool; (iii) optionally, providing a parser if reverse engineering is required. The Conga framework prevents the code generation for a tool whenever the chatbot requirements are *unavailable* in that tool. There may be some *possible* requirements though, meaning that their support is not native in the tool but they can be implemented. For instance, Rasa does not support multi-language chatbots, but this can be emulated by generating one chatbot per language. As another example, Dialogflow only supports one external service call per intent, and so, the generator only considers the first call and warns the developer.

4.6 Architecture and Tool Support

The Conga approach has been realized in a web application. It is designed to be extensible with further services like generators, validators, parsers and recommenders for different chatbot creation tools. The application back-end was mainly developed in Java. The Conga DSL was developed using EMF [95] and Xtext [13] and generators using Xtend [13]. The external services (generators, validators and parsers) are defined as REST APIs developed using Jersey framework [42]. PlantUML [80] is used to visualize the conversation flow as a graph. A video showcasing its use and some examples are available at https://saraperezsoler.github.io/CONGA/.

Section 4.6.1 will describe the architecture of Conga and Section 4.6.2 the tool support.

4.6.1 Architecture

Conga is available to chatbot developers as a web application. Figure 4.5 shows its architecture. The front-end includes project, service and recommender managers. The project manager creates and deletes projects, the service manager allows new external service definitions to validate the model, generate code or transform chatbots into a Conga model, and the recommender manager handles the creation of a new tool description for a code generator. It also contains a DSL editor, a graphical renderer of conversation flow models, a questionnaire for the tool recommender, a visualizer of tool recommendations and importers, exporters and validators for some chatbot tools.

The back-end handles the requests of the front-end. Managers store and read the information in a store model (explained below). Chatbot models created in the DSL

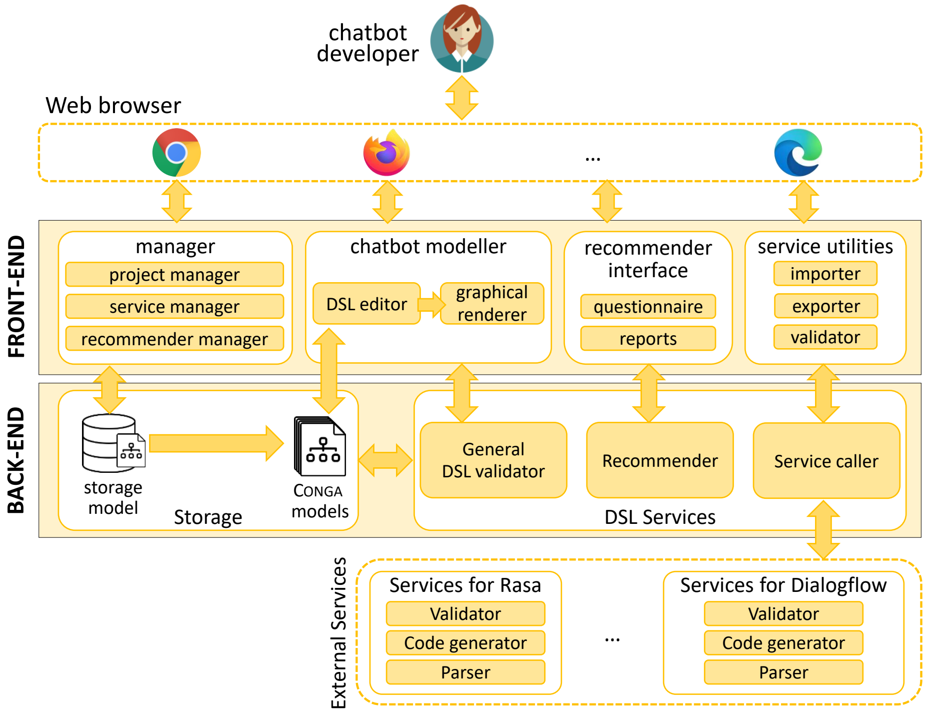


Figure 4.5 CONGA's architecture.

editor are stored in CONGA files. It also contains chatbot model general validation, recommendation computation and a service caller, which handles the external service calls to specific validators, code generators and parsers. Code generators and validators receive the chabot model serialized in an XMI format and return a zip file with the tool-specific files and a JSON with the errors respectively, and parsers receive a zip file with all the files of the chatbot and return the chatbot model in XMI format.

The storage model of CONGA conforms to the meta-model of Figure 4.6. The *Chatbot* definitions are stored in *Project*s. Each project has a *User owner*. Users may own several projects. Projects may also contain the developer's *RecomAnswer*s, made of a list of *Question Answer*s. Question Answer has a reference to the *question*, the *selected options* for that question and the *relevance level* assigned to the question (*irrelevant*, *relevant*, *double relevant*, *critical* or *critical all* (to mark as critical each of the selected options in multi-response questions). *RecomAnswer* also stores the calculated *ranking* list of the recommender with the *Tool Description* and a *score*.

4.6.2 Tool Support

Figure 4.7 shows the main interface of CONGA. The header (label 1) includes a link to the CONGA documentation (hosted in Github), the menus of the project and service

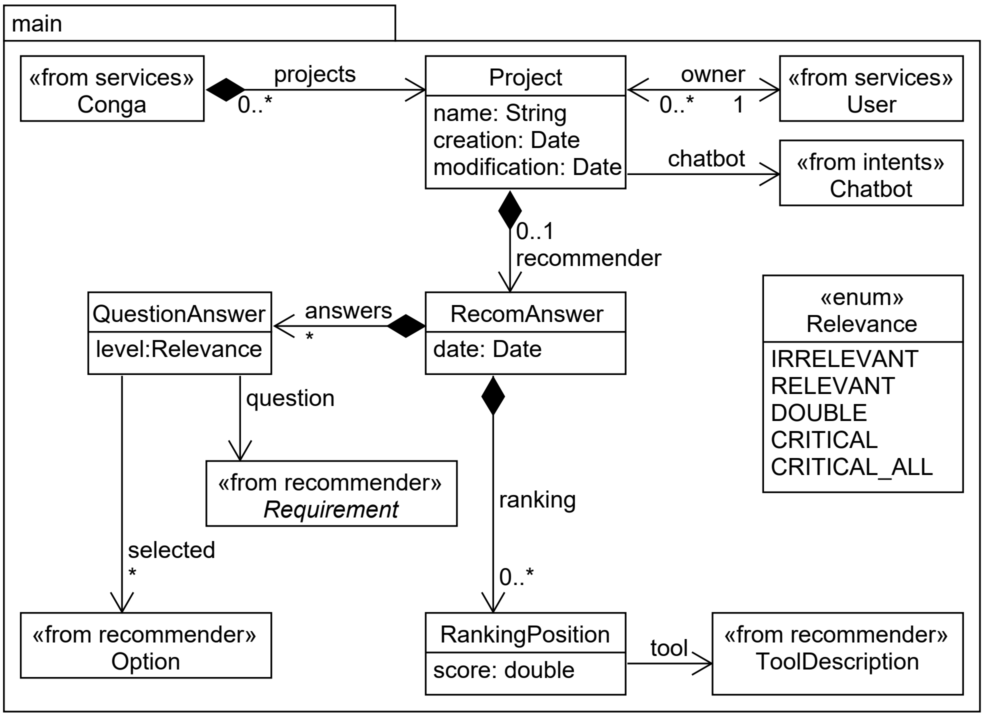
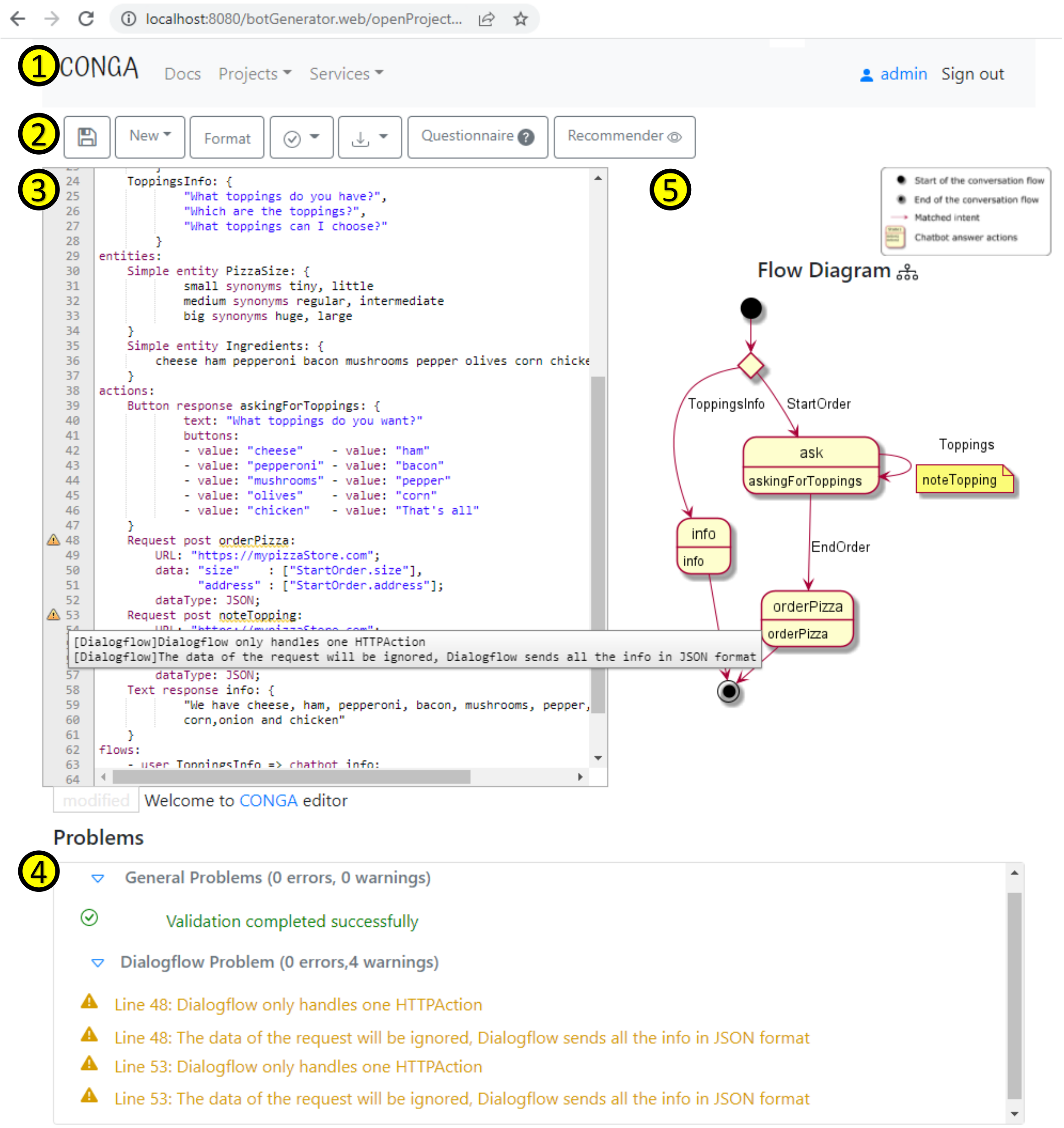


Figure 4.6 Conga storage meta-model.

managers, the name of the logged user, and a sign out button. The button panel (label 2) contains buttons to save the file with the chatbot model, creating a new project, formatting the displayed file, validate the chatbot with a specific validator (Dialogflow or Rasa at the moment), selecting a development tool to generate code for (Dialogflow or Rasa at the moment), filling in the tool recommender questionnaire, and displaying the recommendation results. New projects can be created empty, or be populated from a specific model parsed from an existing chatbot implementation (currently from Dialogflow and Rasa).

The DSL editor (label 3) contains the previous example of the chatbot to order a pizza. The editor is a text area with line numbers. It has syntax highlighting, content assistance and error reporting. In the figure, the editor shows several warnings. Error messages contain their source within brackets. The messages displayed in the figure come from the Dialogflow validator, and warn about the fact that Dialogflow does not allow more than one HttpAction (Validation rule D1 of Table 4.2) and its data will be ignored by the tool (Validation rule D2 of Table 4.2). Technically, the editor is implemented in Xtext, using its web deployment options for the CodeMirror JavaScript library [105].



```
24      ToppingsInfo: {
25              "What toppings do you have?",
26              "Which are the toppings?",
27              "What toppings can I choose?"
28          }
29  entities:
30      Simple entity PizzaSize: {
31              small synonyms tiny, little
32              medium synonyms regular, intermediate
33              big synonyms huge, large
34      }
35      Simple entity Ingredients: {
36          cheese ham pepperoni bacon mushrooms pepper olives corn chicke
37      }
38  actions:
39      Button response askingForToppings: {
40              text: "What toppings do you want?"
41              buttons:
42              - value: "cheese"    - value: "ham"
43              - value: "pepperoni" - value: "bacon"
44              - value: "mushrooms" - value: "pepper"
45              - value: "olives"    - value: "corn"
46              - value: "chicken"   - value: "That's all"
47      }
48      Request post orderPizza:
49          URL: "https://mypizzaStore.com";
50          data: "size"    : ["StartOrder.size"],
51                "address" : ["StartOrder.address"];
52          dataType: JSON;
53      Request post noteTopping:
54          URL: "https://mypizzaStore.com";
[Dialogflow]Dialogflow only handles one HTTPAction
[Dialogflow]The data of the request will be ignored, Dialogflow sends all the info in JSON format
57          dataType: JSON;
58      Text response info: {
59              "We have cheese, ham, pepperoni, bacon, mushrooms, pepper,
60              corn,onion and chicken"
61      }
62  flows:
63      - user ToppingsInfo => chatbot info:
64
```

modified Welcome to CONGA editor

Problems

▽ General Problems (0 errors, 0 warnings)

✓ Validation completed successfully

▽ Dialogflow Problem (0 errors,4 warnings)

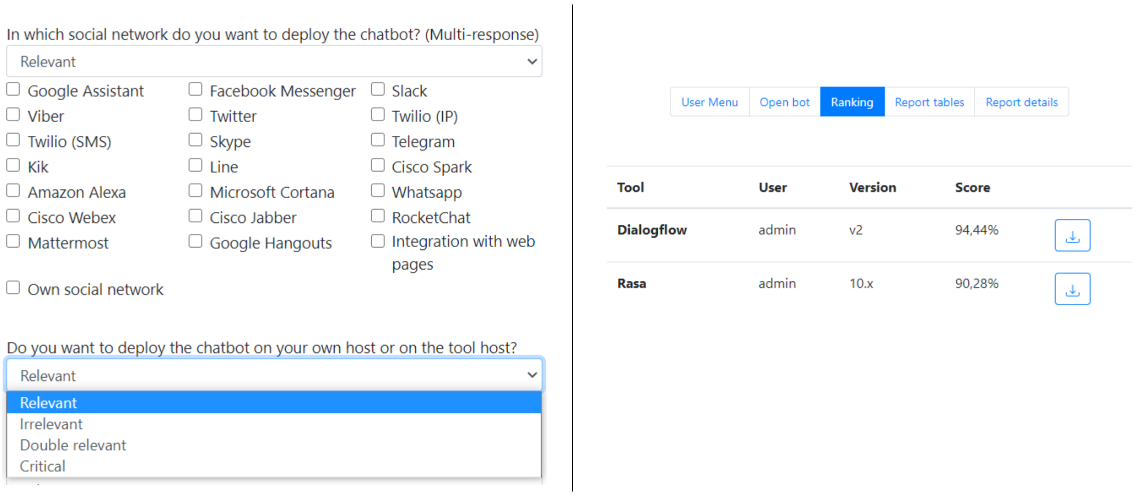
⚠ Line 48: Dialogflow only handles one HTTPAction

⚠ Line 48: The data of the request will be ignored, Dialogflow sends all the info in JSON format

⚠ Line 53: Dialogflow only handles one HTTPAction

⚠ Line 53: The data of the request will be ignored, Dialogflow sends all the info in JSON format

Figure 4.7 CONGA's main interface.



In which social network do you want to deploy the chatbot? (Multi-response)

Relevant

□ Google Assistant
□ Viber
□ Twilio (SMS)
□ Kik
□ Amazon Alexa
□ Cisco Webex
□ Mattermost
□ Facebook Messenger
□ Twitter
□ Skype
□ Line
□ Microsoft Cortana
□ Cisco Jabber
□ Google Hangouts
□ Slack
□ Twilio (IP)
□ Telegram
□ Cisco Spark
□ Whatsapp
□ RocketChat
□ Integration with web pages
□ Own social network

Do you want to deploy the chatbot on your own host or on the tool host?

Relevant

Relevant
Irrelevant
Double relevant
Critical

Tool	User	Version	Score	
Dialogflow	admin	v2	94,44%	⤓
Rasa	admin	10.x	90,28%	⤓

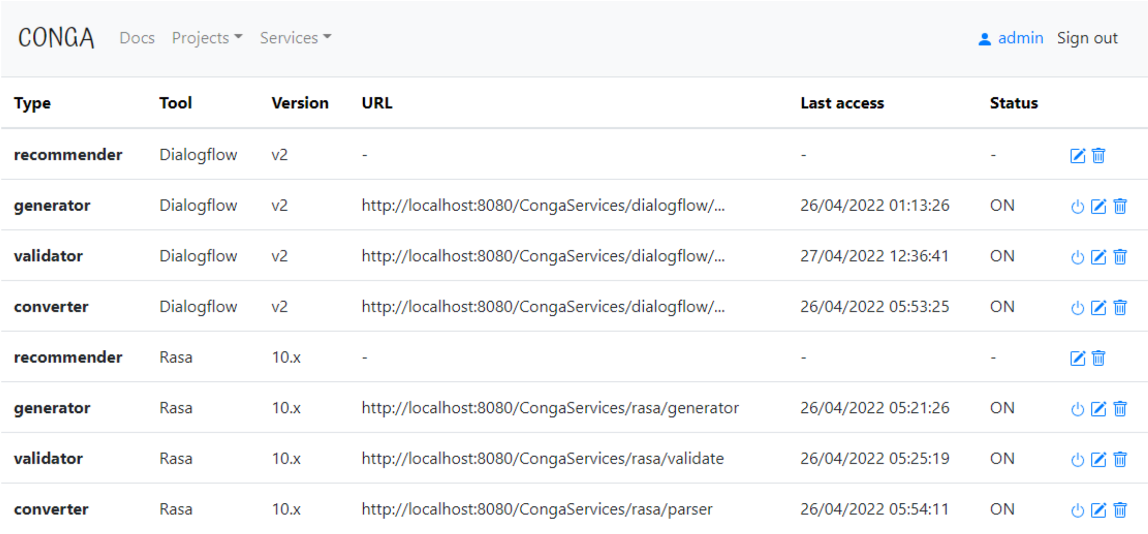
Figure 4.8 Conga recommender support. Requirements questionnaire (left side). Resulting ranking of tools (right side).

Below the editor (label 4) a Problem Console displays the list of errors grouped by their source, with a colour code (green for a successful validation, yellow for warnings and red for errors), the error line and the error message.

The flow diagram to the right (label 5) depicts the conversation flow defined by the chatbot model graphically. The diagram represents the user interactions as transitions and the chatbot interactions as states with the actions that the chatbot performs inside. This view was built atop PlantUML, and is updated whenever the chatbot file is saved.

Figure 4.8 shows the recommender assistant. On the left side, there is an excerpt of the questionnaire that developers must answer to obtain tool recommendations. The questionnaire is created on-the-fly, dynamically according to the modelled requirements (cf. Figure 4.4). Each question has a list of options and a selector of relevance. The right part displays the ranking of tools ordered by decreasing scores. By clicking on the button to the right of a tool, the corresponding code generator is invoked and the developer can download the resulting artefacts.

Figure 4.9 shows the service manager, which contains a list of declared services. It contains the service type (recommender, generator, validator or converted), the tool name, the service version, the location (the recommender does not have location because it is stored and calculated in the Conga tool), the last access to the location, the status (on, off, or error in case the service does not work correctly) and buttons to turn on/off, edit and remove the service.



Type	Tool	Version	URL	Last access	Status
recommender	Dialogflow	v2	-	-	-
generator	Dialogflow	v2	http://localhost:8080/CongaServices/dialogflow/...	26/04/2022 01:13:26	ON
validator	Dialogflow	v2	http://localhost:8080/CongaServices/dialogflow/...	27/04/2022 12:36:41	ON
converter	Dialogflow	v2	http://localhost:8080/CongaServices/dialogflow/...	26/04/2022 05:53:25	ON
recommender	Rasa	10.x	-	-	-
generator	Rasa	10.x	http://localhost:8080/CongaServices/rasa/generator	26/04/2022 05:21:26	ON
validator	Rasa	10.x	http://localhost:8080/CongaServices/rasa/validate	26/04/2022 05:25:19	ON
converter	Rasa	10.x	http://localhost:8080/CongaServices/rasa/parser	26/04/2022 05:54:11	ON

Figure 4.9 CONGA service manager.

4.7 Evaluation

This section reports on two evaluations performed over CONGA. The first one was performed over CONGA's migrations capabilities (Sections 4.7.1). The details can be found in [76] and [77]. The second one was performed over CONGA validation capabilities (Section 4.7.2).

4.7.1 Evaluation of Migration Capabilities

This section reports on an evaluation of CONGA on a migration scenario that involves both backward and forward engineering. The goal is to answer two research questions (RQs): **RQ1**: *Is CONGA expressive enough to capture the details of existing chatbots?* **RQ2**: *Can the migration process be fully automated?* For this purpose, we have migrated eight Dialogflow agents developed by third parties (seven from Github, one built by Google) into Rasa. Table 4.4 summarizes the experiment results.

Game[2] is a conversational agent for a numeric guessing game. Its Dialogflow specification is made of 30 JSON files, has one *http* request, and supports English and French. From this specification, the parser creates a model with 541 objects and 268 lines of CONGA code. Since Rasa does not support multi-language chatbots, two Rasa chatbots are generated from the CONGA model, one for each language. These have 378

[2] https://github.com/actions-on-google/dialogflow-number-genie-nodejs

Table 4.4 Assessment metrics.

	Dialogflow			**Conga**		**Rasa**			
	Back-end	#Files	Lang.	#Obj.	LOC	#Chatbots	Python LOC	Markd. LOC	YAML LOC
Game	yes	30	en/fr	541	268	2	378	242	362
Room reservation	no	17	en	717	196	1	253	166	137
Coffee shop	no	60	en	931	393	1	657	394	269
Nutrition	no	23	en	833	610	1	802	81	99
Bike Shop	yes	13	en	104	80	1	185	61	187
Mystery Animal	yes	199	en	7915	7042	1	9494	13722	879
Smalltalk	no	58	en	2776	1515	1	284	1421	281
IoT: Turn lights	yes	6	en	66	53	1	125	23	168

lines of Python code (to define parameters and actions), 242 lines of Markdown code (to define intents and flows) and 362 lines of YAML code (to configure the chatbot).

Room reservation[3] is a chatbot to book hotel rooms. It has 17 files, and works in English. The migration produces a Rasa chatbot with 253 lines of Python code.

Coffee shop is a Dialogflow pre-built agent to order food to a coffee shop. Its specification has 60 JSON files. These are parsed into a Conga model with 931 objects.

Nutrition[4] is a chatbot to query the nutritional value of meals. Although it is a small chatbot with 23 files, it generates many lines of Python code because the entities have many entries

Bike Shop[5] is an agent to schedule appointments for a shop. It has 13 Dialogflow files and generates 185 lines of Python code, 61 lines of Markdown code and 187 lines of YAML code.

Mystery Animal[6] is a guessing game via QA. It is the largest chatbot of the set with 199 Dialogflow files and generates 13722 lines of Markdown code.

Smalltalk[7] is a chitchatting agent with 58 files and generates 284 lines of Python code, 1421 lines of Markdown code and 281 lines of YAML code.

IoT[8] turns the lights on/off via NL. It is the smallest chatbot of the set with 6 Dialogflow files.

Conga was able to automatically migrate all chatbot logic from Dialogflow to its model, which confirms the expressiveness of Conga (RQ1). However, an aspect required manual intervention. Some Dialogflow agents had back-ends developed using

[3] https://github.com/dialogflow/dialogflow-java-client-v2/tree/master/samples/resources

[4] https://github.com/Viber/apiai-nutrition-sample

[5] https://github.com/dialogflow/fulfillment-bike-shop-nodejs

[6] https://github.com/googlecreativelab/mystery-animal

[7] https://github.com/Janis-ai/Dialogflow

[8] https://github.com/google/voice-iot-maker-demo

Google libraries tightly integrated with Dialogflow. Those cases required configuring the Google services manually and, in one case, implementing a middleware. Generally, the chatbot/back-end connection cannot be migrated fully automatically since it may rely on native technologies of the chatbot platform (e.g., Google's cloud, AWS services). Except for back-end connection, migration was fully automatic (RQ2). These results are very promising, but more case studies are needed to strengthen the confidence in the capabilities of CONGA. Moreover, we manually checked that the produced Rasa chatbots preserved the original Dialogflow behaviour, but we plan to automate this check in future work (e.g., using tools like Botium [14]).

More details of the evaluations can be found in [76] and [77].

4.7.2 Evaluation of Validation Capabilities

This evaluation reports on the CONGA validation capabilities and aims at answering the following research question: *Can CONGA help in the quality assurance of existing chatbots?*.

To perform the evaluation we collected 291 chatbots (140 Dialogflow, 151 Rasa) developed by third parties. We used four different sources: Github to collect open source chatbots, the Dialogflow platform (which includes pre-defined template agents), the Kommunicate website[9] (which includes example agents in Dialogflow), and the Rasa framework website, to obtain Rasa chatbots examples.

To answer this question, we converted the chatbots into CONGA and executed the general validators. Table 4.5 shows a report with the results. The validators found issues in both cases. They are usually warnings, but there are three errors in a Dialogflow chatbot. The chatbot HR-Bot contains three *LanguageInput*s which have languages that are not defined in the chatbot languages. However, the total amount of problems in Rasa is fifty times bigger than in Dialogflow. The highest value in the total of problems in Rasa is the warning G17 (*two training phrases should not be equal in the same intent*), but the highest value in the number of chatbots is G19 (*the chatbot should have a fallback intent*). The G19 warning is one of the lowest values in Dialogflow because, usually, the fallback intent is defined by default, but in Rasa, it should be defined in the configuration file, which is less intuitive. In Dialogflow, the most common issue, in both cases, is G9 (*there should be one LanguageInput per chatbot language*).

[9] https://docs.kommunicate.io/docs/bot-samples

Table 4.5 Report of the general validators

		Dialogflow			Rasa		
	Rule id	#Bots	%Bots	#Total	#Bots	%Bots	#Total
Errors	G1	0	0,00%	0	0	0,0%	0
	G2	1	0,71%	3	0	0,0%	0
	G3	0	0,00%	0	0	0,0%	0
	G4	0	0,00%	0	0	0,0%	0
	G5	0	0,00%	0	0	0,0%	0
	G6	0	0,00%	0	0	0,0%	0
	G7	0	0,00%	0	0	0,0%	0
	G8	0	0,00%	0	0	0,0%	0
Warnings	G9	71	50,71%	756	49	32,5%	260
	G10	0	0,00%	0	5	3,3%	5
	G11	4	2,86%	10	0	0,0%	0
	G12	32	22,86%	58	16	10,6%	34
	G13	8	5,71%	45	84	55,6%	443
	G14	13	9,29%	45	0	0,0%	0
	G15	66	47,14%	250	40	26,5%	103
	G16	8	5,71%	110	17	11,3%	160
	G17	35	25,00%	347	65	43,0%	24414
	G18	30	21,43%	88	75	49,7%	3206
	G19	7	5,00%	7	122	80,8%	122
	G20	22	15,71%	165	0	0,0%	0
	TOTAL	-	-	1884	-	-	28747

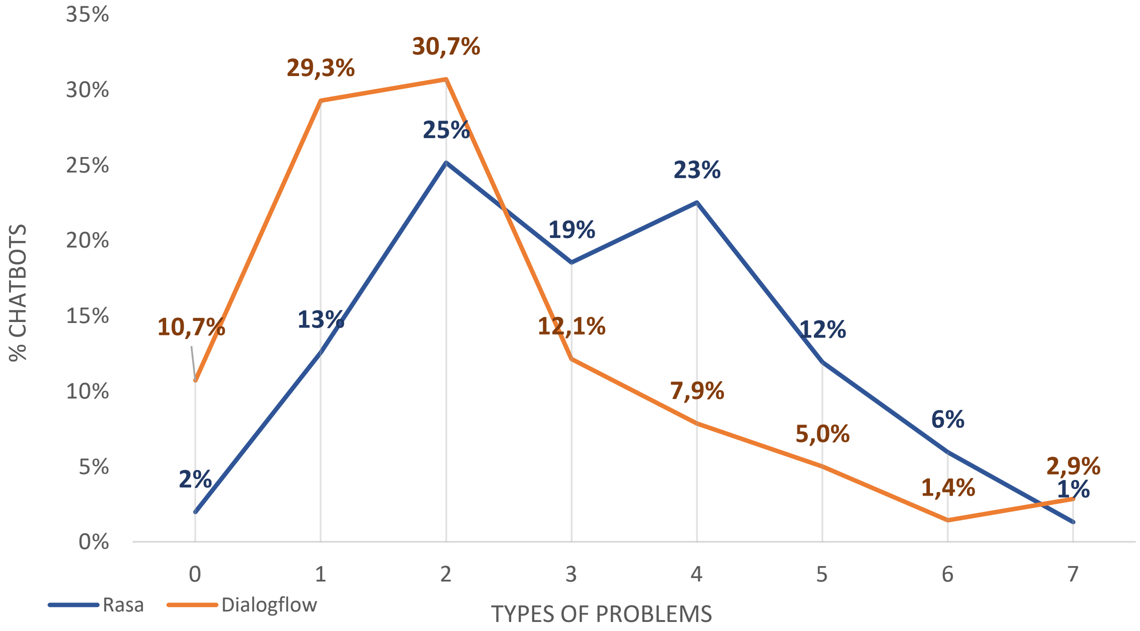


Figure 4.10 Percentage of chatbots per amount of problem types.

Figure 4.10 shows a graphic with the percentage of chatbots per amount of problem type. Only 10,7% in Dialogflow and 2% in Rasa of the chatbots lack problems (15 and 3 respectively). In both cases, the maximum number of problem types in a chatbot is 7 (2,9% of the chatbots in Dialogflow and 1% of the chatbots in Rasa). Most of the chatbots have between 1 and 2 problem types in Dialogflow and between 2 and 4 in Rasa.

Figure 4.11 displays the relation between the percentage of chatbots and the total amount of problems. As we can see, in both cases, most of the chatbots have between 0 and 9 errors and the number of chatbots with more problems decreases. Finally, the maximum amount of problems in a Dialogflow chatbot is 464, and in Rasa 11628.

As we see, the Conga validator is able to find, both, errors and warning in real chatbots.

4.8 Using Conga for Measuring Chatbot Designs

Conga can be used for measuring and comparing heterogeneous chatbots. In this respect, Asymob [22, 50] is a web platform that enables the measurement of chatbots using a suite of 20 metrics. The metrics are defined on Conga as a neutral chatbot design language, becoming independent of the implementation platform. Using Conga

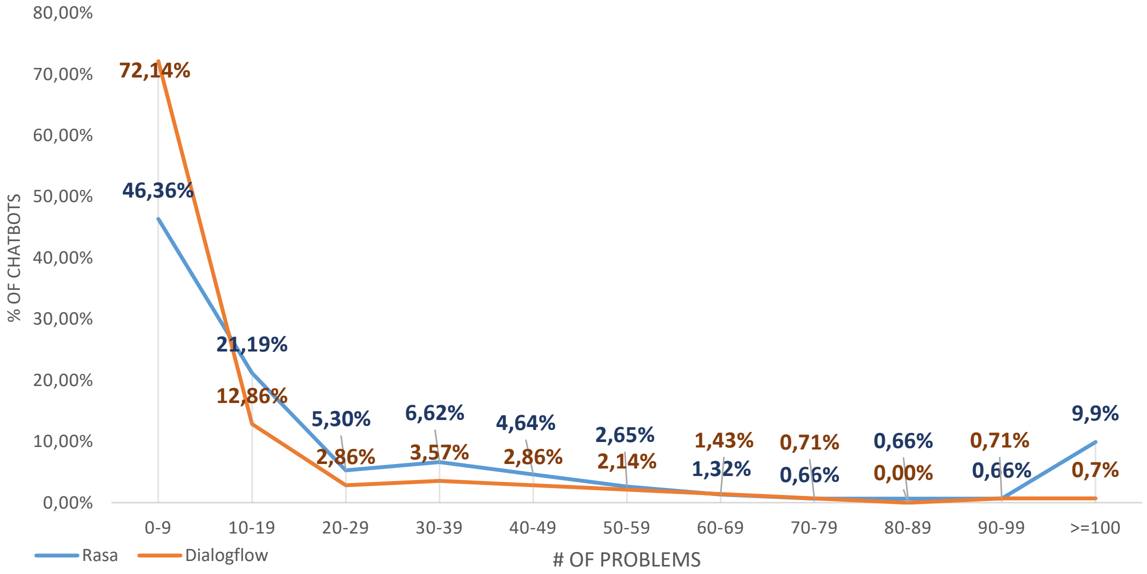


Figure 4.11 Percentage of chatbots per amount of problems.

parsers, Asymob supports the translation of chatbots defined in several platforms into a Conga model to perform the measurements. Asymob's metrics help in detecting quality issues and serve to compare chatbots across and within technologies. The tool also helps in classifying chatbots along conversation topics or design features by means of two clustering methods: based on the chatbot metrics or on the phrases expected and produced by the chatbot. A video showcasing the tool is available at https://www.youtube.com/watch?v=8lpETkILpv8.

4.9 Related Work

The popularity of chatbots has triggered the proposal of many tools for their construction. Section 2.1.2 has a classification of features of these tools, and this section revises works that simplify the forward engineering of chatbots, their reverse engineering and migration, or provide suggestions or guidelines for chatbot development.

Automated development of chatbots. Given the effort and expertise required to build chatbots, some authors have proposed automating their development. The most typical approach consists in automatically generating chatbots from well-defined sources, domains or tasks. This approach is used in [21] to generate chatbots for data exploration starting from data models; in [101] to generate chatbots from OpenAPI specifications to help identifying the right API; in [32] to produce Xatkit chatbots out of open data APIs; in [8] to create Dialogflow chatbots for video game development; in previous sections (Section 3.3.2) to synthesize Dialogflow chatbots from meta-models to support conversational queries; and in Section 3.3.1 to generate Dialogflow chatbots to instantiate meta-models using NL syntax. All these works take as input structured artefacts and apply predefined patterns to generate chatbots for a particular technology and purpose (e.g., data query or data exploration). Instead, Conga enables the generation of chatbots for different technologies, and chatbots may target arbitrary domains, tasks and conversation flows. Actually, Conga could serve as a good target for chatbot generation approaches, since then chatbots for specific platforms could be synthesized.

An alternative way to automate chatbot development is to reuse fragments of existing chatbots. The idea of creating personal chatbots by applying a mashup-based approach was initially proposed in [29]. In [44], the authors discuss how automation, pre-configuration, and templates can aid newcomers to develop FAQ chatbots. Similarly, a template-based chatbot construction approach is proposed in [7], where the templates are common SPARQL queries for Enterprise Knowledge Graphs. As in the case of chatbot generation, all these approaches are not general-purpose, but they produce chatbots for a specific task. Moreover, Conga enables complementary ways of reuse: reusing the knowledge about which chatbot technology to target, and reusing the same design to implement a chatbot for several technologies.

More similar to our proposal, Xatkit [30] is a model-driven solution for developing chatbots that relies on a textual DSL. However, differently from us, Xatkit has its own bot execution engine that builds on Dialogflow to identify the user intent using NLP, and does not generate code for existing chatbot development tools. Moreover, even though Xatkit is model-based, it does not address the recommendation of suitable chatbot platforms, nor reduces the risk of vendor lock-in by supporting chatbot migration.

Baudat et al. [11], facilitate the construction of chatbots for the Watson platform via an OCaml library. The approach synthesizes JSON configuration files, and uses ReactiveML to orchestrate the conversation. This approach is generative, it is limited to Watson and does not support reverse engineering.

Reverse engineering and migration of chatbots. Even if many chatbot development platforms use similar concepts, their realisation varies between platforms [78]. This is problematic as most of the existing solutions are closed, and their use implies vendor lock-in since they typically lack migration capabilities. Given the plethora of available tools and the early stage of many of them, we expect that migration across technologies will be a common need; however, there is currently hardly any support for migration. Just a few solutions like Rasa and early versions of Xatkit are open source. Nonetheless, both demand high technical expertise because they are frameworks where chatbots are developed using general-purpose programming languages (Python in Rasa, and Java in Xatkit). Moreover, only Rasa provides support for migration from Dialogflow, though this support is limited and requires manual data conversion and manual intervention in most migration steps.

Regarding academic works, in [9], the authors envision a reverse engineering process called *botification* to produce a conversational interface for existing web sites. In a similar vein, Chittò et al. [27] translate websites into chatbots based on HTML annotations and a mediator service. The latter chatbots can be used to improve web browsing accessibility [79]. Instead, our system supports migration between chatbot platforms.

Matic et al. [54] propose a microservice architecture for chatbots, which is independent on the NLU component used. This is achieved by the use of a neutral meta-model with NLU concepts (like intents and entities), as well as platform-specific meta-models for Dialogflow and Rasa. This architecture permits selecting the desired NLU component for the chatbot. In our case, Conga includes not only concepts of NLU but also of the bot conversation flow, bot actions, and considers recommendations to specific platforms. Moreover, Conga does not rely on a chatbot execution engine, but on code generators to synthesize code for the target platform.

Overall, there are proposals to botify non-conversational interfaces, but – to the best of our knowledge – not to reengineer existing chatbots. This also implies that there are no general proposals for chatbot migration.

Recommender systems and guidelines for chatbot development. The popularity of chatbots has raised concerns on proper conversational design. For example, IBM's Natural Conversation Framework [61] proposes conversation patterns [62] and conversation design principles [60, 88]. The latter include guidelines like *recipient design* (i.e., allow multiple conversation paths for different user types), *minimization* (i.e., use concise chatbot answers), and *repair* (i.e., provide support for clarifications). In this line, *Chatbottest* [24] defines guidelines for identifying chatbot design issues in categories like answering, error management, intelligence, navigation, personality and understanding. However, the burden is on the developer to manually test whether the chatbot fulfils the guidelines.

In [1], the authors discuss the challenges in chatbot development by analysing posts in StackOverflow. Most posts are about chatbot development, integration, and Natural Language Understanding (NLU). Developers consider the posts of building and integrating chatbots topics more helpful compared to other topics. In [2], the authors compare four NLU components (Dialogflow, LUIS, Watson and Rasa) for their use in software engineering chatbots. They conclude providing five recommendations for fine-tuning the NLUs when developing chatbots. Since Conga is a neutral design notation, we have included validations encoding both platform-specific checkings and general best practices (cf. Table 4.2). Actually, our rule G15 is also mentioned in [2] ("*Train NLUs with multiple queries per intent*").

Some researchers have identified important aspects to consider in chatbot design, like functional, integration, analytics and quality assurance [68]. Section 2.1.2 proposed additional technical and managerial factors which could be used for selecting chatbot development tools. Conga uses these latter factors as a basis for their tool recommendation.

Overall, our contribution in this area is two-fold. On the one hand, extensible validators able to detect – at the design level – errors, potential problems and lack of adjustement to best practices. On the other, an extensible recommender that suggests the most suitable target platform given a chatbot design and other factors.

4.10 Summary and Conclusions

This chapter has presented a model-driven solution for chatbot development, reengineering and migration. The approach is funded on a neutral chatbot design language, a textual DSL called Conga, explained in Section 4.3. It includes an extensible mechanism to define validators, parsers and code generators for specific chatbot construction technologies (Section 4.4). The approach does not involve execution, but a recommender suggests the most suitable chatbot tool given a chatbot model and further requirements (Section 4.5). Section 4.6 presented the architecture and tool support for the approach. Section 4.7.1 evaluated Conga's migration capabilities migrating 8 chatbots from Dialogflow to Rasa with encouraging results. Then, Section 4.7.2 evaluated Conga's validator, also with positive results.

Chapter 5

Conclusions and Future work

This thesis explored the use of chatbots to help in modelling tasks (Chapter 3) and the use of MDE techniques to help in the chatbot definition and selection of the most suitable chatbot creation tool (Chapter 4).

Chapter 3 studied the uses of chatbots as front-ends for modelling services. On one hand, it suggested use a modelling chatbot to create conceptual models (meta-models) in social networks (Section 3.2). The chatbot interprets messages in NL and creates the model accordingly. Using NL as modelling interface has the advantage of lowering the entry barrier to modelling, and any element included in the model is immediately justified by the NL message used for its creation. Moreover, as the chatbot is integrated within social networks, which are widely used in our daily lives, users do not need to install new applications or learn new interfaces. To enrich the traceability and rationale of modelling decisions, the chatbot produces a history model tracking user contributions to model elements. To ease decision-making by a potentially large heterogeneous group, it incorporates a soft-consensus mechanism to measure the degree of agreement based on the group preferences, and avoid the bias that a human moderator may introduce. This approach was prototyped in a tool called Socio. Socio's modelling skill and collaborative mechanisms were evaluated with a preliminary user study with encouraging results (Section 3.4.1). Then, a complete user study was performed with 54 participants to evaluate the usability of Socio with positive results (Section 3.4.2).

On the other hand, complementing the previous approach, Section 3.3 introduces two approaches to create (Section 3.3.1) and query (Section 3.3.2) domain-specific models in social networks using NL. These approaches are based on annotating domain meta-models with configuration information for the NL syntax, and translating these data into a chatbot for creation or query models. The chatbots can be deployed on platforms like Telegram.The feasibility of this solution has been demonstrated by means of a case study where targeting the creation of a modelling chatbot atop an existing cloud system to define and run streaming data applications.

As future work, we plan to improve the NL processing of Socio adding more rules, for example to modify the cardinality of an existing feature. We will incorporate customizable collaboration protocols for different styles of decision making, e.g., based on votings, or roles. We will integrate 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. We plan to perform a usability study with users for the last two approaches, as well as to apply existing quality frameworks for chatbots like [68, 69].

Chapter 4 proposed an MDE approach to chatbot development. It includes a textual chatbot design DSL explained in Section 4.3; extensible mechanisms to define validators, parsers and code generators for specific chatbot construction technologies (Section 4.4) and a platform recommender (Section 4.5). The approach supports both forward and reverse chatbot engineering. Section 4.6 presented the architecture and a tool called Conga. Section 4.7.1 evaluated Conga's migration capabilities migrating 8 chatbots from Dialogflow to Rasa with encouraging results. Then, Section 4.7.2 evaluated Conga's validator, also with positive results.

In the future, we plan to extend Conga framework to support more chatbot creation tools, facilities for model-based testing, and quick-fixes for violations of chatbot best-practices. We plan to perform a user study with developers to assess the advantages of our approach, as well as an evaluation of whether migration fully preserves the chatbot behaviour, automated with Botium [14].

Chapter 6

Conclusiones y Trabajo Futuro

En esta tesis se ha explorado el uso de chatbots para ayudar en las tareas de modelado (Capítulo 3) y el uso de técnicas de MDE para ayudar en el desarrollo de chatbots y en la selección de la mejor herramienta de desarrollo de chatbots (Capítulo 4).

El capítulo 3 ha estudiado el uso de chatbots como interfaz de entrada a servicios de modelado. Por un lado, se ha sugerido el uso de chatbots de modelado para crear modelos conceptuales (meta-modelos) en redes sociales (Sección 3.2). El chatbot interpreta mensajes en NL para inferir la creación del modelo. Usar lenguaje natural como interfaz de modelado tiene la ventaja de reducir la barrera de entrada al modelado, y cualquier elemento incluido en el modelo queda inmediatamente justificado por el mensaje en lenguaje natural usado para su creación. Mas aún, como el chatbot está integrado con redes sociales, que usamos comunmente en nuestro día a día, los usuarios no tienen que instalar nuevas aplicaciones o aprender nuevas interfaces. Para enriquecer la trazabilidad y la lógica tras las decisiones de modelado, el chatbot crea un modelo para el historial que guarda las contribuciones de los usuarios a los elementos del modelo. Para facilitar la toma de decisiones por parte de un grupo heterogéneo potencialmente grande, el chatbot incorpora un mecanismo de consenso para medir el grado de acuerdo basado en las preferencias del grupo y así evitar el sesgo que puede introducir un moderador. Se ha creado un prototipo llamado Socio. Se ha evaluado la capacidad de Socio para asistir en las tareas de modelado, así como el mecanismo de consenso, con unos estudios de usuario preliminares con resultados alentadores (Sección 3.4.1). Posteriormente se ha realizado un estudio completo con 54 participantes para evaluar la usabilidad de Socio, también con resultados positivos (Sección 3.4.2).

Por otro lado, complementariamente al enfoque anterior, la Sección 3.3 introduce dos enfoques para crear (Sección 3.3.1) y consultar (Sección 3.3.2) modelos de dominio en redes sociales usando lenguaje natural. Los enfoques están basados en anotaciones de meta-modelos de dominio con información de configuración para la sintaxis en lenguaje natural, y transformar estos datos en chatbots para la creación o consulta de modelos. Los chatbots se pueden desplegar en plataformas como Telegram. La viabilidad de esta solución se ha demostrado con un caso de estudio donde se ha creado un chatbot de modelado desde un sistema en la nube para definir y ejecutar aplicaciones de trasmisión de datos.

Como trabajo futuro se planea mejorar el procesamiento de lenguaje natural de Socio añadiendo nuevas reglas, por ejemplo para modificar la cardinalidad de propiedades ya existentes. Se incorporarán protocolos de colaboración personalizables para distintos estilos de toma de decisión, por ejemplo, basado en votaciones o roles de usuario. Se integrarán nuevos servicios en nuestros chatbots de modelado, como generadores de código y motores de transformaciones de modelos desplegados en la nube, para facilitar una solución MDE completa a través de lenguaje natural. Se planea realizar un estudio de usabilidad con usuarios para los dos últimos enfoques, además de aplicar marcos de calidad existentes para chatbots como [68, 69].

El Capítulo 4 ha propuesto un enfoque MDE para el desarrollo de chatbots. Incluye un DSL textual para el diseño de chatbots, explicado en la Sección 4.3; un mecanismo extensible para definir validadores, parsers y generadores de código para tecnologías concretas de construcción de chatbots (Sección 4.4); y un recomendador de plataformas (Sección 4.5). El enfoque soporta ingeniería de chatbots directa e inversa. La Sección 4.6 ha presentado una arquitectura y una herramienta de soporte llamada Conga. La Sección 4.7.1 describe una evaluación de la capacidad de Conga para la migración, migrando 8 chatbots desde Dialogflow a Rasa con resultados prometedores. Posteriormente, la Sección 4.7.2 ha evaluado el validador de Conga también con resultados positivos.

En el futuro se planea extender el entorno de Conga con soporte para más herramientas de creación de chatbots, infraestructura para pruebas basadas en modelos, así como quick-fixes para arreglar problemas en el chabot. Se planea realizar un estudio de usuario con desarrolladores para comprobar los beneficios de nuestro enfoque. Además, se pretende llevar a cabo una evaluación que muestre si la migración mediante Conga preserva el comportamiento del chatbot, de forma automatizada usando Botium [14].

References

[1] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. Challenges in chatbot development: A study of stack overflow posts. In *MSR*, pages 174–185. ACM, 2020.

[2] Ahmad Abdellatif, Khaled Badran, Diego Costa, and Emad Shihab. A comparison of natural language understanding platforms for chatbots in software engineering. *IEEE Transactions on Software Engineering*, to appear:1–1, 2021. doi: 10.1109/TSE.2021.3078384.

[3] Aceleo. https://www.eclipse.org/acceleo/. last access in 2022.

[4] AIML. http://www.aiml.foundation/. last access in 2022.

[5] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. Automating the synthesis of recommender systems for modelling languages. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, 2021, pages 22–35, 2021.

[6] Chetan Arora, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proc. MoDELS*, pages 250–260. ACM, 2016.

[7] Caio Viktor S. Avila, Wellington Franco, José Gilvan Rodrigues Maia, and Vânia Maria P. Vidal. CONQUEST: A framework for building template-based IQA chatbots for enterprise knowledge graphs. In *NLDB*, volume 12089 of *LNCS*, pages 60–72. Springer, 2020.

[8] Rubén Baena-Pérez, Iván Ruiz-Rube, Juan Manuel Dodero, and Miguel Angel Bolivar. A framework to create conversational agents for the development of video games by end-users. In *OLA*, volume 1173 of *CCIS*, pages 216–226. Springer, 2020.

[9] Marcos Báez, Florian Daniel, and Fabio Casati. Conversational web interaction: Proposal of a dialog-based natural language interaction paradigm for the web. In *CONVERSATIONS*, volume 11970 of *LNCS*, pages 94–110. Springer, 2019.

[10] Barcelona Open Data. https://opendata-ajuntament.barcelona.cat/. last access in 2022.

[11] Guillaume Baudart, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. Reactive chatbot programming. In *REBLS@SPLASH*, pages 21–30. ACM, 2018.

[12] Ivan Beschastnikh, Mircea F. Lungu, and Yanyan Zhuang. Accelerating software engineering research adoption with analysis bots. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pages 35–38, 2017. doi: 10.1109/ICSE-NIER.2017.17.

[13] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016. ISBN 1786464969.

[14] Botium. https://www.botium.ai/. last access in 2022.

[15] Botkit. https://github.com/howdyai/botkit. last access in 2022.

[16] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *MDSE Principles*, pages 7–24. Springer International Publishing, Cham, 2017. ISBN 978-3-031-02549-5. doi: 10.1007/978-3-031-02549-5_2. URL https://doi.org/10.1007/978-3-031-02549-5_2.

[17] John Brooke. SUS: A retrospective. *J. Usability Studies*, 8(2):29–40, 2013. ISSN 1931-3357.

[18] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. Cognifying model-driven software engineering. In *Proc. STAF Collocated Workshops*, volume 10748 of *LNCS*, pages 154–160. Springer, 2018.

[19] Rubén Campos López. Creación de un asistente de voz para modelado. https://repositorio.uam.es/handle/10486/692816, 07 2020. last access in 2022.

[20] Javier Luis Cánovas Izquierdo and Jordi Cabot. Collaboro: a collaborative (meta) modeling tool. *PeerJ Computer Science*, 2:e84, 2016.

[21] Nicola Castaldo, Florian Daniel, Maristella Matera, and Vittorio Zaccaria. Conversational data exploration. In Maxim Bakaev, Flavius Frasincar, and In-Young Ko, editors, *Web Engineering*, pages 490–497, Cham, 2019. Springer International Publishing. ISBN 978-3-030-19274-7.

[22] Pablo C. Cañizares, José María López-Morales, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Measuring and clustering heterogeneous chatbot designs. *IEEE Transactions on Software Engineering*, pages 1–21, 2022.

[23] Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Automating the measurement of heterogeneous chatbot designs. In *37th ACM Symposium on Applied Computing*, 2022, pages 1491–1498, 2022.

[24] Chatbottest. https://chatbottest.com/. last access in 2022.

[25] Chatfuel. https://chatfuel.com/. last access in 2022.

[26] Chatterbot. https://chatterbot.readthedocs.io/. last access in 2022.

[27] Pietro Chittò, Marcos Báez, Florian Daniel, and Boualem Benatallah. Automatic generation of chatbots for conversational web browsing. In *ER*, volume 12400 of *LNCS*, pages 239–249. Springer, 2020.

[28] Creately. https://creately.com/. last access in 2022.

[29] Florian Daniel, Maristella Matera, Vittorio Zaccaria, and Alessandro Dell'Orto. Toward truly personal chatbots: on the development of custom conversational assistants. In *SE4COG@ICSE*, pages 31–36. ACM, 2018.

[30] Gwendal Daniel, Jordi Cabot, Laurent Deruelle, and Mustapha Derras. Xatkit: A multimodal low-code chatbot development framework. *IEEE Access*, 8:15332–15346, 2020.

[31] Dialogflow. https://dialogflow.com/. last access in 2022.

[32] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Gwendal Daniel, and Jordi Cabot. A model-based chatbot generation approach to converse with open data sources. In *ICWE*, volume 12706 of *LNCS*, pages 440–455. Springer, 2021.

[33] Facebook. https://www.facebook.com/. last access in 2022.

[34] Ismael Fernández Arroyo. Desarrollo de un chatbot para modelado colaborativo en skype. https://repositorio.uam.es/handle/10486/688983, 06 2019. last access in 2022.

[35] FlowXO. https://flowxo.com/. last access in 2022.

[36] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, 2019. ISSN 0098-5589. doi: 10.1109/TSE.2017.2755039.

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

[38] Mario González-Jiménez and Juan de Lara. Datalyzer: Streaming data applications made easy. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering*, pages 420–429, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91662-0.

[39] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA). URL https://aclanthology.org/L18-1550.

[40] Braden Hancock, Antoine Bordes, Pierre-Emmanuel Mazare, and Jason Weston. Learning from dialogue after deployment: Feed yourself, chatbot! In *ACL*, 2019. doi: 10.18653/v1/P19-1358.

[41] Enrique Herrera-Viedma, Francisco Herrera, and Francisco Chiclana. A consensus model for multiperson decision making with different preference structures. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 32(3):394–402, 2002. doi: 10.1109/TSMCA.2002.802821.

[42] Jersey. https://eclipse-ee4j.github.io/jersey/. last access in 2022.

[43] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 286–298, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-47884-3.

[44] Arthur R. T. de Lacerda and Carla S. R. Aguiar. FLOSS FAQ chatbot project reuse: How to allow nonexperts to develop a chatbot. In *OpenSym*. ACM, 2019.

[45] Landbot.io. https://landbot.io/. last access in 2022.

[46] Mathias Landhäußer, Sven J. Körner, and Walter 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. ISSN 0963-9314.

[47] Lex. https://aws.amazon.com/en/lex/. last access in 2022.

[48] Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. Why developers are slacking off: Understanding how software teams use slack. pages 333–336, 02 2016. doi: 10.1145/2818052.2869117.

[49] João Ricardo Esquetim Marques da Costa Lopes, João Miguel Reis Araújo Proença Cambeiro, and Vasco 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.

[50] José María López-Morales, Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Asymob: a platform for measuring and clustering chatbots. In *IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)*, 2022, pages 16–20, 2022.

[51] LUIS. https://www.luis.ai/. last access in 2022.

[52] Madrid Open Data. https://datos.madrid.es. last access in 2022.

[53] Marie-Catherine de Marneffe, Bill Maccartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. LREC*, 2006.

[54] Rade Matic, Milos Kabiljo, Miodrag Zivkovic, and Milan Cabarkapa. Extensible chatbot architecture using metamodels of natural language understanding. *Electronics*, 10(18), 2021. ISSN 2079-9292. doi: 10.3390/electronics10182300. URL https://www.mdpi.com/2079-9292/10/18/2300.

[55] Microsoft Bot Framework. https://dev.botframework.com/. last access in 2022.

[56] Microsoft: QnA Maker. https://www.qnamaker.ai/. last access in 2022.

[57] Microsotf Teams. https://www.microsoft.com/en-us/microsoft-teams/. last access in 2022.

[58] George A. Miller. Wordnet: A lexical database for english. *Comm. ACM*, 38(11): 39–41, 1995.

[59] MOF. https://www.omg.org/mof/. last access in 2022.

[60] Robert J. Moore and Raphael Arar. Conversational UX design: An introduction. In *Studies in Conversational UX Design*, Human-Computer Interaction Series, pages 1–16. Springer, 2018.

[61] Robert J. Moore and Raphael Arar. *Conversational UX Design: A Practitioner's Guide to the Natural Conversation Framework*. ACM, New York, NY, USA, 2019. ISBN 9781450363013.

[62] Robert J. Moore, Eric Liu, Saurabh Mishra, and Guang-Jie Ren. Design systems for conversational UX. In *2nd Conference on Conversational User Interfaces, CUI*, pages 45:1–45:4. ACM, 2020.

[63] Andrés Navarro Blanco. Desarrollo de una aplicación web para modelado colaborativo. https://repositorio.uam.es/handle/10486/688960, 07 2019. last access in 2022.

[64] OCL. https://www.omg.org/spec/OCL/2.3.1/PDF. last access in 2022.

[65] OMG. https://www.omg.org/. last access in 2022.

[66] Pandorabots. https://home.pandorabots.com/. last access in 2022.

[67] Part-of-Speech Tagging Guidelines for the Penn Treebank Project. https://catalog.ldc.upenn.edu/docs/LDC99T42/tagguid1.pdf. last access in 2022.

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

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

[70] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Assisted modelling over social networks with SOCIO. In *Proceedings of MODELS 2017 Satellite Event co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017.*, pages 561–565, 2017. URL http://ceur-ws.org/Vol-2019/demos_1.pdf.

[71] Sara Pérez-Soler, Esther Guerra, Juan de Lara, and Francisco Jurado. The rise of the (modelling) bots: towards assisted modelling via social networks. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 723–728, 2017. doi: 10.1109/ASE.2017.8115683. URL https://doi.org/10.1109/ASE.2017.8115683.

[72] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software*, 35(6): 48–54, November 2018. ISSN 0740-7459. doi: 10.1109/MS.2018.290101511.

[73] Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, and Juan de Lara. Towards conversational syntax for domain-specific languages using chatbots. *Journal of Object Technology*, 18(2):5:1–21, July 2019. ISSN 1660-1769. doi: 10.5381/jot.2019.18.2.a5. URL http://www.jot.fm/contents/issue_2019_02/article5.html.

[74] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Flexible modelling using conversational agents. In *5th Flexible MDE Workshop (satellite event of ACM/IEEE MODELS'19)*, pages 478–482. IEEE Computer Society, 2019.

[75] Sara Pérez-Soler, Gwendal Daniel, Jordi Cabot, Esther Guerra, and Juan de Lara. Towards automating the synthesis of chatbots for conversational model query. In *EMMSAD'2020 (CAISE)*, LNCS, pages 257–265, 2020.

[76] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Model driven chatbot development. In *39th International Conference on Conceptual Modelling (ER'2020)*, Springer, pages 207–222, 2020.

[77] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Creating and migrating chatbots with conga. In *IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)*, 2021, pages 37–40, 2021.

[78] Sara Pérez-Soler, Sandra Juárez-Puerta, Esther Guerra, and Juan de Lara. Choosing a chatbot development tool. *IEEE Software*, 38(4):94–103, 2021. doi: 10.1109/MS.2020.3030198.

[79] Alessandro Pina, Marcos Báez, and Florian Daniel. Bringing cognitive augmentation to web browsing accessibility. In *ICSOC Workshops*, volume 12632 of *LNCS*, pages 395–407. Springer, 2020.

[80] PlantUML. https://plantuml.com/. last access in 2022.

[81] Adam Di Prospero, Nojan Norouzi, Marios Fokaefs, and Litoiu.

[82] Rasa. https://rasa.com/. last access in 2022.

[83] Ranci Ren, John Wilmar Castro, Adrian Santos, Silvia T. Acuña, Pérez-Soler, and Juan de Lara. Collaborative modelling: chatbots or on-line tools? an experimental study. In *EASE'2019*, pages 260–269. ACM, 2020.

[84] Ranci Ren, Sara Pérez-Soler, John W. Castro, Oscar Dieste, and Silvia T. Acuña. Using the socio chatbot for uml modeling: A second family of experiments on usability in academic settings. *IEEE Access*, 10:130542–130562, 2022. doi: 10.1109/ACCESS.2022.3228772.

[85] Davide Di Ruscio, Mirco Franzago, and Ivano Malavolta. Envisioning the future of collaborative model-driven software engineering. *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, 39: 219–221, 2017. doi: https://doi.org/10.1109/ICSE-C.2017.143.

[86] Pablo Rísquez Almodóvar. Desarrollo de un chatbot para modelado colaborativo en slack. https://repositorio.uam.es/handle/10486/688978, 07 2019. last access in 2022.

[87] Rijul Saini, Gunter Mussbacher, Jin L.C. Guo, and Jörg Kienzle. Automated, interactive, and traceable domain modelling empowered by artificial intelligence. *Software and Systems Modeling*, 21:1–31, 06 2022. doi: 10.1007/s10270-021-00942-6.

[88] Emanuel A. Schegloff. *Sequence Organization in Interaction.* Cambridge University Press, 2007.

[89] Renuka Sindhgatta, Alistair Barros, and Alireza Nili. Modeling conversational agents for service systems. In *On the Move to Meaningful Internet Systems, Proc. OTM*, pages 552–560, 2019.

[90] Skype. https://www.skype.com/. last access in 2022.

[91] Slack. https://slack.com/. last access in 2022.

[92] SmartLoop. https://smartloop.ai/. last access in 2022.

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

[94] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley Sons, Inc., Hoboken, NJ, USA, 2006. ISBN 978-0-470-02570-3.

[95] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[96] Margaret-Anne D. Storey, Alexey Zagalsky, Fernando Marques Figueira Filho, Leif Singer, and Daniel M. Germán. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Trans. Software Eng.*, 43(2):185–204, 2017.

[97] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21:147–186, 2014. doi: https://doi.org/10.1007/s10515-013-0130-2.

[98] Telegram. https://telegram.org/. last access in 2022.

[99] Yuan Tian, Ferdian Thung, Abhishek Sharma, and David Lo. Apibot: Question answering bot for api documentation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 153–158, 2017. doi: 10.1109/ASE.2017.8115628.

[100] Twitter. https://twitter.com/. last access in 2022.

[101] Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. Generating chat bots from web API specifications. In *Proc. ACM SIGPLAN Onward!*, pages 44–57, 2017.

[102] Watson. https://www.ibm.com/cloud/watson-assistant/. last access in 2022.

[103] Whatsapp. https://www.whatsapp.com/. last access in 2022.

[104] Xenioo. https://www.xenioo.com/en/. last access in 2022.

[105] Xtext Web Support. https://www.eclipse.org/Xtext/documentation/330_web_support.html. last access in 2022.

[106] Yammer. https://www.yammer.com/. last access in 2022.

Appendix A

Full Text of Journal Papers

A.1 Collaborative modelling and group decision-making using chatbots within social networks

Title	**Collaborative modelling and group decision-making using chatbots within social networks**	
Journal	IEEE Software	
Impact Factor	2018 2.945	Q1, Software Engineering
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1109/MS.2018.290101511	
Date	November 2018	

Abstract Modelling is used in early phases of software and system development to discuss and explore problems, understand domains, evaluate alternatives and comprehend their implications. In this setting, modelling is inherently collaborative as it involves stakeholders with different backgrounds and expertise, who cooperate to build solutions based on consensus. However, modelling tools typically provide unwieldy diagrammatic editors that may hamper the active involvement of domain experts and lack mechanisms to ease decision-making.

To tackle these issues, we embed modelling within social networks, so that the interface for modelling is natural language which a chatbot interprets to derive an appropriate domain model. Social networks have intuitive built-in discussion mechanisms, while the use of natural language lowers the entry barrier to modelling for domain experts. Moreover, we facilitate the choice among modelling alternatives using soft consensus decision-making. This approach is supported by our tool SOCIO, which works on social networks like Telegram.

EDITOR INFO. Name here. Affiliation here.

SECTION TITLE. Section Title

Collaborative modelling and group decision-making using chatbots within social networks

Sara Pérez-Soler, Esther Guerra, Juan de Lara
Modelling and Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Modelling is used in early phases of software and system development to discuss and explore problems, understand domains, evaluate alternatives and comprehend their implications. In this setting, modelling is inherently collaborative as it involves stakeholders with different backgrounds and expertise, who cooperate to build solutions based on consensus. However, modelling tools typically provide unwieldy diagrammatic editors that may hamper the active involvement of domain experts and lack mechanisms to ease decision-making.

To tackle these issues, we embed modelling within social networks, so that the interface for modelling is natural language which a chatbot interprets to derive an appropriate domain model. Social networks have intuitive built-in discussion mechanisms, while the use of natural language lowers the entry barrier to modelling for domain experts. Moreover, we facilitate the choice among modelling alternatives using* soft consensus *decision-making. This approach is supported by our tool* SOCIO*, which works on social networks like Telegram.

EDITOR INTRO HEAD. For example, "From the Editor"

EDITOR INTRO PARAGRAPH. This is the main text for the editor intro.

Keywords: Collaborative modelling, social networks, chatbots, decision-making

1. Introduction

Many software development activities are not individual but require collaboration among teams of stakeholders. Modelling is no exception to this rule since the initial stages of development generally involve heterogeneous partners with diverse background and likely distributed. Among them, the participation of domain experts is essential for building successful domain models. However, the use of highly technical and unwieldy modelling tools may hinder their engagement.

As any group activity, collaborative modelling needs mechanisms for decision-making and consensus building. This includes support for the proposal, discussion and selection of modelling alternatives. Given that large-scale collaboration is often the norm in software projects, modelling tools should provide handy and scalable means for discussion and collaboration[3].

Nowadays, social networks and messaging systems are pervasive to discuss among peers, keep contact with friends and organize all sorts of activities. Not only general-purpose networks like Twitter[1], Facebook[2], Whatsapp[3] or Telegram[4] have boosted, but specialized work-team nets like Slack[5], Workplace[6] or Yammer[7] have spread in enterprises. The reason of this success is their agility, simplicity of use, and the possibility to use them everywhere and in mobility, covering the need to stay connected while being familiar to people.

The use of social media has also disrupted how software engineers work[9], changing the way developers communicate with colleagues and participate in open communities. Advances in natural language (NL) processing have enabled the proliferation of *chatbots* which run on social networks and offer services to users upon NL requests, thereby mimicking human responses. Developers use bots, e.g., to automate deployment tasks, schedule tasks like sending reminders, integrate communication channels, or customer support[6]. They have also been proposed to access API documentation[11] and analyse software projects[1].

Previously[8], we have used social networks for collaborative modelling to leverage on their ubiquity, extended use, scalability and discussion support. Since interaction within social networks is performed through NL, we presented a chatbot called SOCIO that creates domain models out of requirements expressed in NL. This promotes the participation of non-modelling experts in the modelling task. Here, we extend SOCIO to ease decision-making by combining facilities for creating model branches, with soft consensus mechanisms[5] that assist the team in selecting among alternatives.

2. Collaborative modelling in social networks using chatbots

Collaborative modelling can occur offline or online[2]. The former yields asynchronous interactions where users check out models from a version control system, perform local changes, and commit them back to the server. Online collaboration is synchronous, with users meeting in collaborative sessions likely from remote places. This collaboration mode is more appropriate in our context as it supports early discussions and knowledge building, but it demands advanced tooling in terms of context awareness (i.e., knowing who is doing what), discussion (e.g., chats) and coordination (e.g., collaboration protocols).

Existing tools for online collaborative modelling are either diagramming web applications (e.g., GenMyModel[8], Lucidchart[9], AToMPM[10], the online MONDO collaboration framework[2]) or dedicated modelling tools (e.g., MetaEdit+[10], SPACE-DESIGN[4]). In all cases, building models requires the direct manipulation of diagrams, which is laborious and may hinder the active involvement of domain experts with low technical profile. Although some of them provide discussion channels, they are ad-hoc and must be learned.

Our work radically departs from these approaches and looks at modelling as an add-on to a truly discussion environment. In particular, we rely on the discussion and interaction mechanisms offered by social networks based on micro-blogging – like

[1] https://twitter.com/
[2] https://facebook.com/
[3] https://www.whatsapp.com/
[4] https://telegram.org/
[5] https://slack.com
[6] https://www.facebook.com/workplace/
[7] https://www.yammer.com/
[8] https://www.genmymodel.com/
[9] https://www.lucidchart.com/
[10] http://www.metacase.com/

Twitter or Telegram – as they are familiar to many people and do not require installing or learning new applications. Thereby, a collaborative session occurs within a social network, and may involve any number of stakeholders with both technical and non-technical expertise, who can discuss and coordinate via regular short messages.

To assist in the modelling task, a chatbot called SOCIO can be added as a participant to the session. The bot interprets domain requirements in NL and creates a domain model out of them. NL lowers the entry barrier of modelling to domain experts, who may not necessarily know about modelling and may find modelling tools intimidating. Moreover, text interfaces are lightweight and simple to use compared to diagrammatic editors, where one may need to take care of the model layout.

Interacting with SOCIO depends on the social network; in Telegram, the bot is addressed through commands starting by `/`. For example, `/talk` permits describing a domain requirement to the bot. Upon receiving this command, the bot parses the requirement using the Stanford NL parser[7], and applies a set of extensible NL processing rules that trigger update actions on the model. Then, it sends a picture of the resulting model back to the users, highlighting the modified elements. SOCIO also provides other modelling facilities, like direct model manipulation using NL, model validation, exporters to Ecore/EMF (the backend technology used for diagrams), and statistics of user contributions. More information about SOCIO[8] is available at https://saraperezsoler.github.io/ModellingBot/.

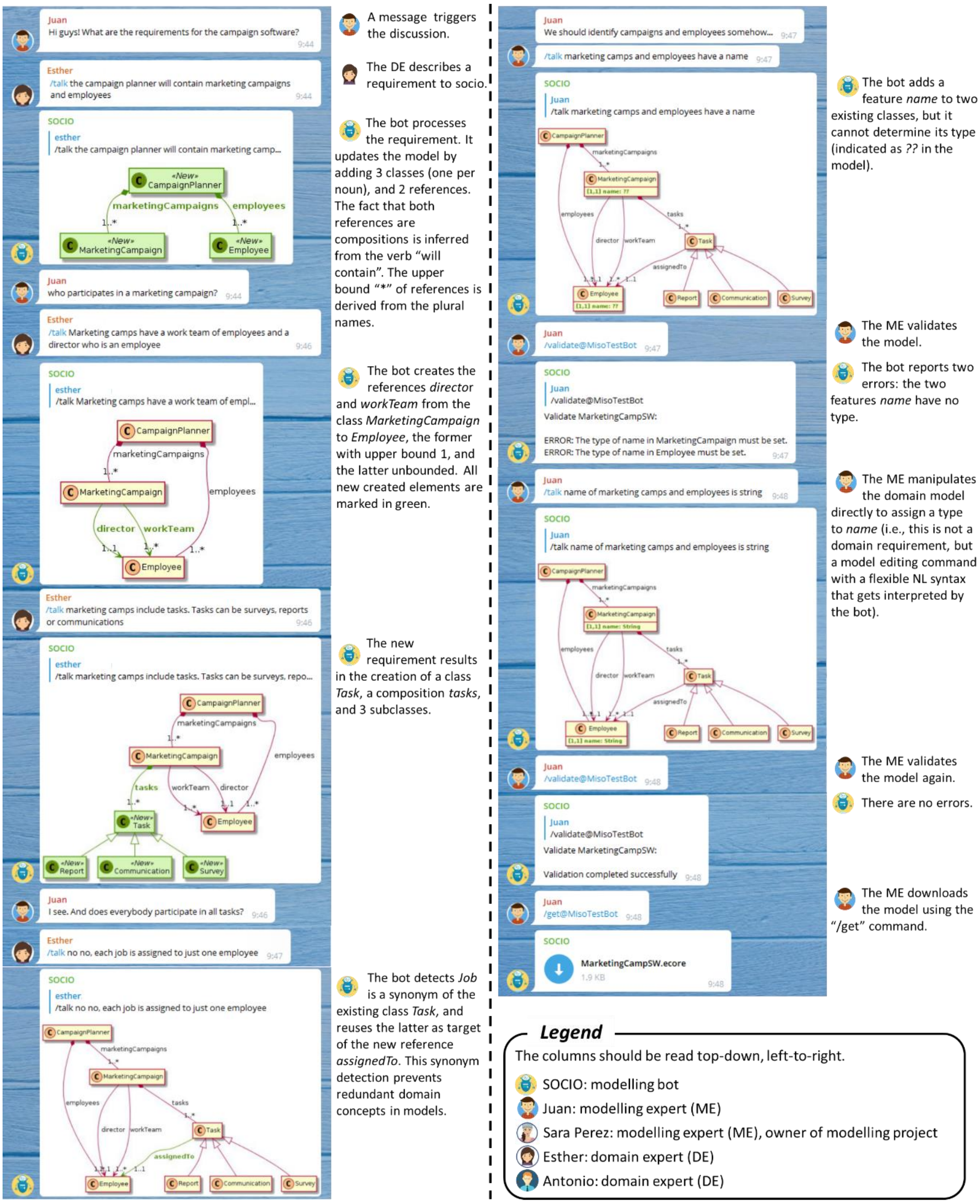
Each domain model is developed within a modelling project. The project creator is its owner and specifies its access policy, which can be *public* (anyone can read and modify), *protected* (anyone can read, but only the owner can modify), or *private* (only the specified users can read and modify). After configuring the project, the modelling session can start.

Figure 1 illustrates a collaborative modelling session in Telegram, where SOCIO assists a modelling expert (ME) and a domain expert (DE) to build a model for marketing campaigns. The DE is the leader of a marketing department and wants an application to manage campaigns and their resources. The session occurs within a Telegram group to which all belong. The figure has two columns to be read top-down, left-to-right. First, the ME sends a message to initiate the discussion, and the DE describes a domain requirement to the bot using the `/talk` command. These messages are received by all group members. As a response to the `/talk` command, the bot first echoes the command, and then adds three classes in the domain model that represent the subject and direct objects of the sentence, and two relationships for the verb. The relationships are compositions because the verb is *to contain* (or a synonym), and are unbounded because the direct objects (e.g., *employees*) are in plural. The figure shows further interactions, which seamlessly mix discussions among human participants and commands addressing the bot. The last interactions show how to validate and download the model.

3. Soft consensus for group decision-making

The participants in a collaborative session may require exploring several solutions to a modelling problem, and eventually, they will have to opt for one of them. If collaboration is distributed or involves many participants, tool assistance to facilitate consensus is essential for agile coordination.

The DEs in our example have expressed the need for a communication channel between the team members of a marketing campaign. The following options are being considered:



Juan
Hi guys! What are the requirements for the campaign software? 9:44

A message triggers the discussion.

Esther
/talk the campaign planner will contain marketing campaigns and employees 9:44

The DE describes a requirement to socio.

SOCIO
esther
/talk the campaign planner will contain marketing camp...

The bot processes the requirement. It updates the model by adding 3 classes (one per noun), and 2 references. The fact that both references are compositions is inferred from the verb "will contain". The upper bound "*" of references is derived from the plural names.

Juan
who participates in a marketing campaign? 9:44

Esther
/talk Marketing camps have a work team of employees and a director who is an employee 9:46

SOCIO
esther
/talk Marketing camps have a work team of empl...

The bot creates the references *director* and *workTeam* from the class *MarketingCampaign* to *Employee*, the former with upper bound 1, and the latter unbounded. All new created elements are marked in green.

Esther
/talk marketing camps include tasks. Tasks can be surveys, reports or communications 9:46

SOCIO
esther
/talk marketing camps include tasks. Tasks can be surveys, repo...

The new requirement results in the creation of a class *Task*, a composition *tasks*, and 3 subclasses.

Juan
I see. And does everybody participate in all tasks? 9:46

Esther
/talk no no, each job is assigned to just one employee 9:47

SOCIO
esther
/talk no no, each job is assigned to just one employee

The bot detects *Job* is a synonym of the existing class *Task*, and reuses the latter as target of the new reference *assignedTo*. This synonym detection prevents redundant domain concepts in models.

Juan
We should identify campaigns and employees somehow... 9:47
/talk marketing camps and employees have a name 9:47

SOCIO
Juan
/talk marketing camps and employees have a name

The bot adds a feature *name* to two existing classes, but it cannot determine its type (indicated as *??* in the model).

Juan
/validate@MisoTestBot 9:47

The ME validates the model.

SOCIO
Juan
/validate@MisoTestBot
Validate MarketingCampSW:

ERROR: The type of name in MarketingCampaign must be set.
ERROR: The type of name in Employee must be set. 9:47

The bot reports two errors: the two features *name* have no type.

Juan
/talk name of marketing camps and employees is string 9:48

The ME manipulates the domain model directly to assign a type to *name* (i.e., this is not a domain requirement, but a model editing command with a flexible NL syntax that gets interpreted by the bot).

SOCIO
Juan
/talk name of marketing camps and employees is string

Juan
/validate@MisoTestBot 9:48

The ME validates the model again.

SOCIO
Juan
/validate@MisoTestBot
Validate MarketingCampSW:

Validation completed successfully 9:48

There are no errors.

Juan
/get@MisoTestBot 9:48

The ME downloads the model using the "/get" command.

SOCIO
MarketingCampSW.ecore
1.9 KB 9:48

Legend

The columns should be read top-down, left-to-right.

SOCIO: modelling bot
Juan: modelling expert (ME)
Sara Perez: modelling expert (ME), owner of modelling project
Esther: domain expert (DE)
Antonio: domain expert (DE)

Figure 1. Collaborative modelling session in Telegram with SOCIO

- A message box per employee, not necessarily working in the same campaign. This would be like an e-mail or peer-to-peer messaging system.
- A special type of work task for discussion, where any employee can post comments and reply to other comments.
- A forum associated to each marketing campaign, where work-team members can contribute news organized into threads, like in bulletin boards.

To study different possibilities, SOCIO supports *branch groups*. These collect the alternatives and discussions to model an aspect of a system as different branches of the current model. In our example, we use this facility to create a group called *communication* with three branches, one for each considered solution: `p2pMessages`, `DiscussionTasks`, and `BulletinBoard`. Anyone with editing access can create a branch group, and the branch creator configures its participants. In each branch, the domain model evolves separately from the other branches, according to the bot-directed messages sent within the branch. SOCIO distinguishes the elements created in a branch from the trunk elements using different colours and the `<Old>` stereotype (see models in Figure 2(a)).

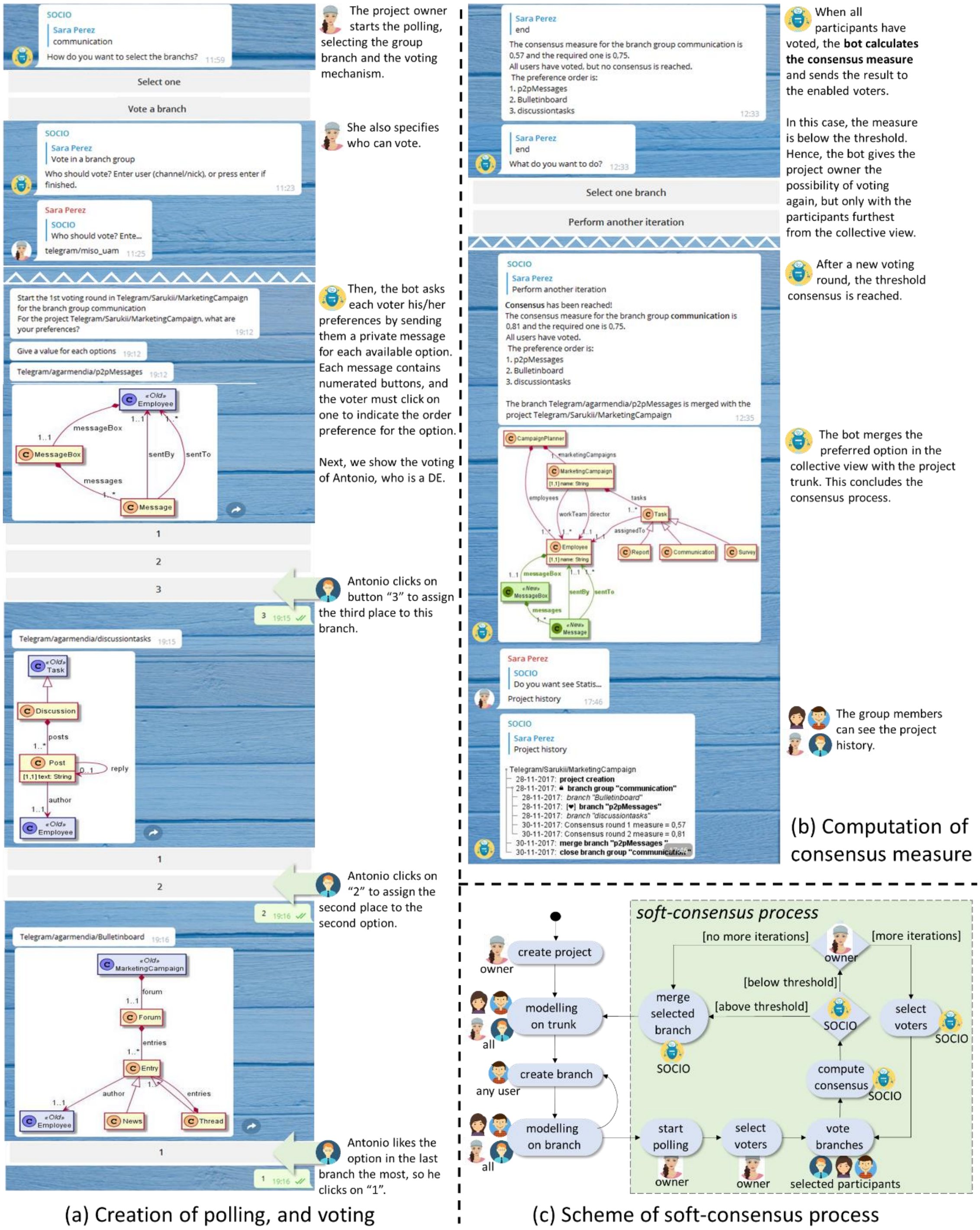
After outlining the alternative solutions in branches, the participants need to agree on the most suitable one. For this, SOCIO incorporates a soft consensus mechanism for multi-person decision-making[5] that assists in choosing the option that is acceptable to all group members. Figure 2(c) shows its working scheme. Participants can express their favourite solution in several ways, like ordering the alternatives from better to worse, or giving a score to each option (e.g., from 1 to 10). Then, an iterative consensus process identifies the preferred alternative based on the expressed preferences. We use "soft" consensus because unanimous agreement can be difficult to achieve, especially in numerous groups or with experts with dissimilar backgrounds. Soft consensus models[5] permit measuring the degree of consensus in a group, provide feedback to each participant on the current consensus, and iterate to improve the consensus and converge towards a shared consensus threshold.

Figure 2(a, b) illustrates the use of SOCIO to select by consensus one solution in the *communication* branch group. Part (a) displays the project owner starting the polling and picking the voters, as well as one voter (Antonio) expressing his preferences through private vote. During the polling, SOCIO will only show the new elements added in each solution and their context, to highlight the variations.

Figure 2(b) shows how consensus is measured. When all voters have indicated their preferences or when a predefined deadline is reached, the system aggregates all answers into a collective preference vector, and computes a global ranking for the alternatives, and a consensus measure ranging from 0 to 1. If the consensus is below a threshold (0.75 by default), another iteration is performed. In the figure, the consensus is 0.57 after the first iteration, which is below the threshold. Hence, voters are ranked according to how far their preference is to the collective preference, and the farthest ones (according to a threshold) are invited to change their choice, while the rest remain unchanged. This process promotes the convergence to an acceptable group consensus[5]. When the consensus reaches the threshold or when the project owner decides so, the most preferred branch is integrated with the main model trunk, and the other branches closed. The branches and voting results can be consulted in the project history.

4. Discussion and outlook

In this paper, we have stressed the collaborative nature of modelling and have argued that this collaboration can take place within social networks and mediated by chatbots which are interfaced by NL and, under the hood, perform modelling tasks. Using NL as modelling interface has the advantage of lowering the entry barrier to modelling, and does not interrupt the group discussion flow because messages for discussion and modelling are intertwined. Moreover, any element included in the model is immediately justified by the NL message used for its creation, hence documenting its provenance. To enrich traceability and rationale of modelling decisions, we also produce a *history* model tracking user contributions to model elements[8].



SOCIO
Sara Perez
communication
How do you want to select the branchs? 11:59

Select one

Vote a branch

The project owner starts the polling, selecting the group branch and the voting mechanism.

SOCIO
Sara Perez
Vote in a branch group
Who should vote? Enter user (channel/nick), or press enter if finished. 11:23

Sara Perez
SOCIO
Who should vote? Ente...
telegram/miso_uam 11:25

She also specifies who can vote.

Start the 1st voting round in Telegram/Sarukii/MarketingCampaign for the branch group communication
For the project Telegram/Sarukii/MarketingCampaign, what are your preferences? 19:12

Give a value for each options 19:12

Telegram/agarmendia/p2pMessages 19:12

1

2

3

Then, the bot asks each voter his/her preferences by sending them a private message for each available option. Each message contains numerated buttons, and the voter must click on one to indicate the order preference for the option.

Next, we show the voting of Antonio, who is a DE.

3 19:15

Antonio clicks on button "3" to assign the third place to this branch.

Telegram/agarmendia/discussiontasks 19:15

1

2

2 19:16

Antonio clicks on "2" to assign the second place to the second option.

Telegram/agarmendia/Bulletinboard 19:16

1

1 19:16

Antonio likes the option in the last branch the most, so he clicks on "1".

(a) Creation of polling, and voting

Sara Perez
end
The consensus measure for the branch group communication is 0,57 and the required one is 0,75.
All users have voted, but no consensus is reached.
The preference order is:
1. p2pMessages
2. Bulletinboard
3. discussiontasks
12:33

Sara Perez
end
What do you want to do? 12:33

Select one branch

Perform another iteration

When all participants have voted, the bot **calculates the consensus measure** and sends the result to the enabled voters.

In this case, the measure is below the threshold. Hence, the bot gives the project owner the possibility of voting again, but only with the participants furthest from the collective view.

SOCIO
Sara Perez
Perform another iteration
Consensus has been reached!
The consensus measure for the branch group **communication** is 0,81 and the required one is 0,75.
All users have voted.
The preference order is:
1. p2pMessages
2. Bulletinboard
3. discussiontasks

The branch Telegram/agarmendia/p2pMessages is merged with the project Telegram/Sarukii/MarketingCampaign 12:35

After a new voting round, the threshold consensus is reached.

The bot merges the preferred option in the collective view with the project trunk. This concludes the consensus process.

Sara Perez
SOCIO
Do you want see Statis...
Project history 17:46

SOCIO
Sara Perez
Project history

```
Telegram/Sarukii/MarketingCampaign
├ 28-11-2017: project creation
├ 28-11-2017: 🔒 branch group "communication"
│ ├ 28-11-2017: branch "Bulletinboard"
│ ├ 28-11-2017: [♥] branch "p2pMessages"
│ ├ 28-11-2017: branch "discussiontasks"
│ ├ 30-11-2017: Consensus round 1 measure = 0,57
│ └ 30-11-2017: Consensus round 2 measure = 0,81
├ 30-11-2017: merge branch "p2pMessages "
└ 30-11-2017: close branch group "communication"
```

The group members can see the project history.

(b) Computation of consensus measure

(c) Scheme of soft-consensus process

Figure 2. Consensus decision-making with SOCIO (a, b). Consensus process scheme (c)

To ease decision-making by a potentially large heterogeneous group, we incorporate a soft-consensus mechanism to measure the degree of agreement based on the group preferences, and avoid the bias that a human moderator may introduce[5]. To assess this hypothesis, we performed a small-scale evaluation with 8 participants recruited from the Master and Doctoral programs of the Department of Computer Science of our university. 6 participants were computer scientists, 1 engineer in telecommunication, and 1 physicist. After attending a 10-minutes tutorial about SOCIO, they used it to select the best solution among three possibilities for two different projects, first without consensus mechanism, and then using it. Interestingly, without the consensus mechanism, they ended up organizing a public polling within Telegram, but discrepancies among participants remained until the end of the experiment. They also answered a five-point Likert scale survey on the consensus mechanism, which was considered especially useful for large groups (average 4.7/5), and with an outcome that reflected the opinion of the majority (4.8/5) and was deemed objective because of the private voting (4.3/5).

The practical usability of modelling chatbots depends on their ability to interpret NL. A preliminary assessment of a previous version of SOCIO revealed that modelling using NL was liked more than using graphical editors (75%), while its NL processing was reasonable but improvable (participants gave an average of 62.5% for accuracy)[8]. To mitigate possible mistakes of the NL processor, SOCIO permits manipulating models directly through NL (i.e., using sentences like "remove employee" which are parsed and interpreted) and undoing actions. While messages in micro-blogging systems are short, which facilitates NL processing, they foster the use of slang and abbreviations, which we will consider in future work. We also plan to extend SOCIO with the ability to answer NL questions about a model, to assist domain experts in subsequent decisions.

Enabling social networks for collaborative modelling brings exciting possibilities, like involving large groups of people (i.e., crowdsourced modelling), or its use in Software Engineering education. We also foresee chatbots for other modelling tasks, like model quality monitoring and refactoring suggestions, and for other diagram types.

Acknowledgements
Work funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

References

1. Beschastnikh, Lungu, Zhuang. Accelerating software engineering research adoption with analysis bots. *ICSE-NIER*, pages 35-38. IEEE, 2017.
2. Debreceni, Bergmann, Búr, Ráth, Varró. The MONDO collaboration framework: secure collaborative modeling over existing version control systems. *ESEC/FSE*, pages 984–988. ACM, 2017.
3. Franzago, Di Ruscio, Malavolta, Muccini. Collaborative model-driven software engineering: A classification framework and a research map. *IEEE Transactions on Software Engineering*, in press, 2017. DOI 10.1109/TSE.2017.2755039.
4. Gallardo, Bravo, Redondo. A model-driven development method for collaborative modeling tools. *J. Network and Computer Applications*, 35(3):1086–1105, 2012.
5. Herrera-Viedma, Herrera, Chiclana. A consensus model for multiperson decision making with different preference structures. *Trans. Sys. Man Cyber*. Part A, 32(3):394–402, 2002.
6. Lin, Zagalsky, Storey, Serebrenik. Why developers are slacking off: Understanding how software teams use slack. *CSCW*, pages 333–336. ACM, 2016.
7. Marneffe, Maccartney, Manning. Generating typed dependency parses from phrase structure parses. *LREC*, pages 449–454. ELRA, 2006.
8. Pérez-Soler, Guerra, de Lara, Jurado. The rise of the (modelling) bots: towards assisted modelling via social networks. *ASE*, pages 723–728. ACM, 2017.
9. Storey, Zagalsky, Figueira Filho, Singer, German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, 2017.
10. Syriani, Vangheluwe, Mannadiar, Hansen, Mierlo, Ergin. AToMPM: A web-based modeling environment. *MODELS Satellite Events*, volume 1115 of CEUR, pages 21–25, 2013.
11. Tian, Thung, Sharma, Lo. APIbot: question answering bot for API documentation. *ASE*, pages 153–158. ACM, 2017.

Sara Pérez-Soler is a Master's student in Computer Science at the Universidad Autónoma de Madrid, and the main developer of SOCIO. Contact her at sara.perezs@estudiante.uam.es.

Dr. Esther Guerra is a professor at the Universidad Autónoma de Madrid. Her research interests include model-driven engineering, automated software development and domain-specific languages. Contact her at esther.guerra@uam.es.

Dr. Juan de Lara is a professor at the Universidad Autónoma de Madrid. His research interests are in model-driven engineering, including meta-modelling, model transformations and domain-specific languages. Contact him at juan.delara@uam.es.

A.2 Towards Conversational Syntax for Domain-Specific Languages using Chatbots

Title	Towards conversational syntax for domain-specific languages using chatbots	
Journal	Journal of Object Technology	
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Mario González-Jiménez	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.5381/jot.2019.18.2.a5	
Date	July 2019	

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 domainspecific 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.

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

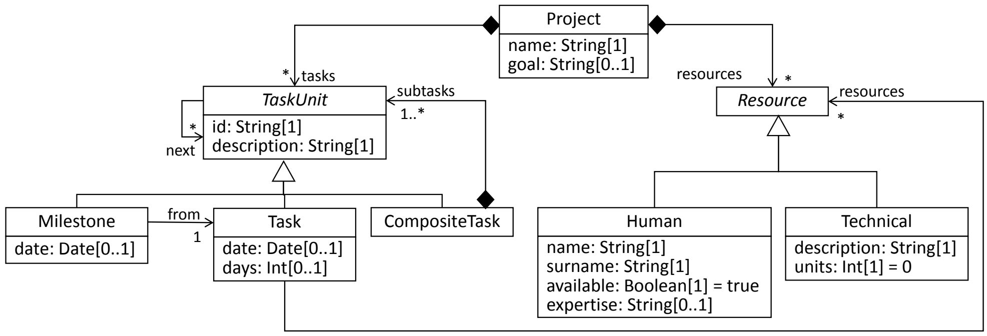


Figure 1 – Running example: A domain meta-model to describe project plans.

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 descri
interfaces, and it offers support f
choice is motivated by its automat
social networks, its flexibility to
deployed on specific clouds, like
possibility to define the chatbot u
cloud development environment)

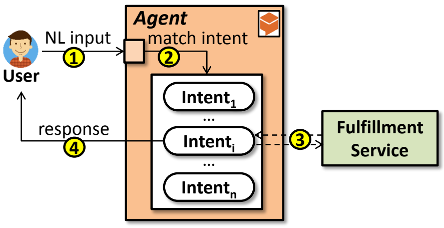


Figure 2 – Agent working scheme.

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 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

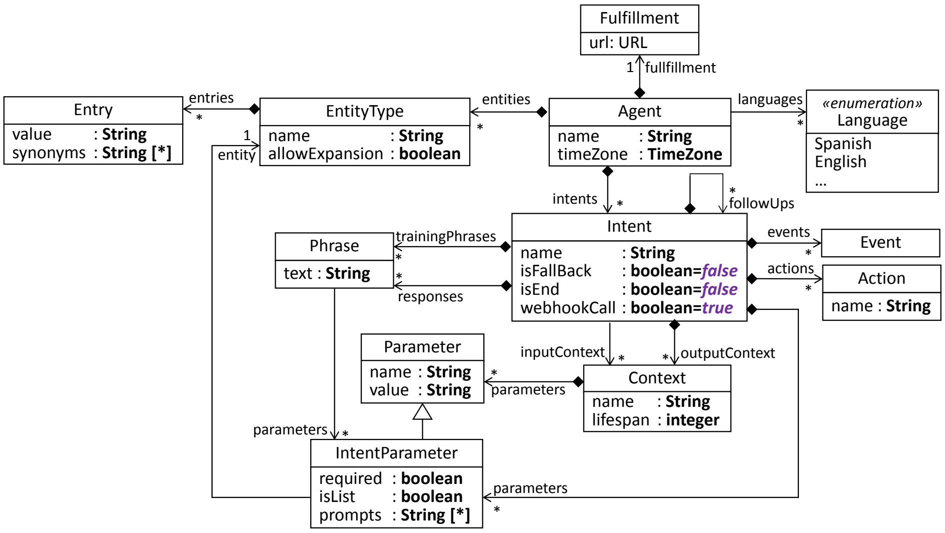


Figure 3 – A meta-model for DialogFlow.

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.

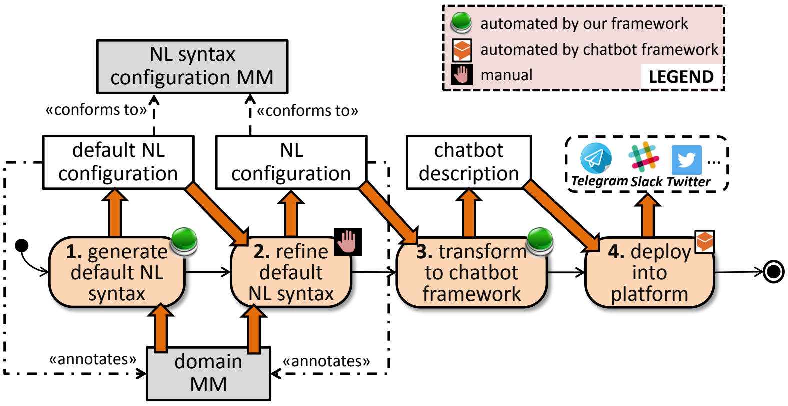

automated by our framework
automated by chatbot framework
manual
LEGEND
NL syntax configuration MM
«conforms to»
«conforms to»
default NL configuration
NL configuration
chatbot description
Telegram Slack Twitter ...
1. generate default NL syntax
2. refine default NL syntax
3. transform to chatbot framework
4. deploy into platform
«annotates»
domain MM
«annotates»

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

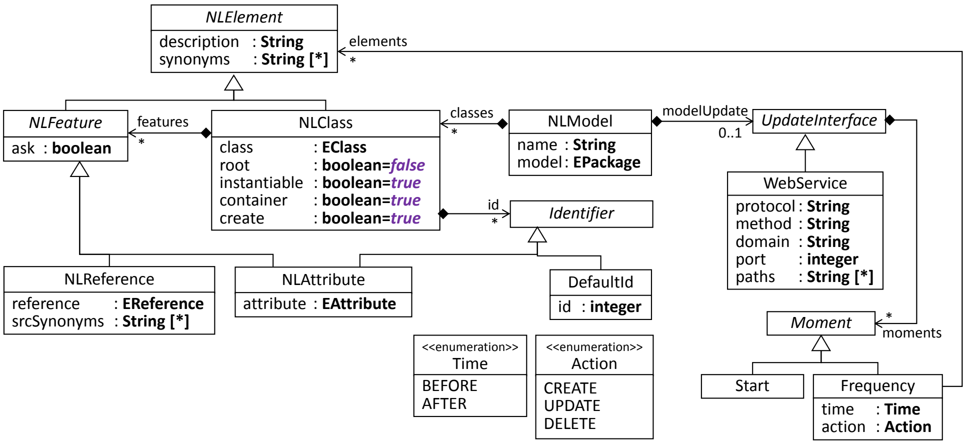


Figure 5 – The meta-model for configuring the NL syntax.

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 `WebService`s. For this purpose, it is necessary to specify the `protocol`, `method`, `domain`, `port` and `paths` of the web service; and to configure the `Moment`s 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

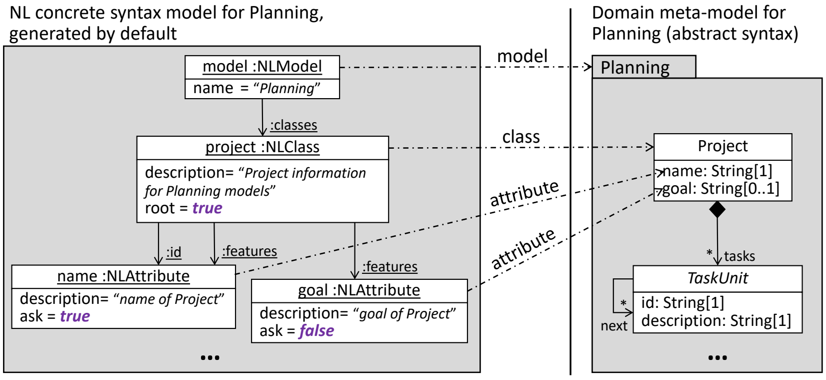


Figure 6 – Excerpt of default NL concrete syntax model for the running example.

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.

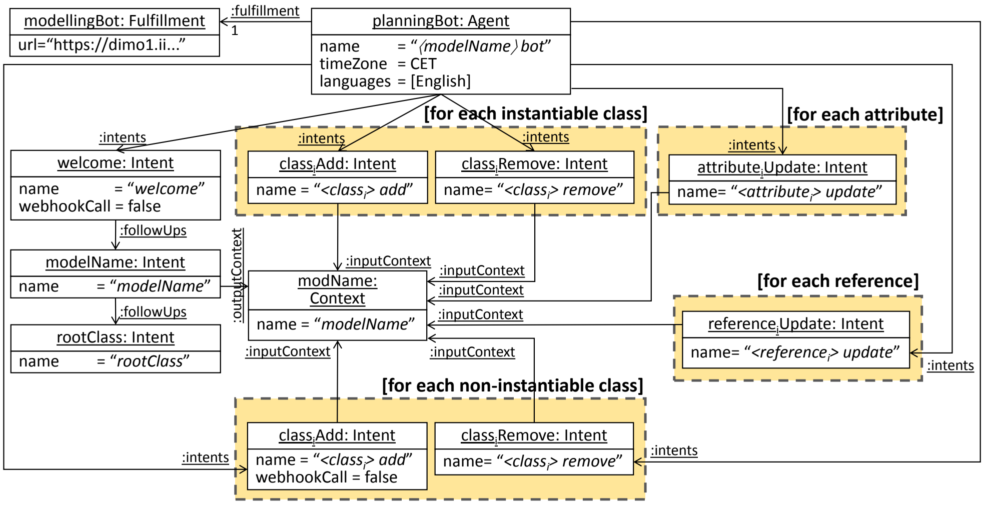


Figure 7 – Scheme of the generated DialogFlow chatbots.

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.

```
<create>? <natural-number>? <class-name>
   ( 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.

```
<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.

```
<att-name> of <class-name>? <id-value> (is | are) <att-value>
<id-value>('s)? <att-name> is <att-value>
<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"*.

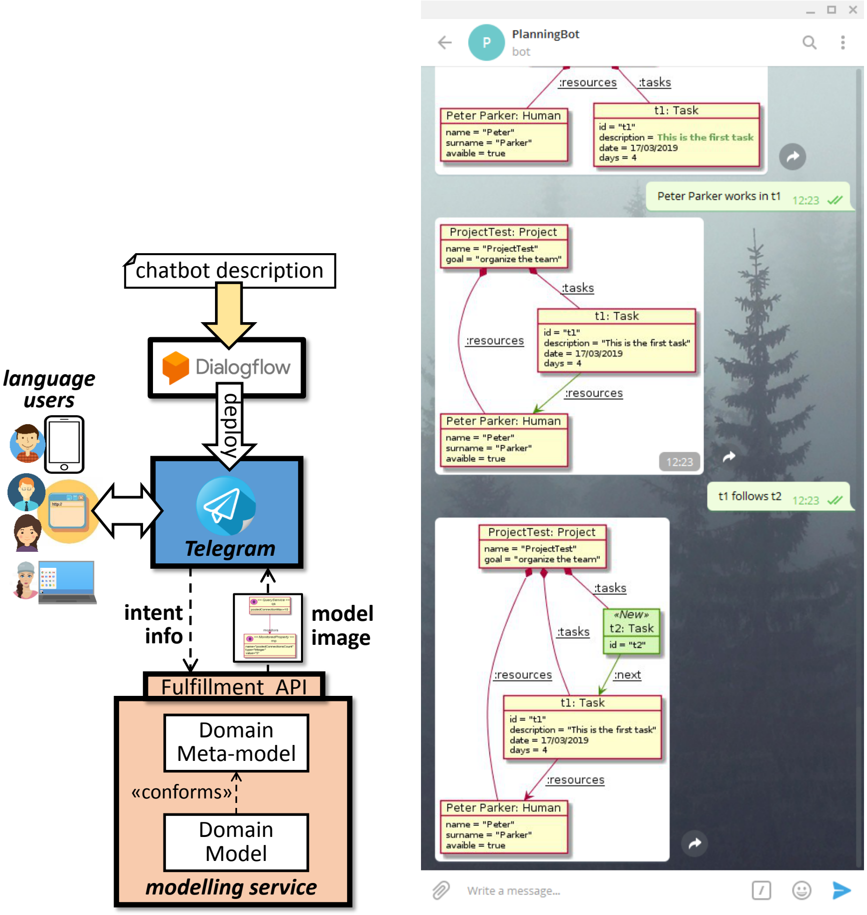
Regarding references, if their name is a noun, then we generate the training phrases using the templates for attribute modification in Listing 3. However, when their name is a verb, we use the two templates in Listing 4. In these templates, ⟨**ref-name**⟩ is the name and synonyms of the reference to 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.

```
<class-name>? <id-value> <ref-name> <natural-number>? <ref-class-name>? <ref-value>
<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.



Peter Parker works in t1 12:23

12:23

t1 follows t2 12:23

(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?

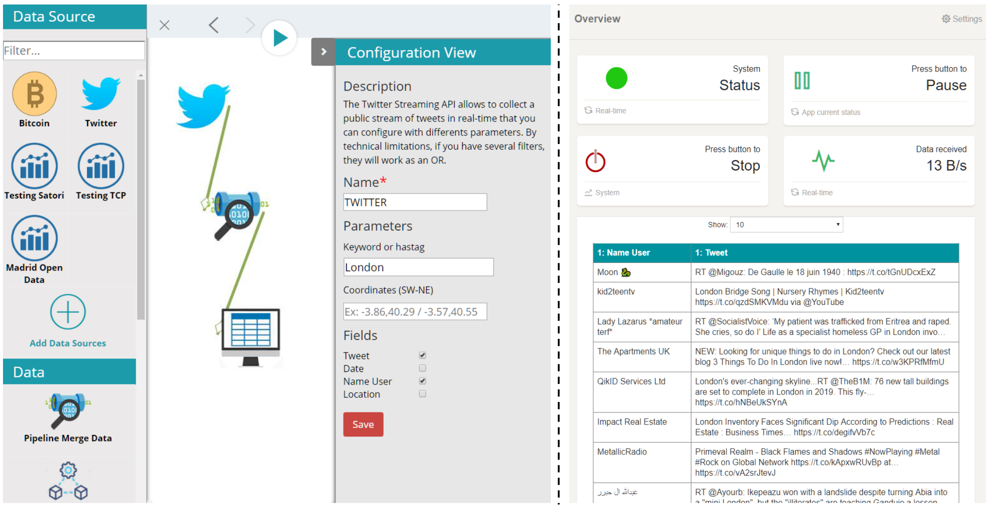
RQ2 What is the added value – in terms of functionality – that a modelling chatbot brings?

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.



Description

The Twitter Streaming API allows to collect a public stream of tweets in real-time that you can configure with differents parameters. By technical limitations, if you have several filters, they will work as an OR.

Name*

TWITTER

Parameters

Keyword or hastag

London

Coordinates (SW-NE)

Ex: -3.86,40.29 / -3.57,40.55

Fields

Tweet ☑
Date ☐
Name User ☑
Location ☐

Save

Show: 10

1: Name User	1: Tweet
Moon 🐉	RT @Migouz: De Gaulle le 18 juin 1940 : https://t.co/tGnUDcxExZ
kid2teentv	London Bridge Song | Nursery Rhymes | Kid2teentv https://t.co/qzdSMKVMdu via @YouTube
Lady Lazarus *amateur terf*	RT @SocialistVoice: 'My patient was trafficked from Eritrea and raped. She cries, so do I' Life as a specialist homeless GP in London invo…
The Apartments UK	NEW: Looking for unique things to do in London? Check out our latest blog 3 Things To Do In London live now!… https://t.co/w3KPRfMfmU
QikID Services Ltd	London's ever-changing skyline...RT @TheB1M: 76 new tall buildings are set to complete in London in 2019. This fly-… https://t.co/hNBeUkSYnA
Impact Real Estate	London Inventory Faces Significant Dip According to Predictions : Real Estate : Business Times… https://t.co/degifvVb7c
MetallicRadio	Primeval Realm - Black Flames and Shadows #NowPlaying #Metal #Rock on Global Network https://t.co/kApxwRUvBp at… https://t.co/vA2srJtevJ
عبدالله ال جبرر	RT @Ayourb: Ikepeazu won with a landslide despite turning Abia into a "mini London", but the "illiteratos" are teaching Ganduje a lesson

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`

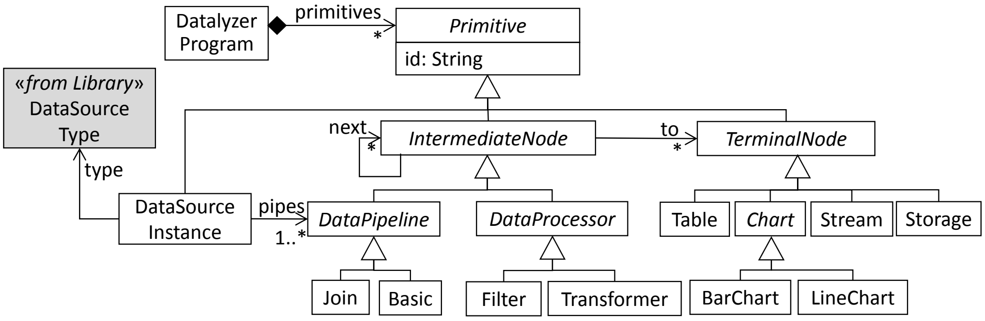


Figure 10 – Datalyzer meta-model excerpt.

(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

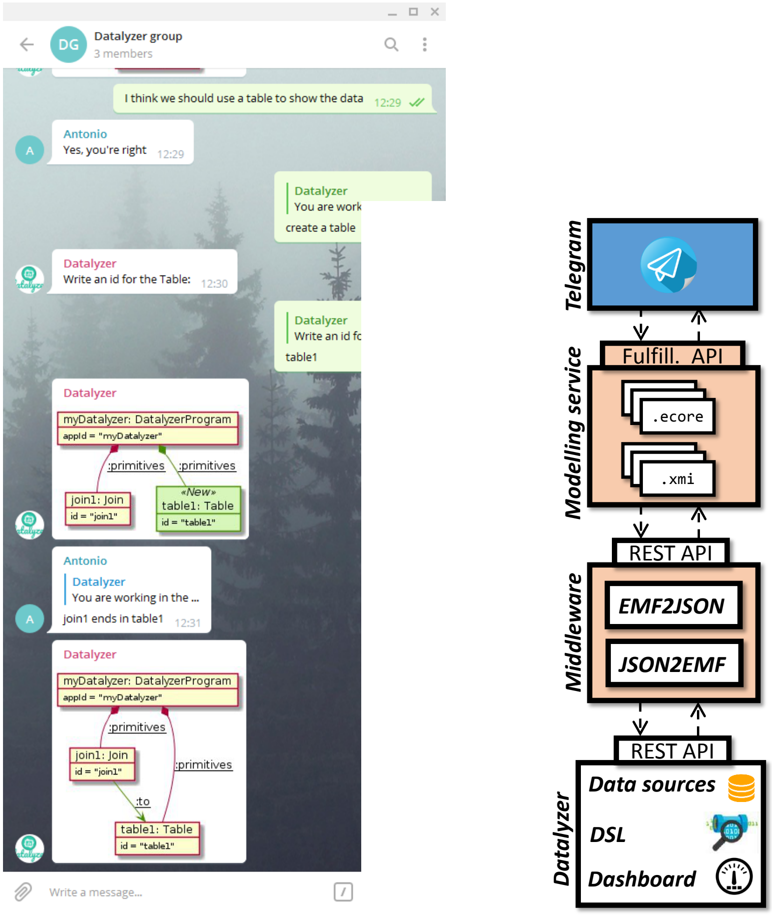
We used our approach to automatically generate a default NL concrete syntax model from the Datalyzer meta-model. In this model, the `NLClass` pointing to `DatalyzerProgram` was correctly identified as root, and all non-abstract domain classes were set to instantiable.

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:



Datalyzer group
3 members

I think we should use a table to show the data 12:29

Antonio
Yes, you're right 12:29

Datalyzer
You are work
create a table

Datalyzer
Write an id for the Table: 12:30

Datalyzer
Write an id fo
table1

Datalyzer

Antonio
Datalyzer
You are working in the ...
join1 ends in table1 12:31

Datalyzer

Write a message...

(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, 3^{rd} 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.

A.3 Choosing a chatbot development tool

Title	Choosing a chatbot development tool	
Journal	IEEE Software	
Impact Factor	2021 3.000	Q2, Software Engineering
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Sandra Juárez-Puerta	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1109/MS.2020.3030198	
Date	February 2021	
Abstract	Chatbots are programs that supply services to users via conversation in natural language, acting as virtual assistants within social networks or web applications. Companies like Google, IBM, Microsoft or Amazon have released chatbot development tools with different functionalities, capabilities, approaches and pricing models. With so many options, companies that want to offer services through chatbots can find choosing the right tool difficult. To help them make an informed choice, we review the most representative chatbot development tools with a focus on technical and managerial aspects.	

Choosing a chatbot development tool

Sara Pérez-Soler, Sandra Juárez-Puerta, Esther Guerra, Juan de Lara
Modelling and Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Abstract. *Chatbots are programs that supply services to users via conversation in natural language, acting as virtual assistants within social networks or web applications. Companies like Google, IBM, Microsoft or Amazon have released chatbot development tools with different functionalities, capabilities, approaches and pricing models. With so many options, companies that want to offer services through chatbots can find choosing the right tool difficult. To help them make an informed choice, we review the most representative chatbot development tools with a focus on technical and managerial aspects.*

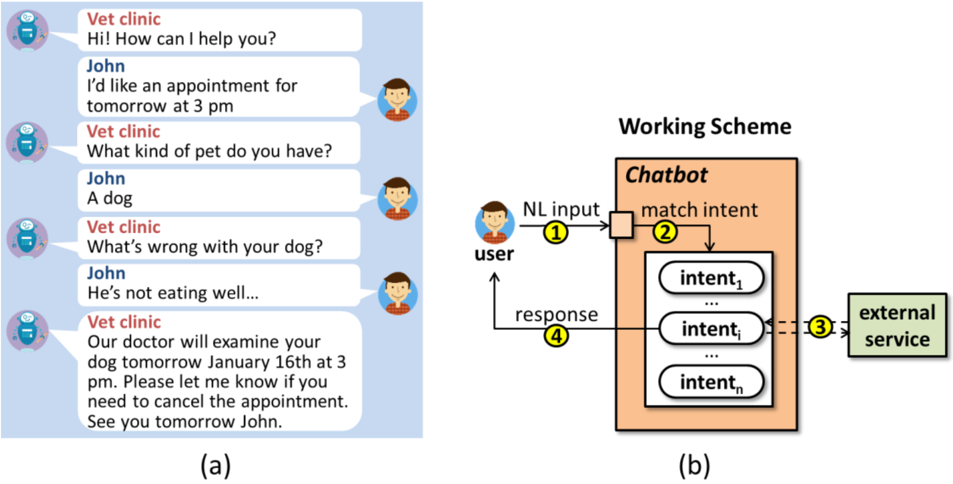
Keywords: Software Engineering, Chatbots, Natural Language Processing

1. Introduction

Chatbots are programs with a conversational user interface. Their popularity is rising because they enable accessing all sorts of services (e.g., booking flights, checking weather conditions) from web applications or social networks like Telegram, Twitter, Skype or Slack. This way, users can access those services without installing new apps and interacting with the service is simplified by the use of natural language (NL) [5].

Many companies are developing chatbots to automate customer support and provide ubiquitous access to the company services. At the same time, plenty of platforms and frameworks have emerged to ease chatbot construction. Large software companies like Google, Microsoft, IBM or Amazon have created chatbot development platforms, but many other alternatives exist. These platforms provide diverse functionality regarding natural language processing (NLP), the structure of the conversation flow, the ability to connect the chatbot to existing information systems, or the support for testing and deployment.

Choosing the best chatbot development tool for a particular need is difficult. Making an incorrect tool decision may lead to non-compliance with chatbot technical requirements or with software development company policies. Some websites and informal blogs compare some available options to build chatbots [1, 2, 3, 4], and researchers have identified aspects to consider in chatbot design (functional, integration, analytics and quality assurance) [8]. Instead, we analyse technical and managerial factors of the most representative chatbot creation tools, to help developers and managers in making informed choices on the optimal tools for their interest. This analysis can be used as a reading grid to select a tool based on technical criteria (e.g., "*we need a chatbot to access our current information system by text and voice, in both English and Spanish,*") and managerial constraints (e.g., "*my developers lack experience in developing chatbots, we do not have the capacity to deploy on-premises, and we are already using Amazon cloud*").



Vet clinic
Hi! How can I help you?

John
I'd like an appointment for tomorrow at 3 pm

Vet clinic
What kind of pet do you have?

John
A dog

Vet clinic
What's wrong with your dog?

John
He's not eating well...

Vet clinic
Our doctor will examine your dog tomorrow January 16th at 3 pm. Please let me know if you need to cancel the appointment. See you tomorrow John.

Figure 1: (a) Example of user interaction. (b) Working scheme of a chatbot.

2. What's in a chatbot?

A chatbot is a program supporting user interaction via conversation in NL, and normally accessible through the web or social networks. As an example, assume that a vet clinic has an information system with a database storing information about veterinarians and appointments, and decides to bring its services closer to customers by means of a chatbot to which customers can ask about opening hours and make appointments. This chatbot would allow the clinic to offer 24/7 service, reduce costs (e.g., decreasing customer telephone calls) and widen the potential customers. Figure 1(a) shows an example of a user interacting with the envisioned chatbot.

As Figure 1(b) shows, a chatbot is organized around intents that represent possible user's intentions and permit accessing the offered services. These intents typically reflect use cases of the chatbot. As an example, the chatbot for the clinic would define two intents: one to inform about opening hours, and another for making appointments. Upon receiving a user input in NL (label 1 in the figure), the chatbot identifies the matching intent (label 2). Depending on the intent, the chatbot may need to access external services, like the clinic database if the intent is setting an appointment (label 3). Finally, the chatbot replies to the user, e.g., confirming the appointment (label 4).

Figure 2(a) shows a process diagram with the main activities that designing a chatbot entails. The development process is not necessarily linear, but often requires iteration. Moreover, activities like validation and testing are needed throughout the process. Figure 2(b) contains a structural diagram (a UML class diagram) with the constituent elements of a chatbot. The numbers in this diagram identify the process step where the elements are defined.

First, developers must identify the *intents* that the chatbot will handle. While traditional applications typically offer their functionality via graphical interfaces, chatbots expose it through conversation. To match the intent corresponding to a user input phrase, developers can resort to NLP libraries – like the *Stanford Parser* [6] or the *Natural Language Toolkit* (NLTK) [7] – as they permit analysing the phrase structure and provide facilities for tokenizing and part-of-speech tagging, among others. This gives unlimited flexibility regarding the NL structure, but the implementation is costly. Hence, for narrow domains (like our clinic), it is simpler to train the chatbot with *training phrases* (i.e., examples of expected

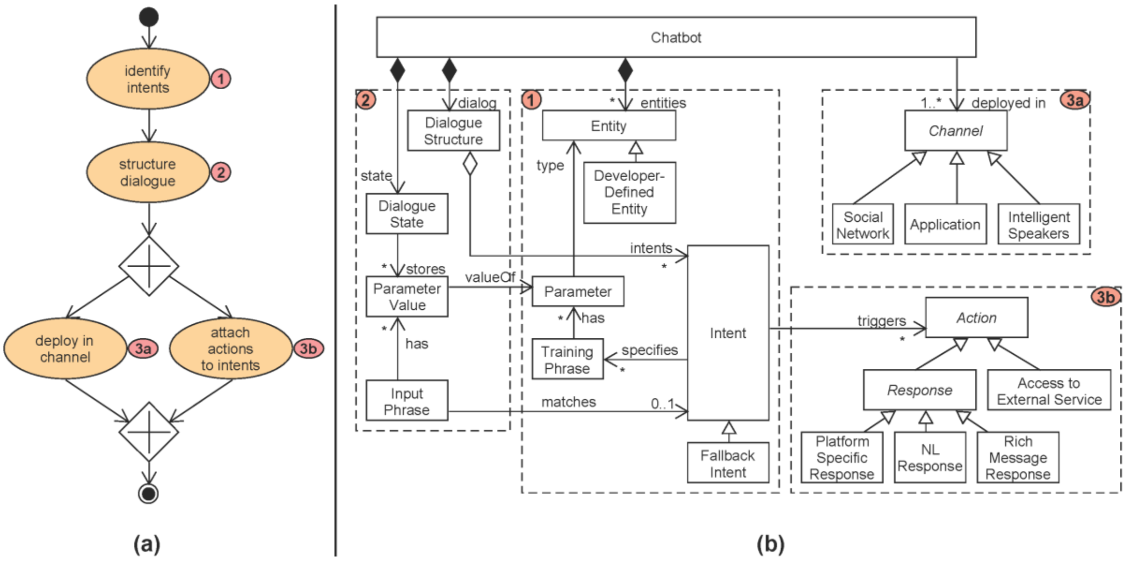


Figure 2: (a) Process diagram for chatbot design. (b) Structural diagram of chatbot concepts.

phrases) characterizing each intent. We can find libraries and services that apply machine learning for this purpose, like Microsoft's *LUIS* (https://luis.ai) or *Rasa NLU* (https://rasa.com). These libraries also support extracting *parameters* from phrases. A parameter is a piece of relevant information that needs to be extracted from a phrase, such as the date of an appointment. Parameters are conformant to a given *entity* type. Most chatbot development tools provide predefined entities (e.g., dates, numbers) and developers can define new ones (e.g., pet types). In addition, chatbots may define *fall-back intents,* used when the chatbot does not recognize the user utterance.

Besides intents, developers need to define the *dialogue structure* to accomplish a task. For example, after the user requests an appointment in Figure 1(a), the chatbot asks the kind of pet and problem, and only then the appointment is fixed. For this purpose, the chatbot needs to store the *dialogue state* – often in so-called *contexts* – to carry the information of previous *input phrases* through the stages of the conversation.

Moreover, developers need to identify the *actions* that each intent triggers. These may comprise invocations to *external services*, and include the chatbot *response* either in NL or using rich messages or mechanisms specific to the deployment platform. In our example, the chatbot needs to access the clinic information system to check for available slots and set appointments, and replies with the appointed date and time.

Finally, developers must deploy the chatbot in some *channel*. Typical channels are social networks, websites, or smart speakers like *Amazon Echo* or *Google Home*. In addition to NL, each channel may support specific interaction possibilities that can be exploited to obtain effective chatbots. For instance, to prompt the user to select among a small set of options (e.g., the available appointment slots within one day), presenting each option as a button can be less error-prone. However, different channels may support distinct interaction mechanisms. For example, Telegram supports buttons, but Twitter and intelligent speakers do not.

3. Choosing a tool based on technical factors

The growing popularity of chatbots has caused the emergence of many tools for their construction. These range from low-level NLP services helping in the encoding of intents and their training phrases, to comprehensive low-code development platforms covering most steps in the chatbot creation process.

Table 1 compares the main available software options for chatbot construction. It includes proposals of both large companies (*Dialogflow* by Google, *Watson* by IBM, *Lex* by Amazon, *Bot Framework* and *LUIS* by Microsoft) and younger chatbot specialized companies (*FlowXO*, *Landbot.io*, *Chatfuel*, *Rasa*, *SmartLoop*, *Xenioo*, *Botkit* which has been recently acquired by Microsoft, *ChatterBot* and *Pandorabots*). All are domain-independent but *Chatfuel*, which targets marketing applications.

The features analysed in the table stem from a thorough analysis of each tool. We distinguish between technical features (e.g., input processing) which are discussed in this section, and managerial features (e.g., pricing model) presented in the next section.

The first row in the table indicates whether the software is a library, a framework, a platform or a service. While platforms and frameworks offer support for the whole bot creation life-cycle, services and libraries support only some steps, typically related to NLP. Frameworks provide sets of classes that need to be complemented with custom code for each created chatbot, and hence chatbots are built via programming. Most platforms are cloud-based, low-code development environments to define chatbots graphically or via forms, and frequently support hosting the deployed chatbot logic for a channel. In addition, some platforms and frameworks (e.g., *Dialogflow*, *Bot Framework*, *Rasa*) also support the use of their NLP modules via services.

Rows 2–26 in the table analyse decisive technical dimensions when selecting a chatbot development tool. These comprise aspects related to the processing of the user input text (rows 2–7), the dialogue support (rows 8–13), the chatbot deployment (rows 14–15), the integration with other systems (rows 16–17), testing and development support (rows 18–22), execution support (rows 23–25) and security aspects (row 26).

Input processing. Some approaches allow defining the expected input phrases using regular expressions or patterns (row #2), while others permit specifying intents via training phrases and then apply NLP (row #3). In addition, platforms like *Landbot.io* also support user inputs by means of buttons and widgets. Most approaches based on NLP can identify parameters in the input phrases, with the exception of *Chatfuel* and *ChatterBot* (row #4). Another important aspect in NLP is the language support (row #5). All approaches consider some of the most spoken languages (English, Spanish), and some platforms excel for their wide language support (e.g., *Dialogflow* includes 22). Interestingly, *Rasa* can use pre-trained language models (e.g., *fastText* word vectors are available for hundreds of languages [9]) but developers can train their own. Only a few approaches – the NLP service *LUIS*, *Watson*, *Lex*, *Bot Framework,* and the Enterprise non-free edition of *Dialogflow* – provide sentence sentiment analysis, which can be useful in specific domains such as marketing. Finally, in addition to text, several approaches natively support voice-based interaction (row #7). This interaction kind could be added by hand to approaches based on programming languages (e.g., *Botkit*) or which are open source.

Dialogue. This dimension looks at the capabilities to organize the conversation flow. All platforms and most frameworks automatically store the parameter values extracted from user phrases to allow their reuse in the future, while libraries require programming this facility (row #8). This storage can be volatile (active only during the current user interaction) or persistent. Intents and entities (rows #9 and #10) are common primitives of platforms like *Dialogflow*, *Watson* and *Lex*. Approaches

Table 1: Comparison of chatbot libraries, frameworks, platforms and services.

			Dialogflow (Google) [v2]	Watson (IBM) [v2]	Lex (Amazon) [07/06/2020]	Bot Framework + LUIS (Microsoft) [v4]	FlowXO [07/06/2020]	Landbot.io [07/06/2020]	Chatfuel [07/06/2020]	Rasa [10.1.2]	SmartLoop [07/06/2020]	Xenioo [07/06/2020]	Botkit (part of Bot Framework) [4.9.0]	LUIS [05/192020]	ChatterBot [1.0.5]	Pandorabots [07/06/2020]
Technical Factors		1. Kind (Library, Framework, Platform, Service)	P	P	P	F	P	P	P	F	P	P	F	S	L	P
	Input processing	2. Regular expressions/patterns	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
		3. NLP for phrase match	✓	✓	✓	✓			✓	✓	✓	✓		✓	✓	
		4. Text processing to obtain phrase parameters	✓	✓	✓	✓				✓	✓	✓		✓		✓
		5. Number of languages: very high (≥50), high (≥10), some (<10), 1 (represented with flag)	h	h	🇺🇸	h	h		h	v		s		h	v	
		6. Sentiment analysis	✓	✓	✓	✓								✓		
		7. Speech recognition	✓	✓	✓	✓						✓		✓		
	Dialogue	8. Storage of phrase parameters: volatile, persistent, both	b	b	b	b	b	v	v	b	v	v		v		v
		9. Support for intents	✓	✓	✓	✓	✓			✓	✓			✓		✓
		10. Support for entities: predefined, user-def, both	b	b	b	b	p	p		b	b	b		b		
		11. Dialogue structure: tree, followup intents, dsl	f	f	f	f	t	t	t	t	f	t			f	d
		12. Utterances to reengage users					✓		✓		✓	✓				
		13. Specification of chatbot answers	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓
	Deployment	14. Integration with social networks/websites: high (≥10), some (<10), 1 (represented with logo)	h	s	s	h	s	✆ www	f	h	s	s	s			s
		15. Interaction support for specific social networks	✓			✓						✓				✓
	Sys. integ.	16. Call to services from chatbot	✓	✓	✓	✓	✓	✓	✓	✓					✓	
		17. Chatbot usage via API	✓	✓	✓						✓	✓		✓		✓
	Development and testing	18. Pre-built components: chatbot templates, intents, small talks, services	cts	c	i	cs	c	c							c	t
		19. Version control: native, code-based	n	n	n	c				c			c		c	
		20. Chat console for testing	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓
		21. Debug mechanisms	✓			✓				✓			✓		✓	
		22. Validation support	✓													
	Execution	23. Hosted deployment	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓		✓
		24. Support for analytics	✓	✓		✓	✓	✓			✓	✓				
		25. User message persistence	✓		✓	✓	✓	✓	✓			✓	✓			
	Sec	26. Cloud security	✓	✓	✓	✓								✓		
Managerial Factors	Organization	27. Pricing model: free, pay-as-you-go, quota, advanced feats	fp	fp	fp	fpa	fq	fa	fa	fa	fa	fq	f	fq	f	fqa
		28. Developer expertise: low, high	l	l	l	h	l	l	l	h	l	l	h	h	h	l
	Development	29. Code hosting: external, on-premises	e	e	e	o	e	e	e	o	e	e	o	o	o	e
		30. Group work				✓				✓			✓	✓	✓	
		31. i8n	✓													
		32. Open source								✓			✓		✓	
	Operational	33. New channels								✓			✓	✓	✓	
		34. No vendor lock-in								✓			✓	✓	✓	

supporting NLP define intents by sets of training phrases. These phrases may be examples of expected user utterances, or to improve the user experience, they may be obtained from existing conversation logs (e.g., when migrating a traditional customer support system into a chatbot). Regarding the dialogue structure (row #11), we find two main definition styles: explicitly by means of a conversation tree where nodes correspond to dialogue steps, or implicitly via dependent contexts and *follow-up* intents which are activated upon matching their parent intent (e.g., an intent for making appointments which declares a follow-up intent to inform the kind of pet). More differently, *Pandorabots* uses the Artificial Intelligence Markup Language (AIML, http://www.aiml.foundation/), an XML format from the '90s aimed to be a scripting standard for chatbots. Being based on templates, it is in stark contraposition to modern approaches based on NLP. Some platforms also permit defining utterances that the chatbot can use to reengage unresponsive users (row #12). Finally, all approaches but *LUIS* and *Botkit* permit specifying the chatbot answers (row #13).

Deployment. While some approaches allow deploying chatbots in many social networks, others target specific ones (row #14). For example, *Chatfuel* chatbots are specific for Facebook messenger, *Landbot.io* chatbots can be deployed just on *WhatsApp Business* and websites, while *Dialogflow* has 15 channel integrations including websites, services like Skype, intelligent speakers and social networks like Slack, Viber, Twitter and Telegram. Libraries and services lack deployment options, since this is out of their scope. In addition, *Dialogflow*, *Bot Framework*, *Xenioo* and *Pandorabots* permit including custom interaction mechanisms for the selected channel, like buttons in Telegram (row #15).

System integration. Several approaches enable calling services from the chatbots (row #16). In some cases, like *Dialogflow*, this is done by associating the URL of the service to an intent, so that matching the intent triggers a POST message to the service. In other cases, it is possible to define programs with custom code. For this purpose, *Dialogflow* supports Cloud Functions for Firebase, and *Lex* supports AWS lambdas.

Conversely, some approaches offer an API that permits integrating parts of the chatbots with existing applications (row #17). For example, *Dialogflow* chatbots can be used programmatically to check the most probable matching intent given a user phrase.

Development and testing. Some tools offer pre-built components that can be added into new chatbots (row #18). These include generic chatbot templates (e.g., for a coffee shop or a hotel booking system), predefined intents, predefined small talks (answers to simple phrases and questions), or services (e.g., to build a *Q&A* chatbot from a knowledge base). Regarding version control (row #19), all frameworks and libraries rely on code and can be used with any generic version control system, while only some platforms (*Dialogflow*, *Watson* and *Lex*) give native support for versioning though this is simpler than state-of-the-art versioning systems like *GitHub*.

As for testing, most approaches provide a web chat console to test the chatbots manually (row #20). For debugging (row #21), frameworks and libraries can rely on the support of the programming language, while only one platform (*Dialogflow*) offers debugging facilities to inspect the matched intent and related information. In addition, *Dialogflow* incorporates checks of the chatbot quality, such as detecting intents with similar training phrases (row #22).

Execution. Once a chatbot is defined, all platforms and most frameworks support its execution on their clouds (row #23). This solution can be optimal for many companies, especially if they already use the cloud services of the platform vendor (e.g., Google, Azure or AWS); however, this may not be always suitable. In some cases, like *Watson*, there is a special pricing plan to deploy the chatbot on third-party clouds. Finally, some approaches permit obtaining analytics about the chatbot usage (row #24) or persisting the user phrases (row #25). Developers might find the latter feature useful to adjust the accuracy of the intent recognition and improve the user experience [10]. Approaches like *Watson* automate this task, while others like *Dialogflow* require uploading the conversation logs and retraining.

Security. Chatbots may need to incorporate security aspects, especially if they work with private user data. While in general, implementing any security capability is the developers' responsibility, some tools can provide a security layer atop the cloud where the chatbot is deployed (row #26). Hence, approaches without deployment services do not offer this possibility natively. Instead, *Dialogflow*, *Watson*, *Lex* and Azure (Microsoft cloud for the *Bot Framework* and *LUIS*) provide a layer with features like firewalls; authentication and authorization when used via API; and secure connections (e.g., SSL or HTTPS/TLS). In addition, social networks like Whatsapp or Telegram support message encryption and user authentication.

4. Adding managerial factors to the equation

In addition to technical factors, some managerial factors may influence the selection of a development tool. Rows 27–34 in Table 1 classify those factors among organizational, related to development, or operational. We elicited those factors by a thorough analysis of the tools' features, and classified them using as a basis typical concerns in software projects.

Organizational factors. A critical selection factor is the pricing model of the approach (row #27). Most offer a free version suitable for small businesses or for experimentation (e.g., *Dialogflow* provides five free assistants and *Watson* supports 10,000 API calls). In addition, they provide other pricing models, typically collecting small fees for every interaction with the chatbot (the pay-as-you-go option of *Dialogflow*), limiting the number of interactions or active chatbots (the different plans of *FlowXO*), or supplying advanced features (e.g., webhooks in *Landbot.io* are not free).

The expertise of the development team on chatbot-related technology is also important (row #28). Development platforms allow creating simple chatbots with no need for coding and require less expertise than approaches based on programming, though these latter are less constrained.

Development related factors. Like any kind of software, chatbot construction should follow proper engineering processes. In this respect, using a platform may be problematic if the chatbot development has to be harmonized with the company development culture and processes. For example, platforms host the chatbot specifications on their clouds, while the backend needs to reside in a different place; instead, chatbots developed with libraries, frameworks and services can run on-premises (row #29). Likewise, some code facilities such as versioning or debugging are standard for frameworks and libraries but may be unavailable for some platforms. The same applies to group work (row #30): platforms currently do not support synchronous collaborative development, so working on different parts of a chatbot cannot be parallelised among developers.

Depending on the domain or the company strategy, the need to support several languages (i8n) can be necessary (row #31). Rather than developing a chatbot for each language, platforms like *Dialogflow* offer multi-language support by enabling the specification of different training phrases for each language over the same intent.

Interestingly, among the reviewed approaches, only the community edition of *Rasa*, *Botkit* and *ChatterBot* are open source (row #32). No platform is open source, which may result in vendor lock-in, but it is possible to make public the chatbot specifications built with any platform.

Operational factors. Once a chatbot is in operation, the need to deploy it in novel channels or new versions of existing ones may arise (row #33). If the chatbot was developed using a platform, the available deployment options might be limited (e.g., *Watson* does not provide out-of-the-box deployment in Telegram). Libraries and (extensible) frameworks like *Rasa*, *Botkit*, *LUIS* and *ChatterBot* are more flexible, as they allow the manual implementation of the required deployment.

Finally, platform-based approaches imply vendor lock-in as there are currently no migration tools using neutral exchange formats between platforms (row #34); however, an advantage of platforms is the ability to use the services of the provider (IBM, Google). Instead, libraries and frameworks require coding the chatbot logic in a programming language (like Python in case of *Rasa*), which brings more independence and safety with respect to possible policy changes of the platform owner company. This independence is stronger in open-source systems (row #32) since they could even be personalized to the developer needs.

5. Building a chatbot in practice

Practitioners can exploit the information in Table 1 to select the best tool depending on the scenario. While this analysis can be hand-crafted, we envision a recommender system that automatically identifies the optimal tools from the chatbot requirements.

As an illustration, let's assume two scenarios for our vet clinic chatbot. In the first one, the clinic wants to reach as many potential clients as possible, so it asks for a chatbot that is multi-language and works on different social networks and intelligent speakers. Moreover, the software company that will develop the chatbot lacks the infrastructure to host the bot. Given these requirements, the only suitable chatbot creation tool is *Dialogflow*.

In the second scenario, the clinic is in a process of expansion so the chatbot may be likely extended in the future. Hence, the software company is thinking of using either *Rasa* or *Botkit* to avoid vendor lock-in. Since the company has an expert team of Python developers, and wants to have support for debugging and testing, it opts for *Rasa*.

We have built prototypical chatbots using the tools selected in the scenarios: *Dialogflow* and *Rasa*. The chatbots communicate with a backend that holds a database written in Java and PostgreSQL. The chatbots for Telegram, including their specification, are available at https://github.com/SaraPerezSoler/VetClinic.

The chatbot specification in *Dialogflow* has four intents: a welcome intent, a fall-back intent, an intent to query the opening hours, and another to set appointments. The welcome and fall-back intents were predefined in *Dialogflow* and reused in our chatbot without modification. To make the chatbot multi-language, each intent has to be trained with phrases in every targeted language. The appointment intent has a follow-up asking for the type of pet. This control flow is specified via a context. We defined an entity to recognise pet types, and reused the *date* and *time* system entities. The backend is accessed by a webhook that calls the database service via a POST request; alternatively, the behaviour could be implemented with a JavaScript in-line editor available in the platform. The deployment in Telegram was straightforward using *Dialogflow*'s integration options, and there are integrations for intelligent speakers as well.

Differently from *Dialogflow*, creating a chatbot in *Rasa* is not done via a graphical interface, but requires programming in Python and defining configuration files (YAML and markdown) storing the entities, intents, conversation flow, training phrases, bot responses, actions, forms, NLP configuration and credentials to access external services. The *Rasa* chatbot has one fall-back action and three intents:

greeting, *time* and *make_appointment*. To define the parameters of the last intent, we subclassed a specific *Rasa* class to store the name and type of the parameters, validation methods, and other details. The chatbot actions (e.g., querying the database, calling external services) were programmed in Python as well. The chatbot behaviour can be debugged and tested using standard Python tooling. Unlike *Dialogflow*, the developer must perform the chatbot deployment as *Rasa* does not host bots.

6. Open Challenges

Overall, the existing tools cover a wide spectrum of possibilities to ease chatbot creation in different scenarios. However, designing, developing and testing chatbots still pose some challenges. First, most platforms offer general, informal guidelines for chatbot design, but design patterns and quality metrics for chatbots are missing. With regards to development, most tools rely on training phrases to specify intents; while this is suitable in closed domains, supporting less constrained conversations would require the tools to incorporate more sophisticated NLP mechanisms [13, 14] and better support to expand the training set using techniques such as reinforcement learning (e.g., via trial-and-error conversations with real or simulated users). Also related to quality, existing tools give poor support for testing chatbots in a systematic and automated manner; at best, they provide a console for manual testing, and basic debugging mechanisms (rows 20–21 in Table 1). Some dedicated testing tools are emerging, like https://www.botium.at/.

Ultimately, the success of a chatbot depends on its usability and the user experience. Some technical factors in Table 1 may help to improve this usability: NLP enables more natural conversations, phrase parameters avoid users to provide a different sentence per piece of information, sentiment analysis can contribute to better grasp the meaning of a phrase and act accordingly, speech recognition supports spoken conversation, rich dialog structuring mechanisms allow more sophisticated conversation flows, and message persistence can be exploited to improve chatbot accuracy by the analysis of real conversations. To complement this, chatbot development tools should invest in embedding guidelines and heuristics targeted to chatbot usability [11, 12].

Chatbot development tools are rapidly expanding, but we believe that after diversification comes unification. The analysed technologies use their own proprietary formats to define chatbots, and automated migration tools are missing. To unify the different approaches, the W3C is developing a standard for conversational agents (https://www.w3.org/community/conv/), and some open-source initiatives aim to integrate the best of every chatbot platform, helping to solve the vendor lock-in problem [15].

References

1. PAT Research. "*How to select the best chatbot platforms for your business?*". Available at https://www.predictiveanalyticstoday.com/what-is-chatbot-platform/ (last visited Jan-2020).

2. OMetrics. "*2019 chatbot platform comparison reviews*". Available at: https://www.ometrics.com/blog/chatbot-platform-comparison-reviews/ (last visited Jan-2020).

3. Davydova. "*25 chatbot platforms: A comparative table*". Chatbots Journal (May 2017). Available at: https://chatbotsjournal.com/25-chatbot-platforms-a-comparative-table-aeefc932eaff (last visited Jan-2020).

4. VentureHarbour. "*10 best chatbot builders in 2019*". https://www.ventureharbour.com/best-chatbot-builders/ (last visited Jan-2020).

5. Lebeuf, Storey, Zagalasky. "*Software bots*". IEEE Software 35(1): 18–23 (2018).

6. Marneffe, Maccartney, Manning. "*Generating typed dependency parses from phrase structure parses*". Proc. LREC'06: 449–454. See also https://nlp.stanford.edu/software/lex-parser.html (last visited Jan-2020).

7. Bird, Klein, Loper. "*Natural language processing with Python - Analyzing text with the Natural*

Language Toolkit". O'Reilly 2009. See also https://www.nltk.org/ (last visited Jan-2020).

8. Pereira, Díaz. "*Chatbot dimensions that matter: Lessons from the trenches*". Proc. ICWE'18: 129–135.
9. Grave, Bojanowski, Gupta, Joulin, Mikolov. "*Learning word vectors for 157 languages*". Proc. LREC 2018.
10. Hancock, Bordes, Mazaré, Weston. "*Learning from dialogue after deployment: Feed yourself, chatbot!*". Proc. ACL (1) 2019: 3667–3684.
11. Ren, Castro, Acuña, de Lara. "*Usability of chatbots: A systematic mapping study*". Proc. SEKE'19: 479–617.
12. Haptiki.ai. "*10 usability heuristics to design better chatbots*". https://haptik.ai/blog/usability-heuristics-chatbots/ (last visited Jan-2020).
13. Pérez-Soler, Guerra, de Lara, Jurado. "*The rise of the (modelling) bots: Towards assisted modelling via social networks*". Proc. ASE'17: 723–728.
14. Arora, Sabetzadeh, Nejati, Briand. "*An active learning approach for improving the accuracy of automated domain model extraction*". ACM Trans. Softw. Eng. Methodol. 28(1): 4:1–4:34 (2019).
15. Daniel, Cabot, Deruelle, Derras. "*Multi-platform chatbot modeling and deployment with the Jarvis framework*". Proc CAiSE'19: 177–193. See also https://xatkit.com/ (last visited Jan-2020).

A.4 Using the SOCIO Chatbot for UML Modeling: A Second Family of Experiments on Usability in Academic Settings

Title	Using the socio chatbot for uml modeling: A second family of experiments on usability in academic settings	
Journal	IEEE Access	
Impact Factor	2021 3.476	Q2 Computer Science, Information Systems
Authors	Ranci Ren	Universidad Autónoma de Madrid
	Sara Pérez-Soler	Universidad Autónoma de Madrid
	John W. Castro	Universidad de Atacama
	Oscar Dieste	Universidad Politécnica de Madrid
	Silvia T. Acuña	Universidad Autónoma de Madrid
Doi	10.1109/ACCESS.2022.3228772	
Date	December 2022	
Abstract	After improving the SOCIO chatbot prototype model, we wanted to know how/if its usability has changed. An evidence-based empirical evaluation of the usability of SOCIO V1 (updated version) requires an extensive verification of the experimental results. A family of experiments is a method of verification whereby we can check if the experimental results are reproducible. Through comparison with the updated control tool Creately, we aimed to gain a better understanding of the usability of the collaborative modelling chatbot and how it could be improved based on experimental evidence of changes in terms of efficiency, effectiveness, satisfaction, and quality. A total of 87 students from three countries were recruited. We conducted a family of three experiments to compare the usability of SOCIO V1 and updated Creately in academic settings. Students appeared to be more satisfied with SOCIO V1, and SOCIO V1 scored better on completeness. There were no significant differences between the two tools regarding efficiency and quality. This study provides evidence on how to employ a family of experiments to improve chatbot usability and enrich knowledge on chatbot usability experimentation.	

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022. Doi Number

Using the SOCIO Chatbot for UML Modeling: A Second Family of Experiments on Usability in Academic Settings

Ranci Ren[1], Sara Pérez-Soler[1], John W. Castro[2], Oscar Dieste[3], and Silvia T. Acuña[1]

[1]Departamento de Ingeniería Informática, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Madrid, Spain
[2]Departamento de Ingeniería Informática y Ciencias de la Computación, Universidad de Atacama, Copiapó, Chile
[3]Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid, Madrid, Spain

Corresponding author: John W. Castro (john.castro@uda.cl)

This research was funded by the Spanish Ministry of Science, Innovation and Universities research grant PGC2018-097265-B-I00 and the FINESSE project (PID2021-122270OB-I00). This research was also supported by the Madrid Region R&D program (FORTE project, P2018/TCS-4314) and the SATORI-UAM project (TED2021-129381B-C21).

ABSTRACT After improving the SOCIO chatbot prototype model, we wanted to know how/if its usability has changed. An evidence-based empirical evaluation of the usability of SOCIO V1 (updated version) requires an extensive verification of the experimental results. A family of experiments is a method of verification whereby we can check if the experimental results are reproducible. Through comparison with the updated control tool Creately, we aimed to gain a better understanding of the usability of the collaborative modeling chatbot and how it could be improved based on experimental evidence of changes in terms of efficiency, effectiveness, satisfaction, and quality. A total of 87 students from three countries were recruited. We conducted a family of three experiments to compare the usability of SOCIO V1 and updated Creately in academic settings. Students appeared to be more satisfied with SOCIO V1, and SOCIO V1 scored better on completeness. There were no significant differences between the two tools regarding efficiency and quality. This study provides evidence on how to employ a family of experiments to improve chatbot usability and enrich knowledge on chatbot usability experimentation.

INDEX TERMS Chatbot, usability, family of experiments.

I. INTRODUCTION

Collaborative modeling is an approach that deals with methods, processes, and tools for enhancing collaboration, communication, and coordination (3C) in teamwork [1]. Synchronization is used pervasively in software engineering (SE) collaborative modeling, providing for simple and efficient design changes in the collaboration environment. Many real-time collaboration modeling tools have been developed for target groups, e.g., Lucidchart, Creately, and Cacoo. Social networks like Telegram and Twitter have gained popularity and recognition [2]. With a view to integrating collaborative modeling tools into social networks, our colleague de Lara and his research group developed SOCIO chatbot (nick @ModellingBot), a collaborative modeling chatbot integrated into Telegram and Twitter [3]. SOCIO chatbot is an alternative collaborative modeling option to help stakeholders from different backgrounds perform lightweight tasks [3].

Usability deals with all sorts of activities related to software that is under development or has already been developed. Usability is defined in ISO/IEC50 25010:2011 [4] as a subset of quality in use, characterized explicitly by efficiency, effectiveness, and satisfaction. Experimentation is critical for evaluating usability in SE research [5]. Experimentation is a valuable tool for all software engineers involved in evaluation [6]. Back in 1998, Tichy reported his perspective on experimentation in software engineering (ESE) [7] as follows: “Experimentation can help build a reliable base of knowledge and thus reduce uncertainty about which theories, methods, and tools are adequate.” Nowadays, however, ESE is still a young and immature field where there is much debate on the appropriate research typology and evaluation criteria. Additionally, experiment replication types are not standardized at either the intra or interdisciplinary level [5].

A single experiment is unlikely to output reliable empirical results [5]. The outcomes of the experiment should be validated by replication. Lykken claimed in 1968 that "the majority of theories should be evaluated through multiple corroborations and the majority of empirical generalizations through constructive replication" [8]. Empirical evaluation has evolved considerably since its early beginnings, and the need for replication has been widely acknowledged in various scientific disciplines, including social science, business, and philosophy [5]. Replications of experiments have proven the need to be careful about accepting evidence that has not been subjected to strict corroborations [5]. To increase the robustness of the gathered experimental evidence, SE experiment replication is an indispensable part of ESE research [9]. The general purpose of replication is to check a previously observed finding. If the same results are reproduced in different replications, we can infer that these results are regularities existing in the portion of reality under study [10].

Quantitative analysis is widely used in experimental analysis and usability evaluation. Quantitative analysis interprets hard data collection [11]. However, qualitative analysis is a valuable paradigm for investigations where the data cannot be expressed numerically due to the complexity of the subjective characteristics and opinions involved. Thematic analysis is one of the most common forms of qualitative analysis. Thematic analysis is widely used across a range of epistemologies and research questions [12]. Thematic analysis has a number of advantages for evaluating the feedback from participants [12], [13], [14]: (i) researchers can apply a highly flexible approach that can be adapted to the needs of thematic analysis, (ii) it is an effective strategy for comparing and contrasting the perspectives of various research participants, revealing similarities as well as differences, and (iii) it is advantageous for summarizing significant characteristics of an extensive data set, as it helps to create a concise and ordered report.

This paper investigates a modified version of SOCIO with improved usability characteristics (SOCIO V1), based on the findings of a previous family of experiments. This second family of experiments tests whether or not SOCIO V1 consolidates the implemented usability characteristics. This article reports one of a number of families of experiments. In our case, the experimentation is performed in an academic setting, because the Unified Modeling Language (UML) modeling task is performed by senior computer engineering and mathematics students.

Our family of experiments aims to answer the following research question (RQ): How can chatbot usability be improved based on evidence from a family of experiments in academic settings?

In response to the research question, we designed an identical experiment for each experiment. We quantitatively analyzed the data using violin plots, descriptive statistics, and meta-analysis combined with linear mixed models (LMM) for each metric of each variable. Then we complemented the quantitative analysis by means of thematic analysis. The main contributions of the paper are: (1) the provision of evidence to enrich the body of knowledge to improve chatbot usability through the family of experiments, (2) demonstration of how chatbot usability can be improved by means of the family of experiments; and (3) provision of a summary and suggestions based on user feedback on how to improve software modeling.

The remainder of the paper is structured as follows. Section 2 describes the experiment background, indicating our improvement based on previous work. Section 3 reviews related work. Section 4 describes the design of our family of experiments. Section 5 reports the experimental result and quantitative and qualitative analysis. Section 6 describes the threats to validity. Section 7 discusses the experimental results. Section 8 outlines the conclusions and future work.

II. BACKGROUND

The first family of experiments comparing the usability of the basic version of SOCIO and Creately [15] was conducted in 2019. In this family of experiments, we adopted Creately (creately.com) as the control tool for comparison with the SOCIO chatbot, as Creately is one of the most commonly used modeling tools [16]. Creately is a web-based real-time collaboration tool for creating more than 50 types of diagrams, including UML diagrams.

SOCIO chatbot is a collaboration tool for creating class diagrams. By communicating with SOCIO in natural language (English), the team could create a class diagram in a group chat on Telegram or Twitter. From our first family of three experiments implementing a basic version of the SOCIO chatbot, we observed quantitative and qualitative feedback from this study, involving 132 participants. Quantitative data results provide: (1) proof of unsatisfactory chatbot usability, and (2) insights on how to improve chatbots. The conclusion and future work of the previous publication [15] lists the usability improvements for SOCIO as follows:

- *Provide more help.* As some participants complained that they did not know where they went wrong when the chatbot did not understand their commands, they suggested the need for better help. Other participants said they were using the chatbot for the first time and therefore needed more help from the help page and during the interaction with the chatbot.
- *Delete any element that the user wants to delete, regardless of whether it was created by themselves or another team member.* In fact, some participants in the first experiments of the family suggested that the /undo command should be modified to enable a participant to undo his/her own action instead of the last action performed by the team.
- *Beautify the user interface.* Regarding the interface, some participants claimed that the look of the class diagram generated by the chatbot is old-fashioned and unchangeable.

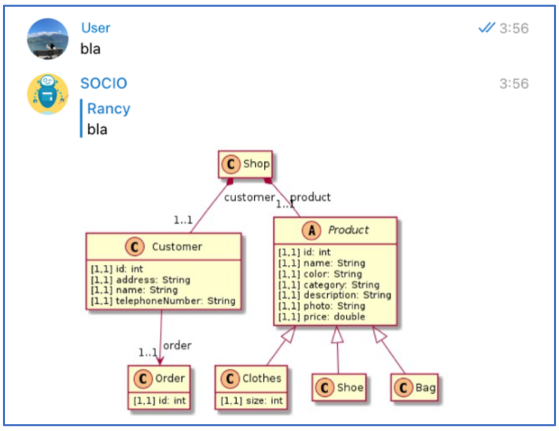
After discussing with HCI experts and the entire SOCIO chatbot developer team, we prioritized the aspects on the list according to the evidence that we gathered from the results of the data aggregation. We decided to develop three updated versions with different advances. The changes that we made to versions 2 and 3 are outlined in Appendix A.

Common Change: Change the guidance and help page. To provide more help to users, the following changes were made to all three updated versions:

1. Show the attribute types accepted by the chatbot as tips (int, double, float, date, string).
2. Update the guidance page in both English and Spanish (the native language of our subjects).
3. Provide examples on the help page to better explain the commands to help build the class diagram. For instance, we specify that point 3.5.6 would help relate entities and point 1.2 is helpful for directly making a command.

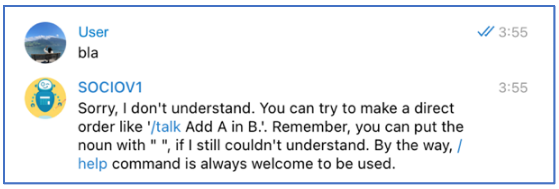
Updated Version 1 (SOCIO V1): Alternative context-sensitive help. Apart from modifying the help page, we decided that, with a view to providing users with more help, the chatbot should have more than one optional response when it does not understand the user's command. Note that the improvement of context-sensitive help messages only affects SOCIO V1 for Task 1 and Task 2 and does not affect Creately.

1. When the user's command is properly formatted but is not understood by the chatbot, the SOCIO V1 chatbot sends an unchanged the project diagram. In the light of this, we modified the response to be an autoreply, alerting the user that the chatbot does not understand the command and providing some sample sentences that the chatbot can understand (see Figs. 1 and 2).


User ✓✓ 3:56
bla

SOCIO 3:56
Rancy
bla

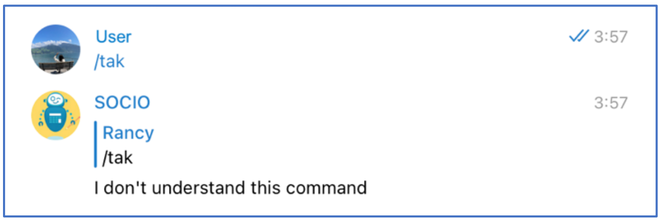
FIGURE 1. **Before the first modification of Version 1.**


User ✓✓ 3:55
bla

SOCIOV1 3:55
Sorry, I don't understand. You can try to make a direct order like '/talk Add A in B.'. Remember, you can put the noun with " ", if I still couldn't understand. By the way, /help command is always welcome to be used.

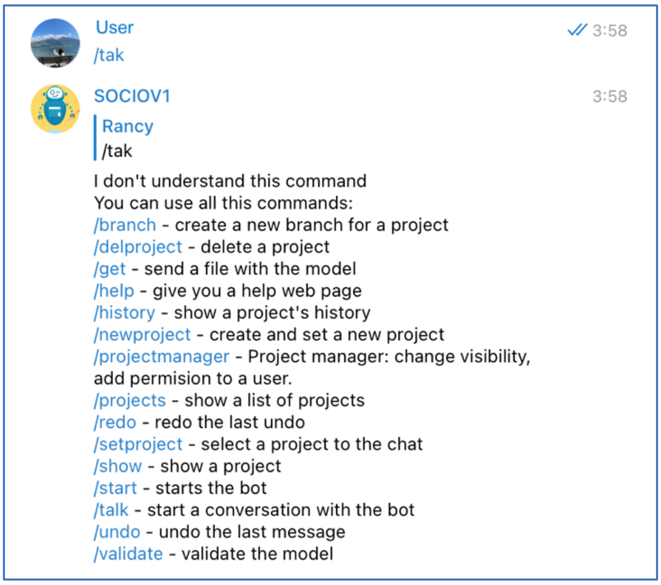
FIGURE 2. **After the first modification of Version 1.**

2. When a user's command is not in the correct format, we provide suggestions on how to organize the command correctly. For instance, we change the autoreply from "I don't understand this command" to "I don't understand this command. You can use all these commands: + command list" to remind users of the commands they can use (see Figs. 3 and 4).


User ✓✓ 3:57
/tak

SOCIO 3:57
Rancy
/tak
I don't understand this command

FIGURE 3. **Before the second modification of Version 1.**


User ✓✓ 3:58
/tak

SOCIOV1 3:58
Rancy
/tak
I don't understand this command
You can use all this commands:
/branch - create a new branch for a project
/delproject - delete a project
/get - send a file with the model
/help - give you a help web page
/history - show a project's history
/newproject - create and set a new project
/projectmanager - Project manager: change visibility, add permision to a user.
/projects - show a list of projects
/redo - redo the last undo
/setproject - select a project to the chat
/show - show a project
/start - starts the bot
/talk - start a conversation with the bot
/undo - undo the last message
/validate - validate the model

FIGURE 4. **After the second modification of Version 1.**

In this article, we adopt the updated version 1 to conduct the second family of experiments with the aim of improving SOCIO chatbot's usability. Because Creately is a commercial product, it has undergone significant improvements. For example, the development team has upgraded the user interface, which no longer relies on Adobe Flash. We approached the Creately support team to request access to the version of Creately (used in the first family of experiments). However, they could not provide this version since Creately Classic was built on Adobe Flash, which is no longer supported by Adobe. Consequently, we used the updated Creately in the second family of experiments.

III. RELATED WORK

Ren et al. conducted a secondary study on chatbot usability experimentation [17]. They found that more and more chatbots had been evaluated with respect to various aspects, ranging from usability to practicability (or quality of outcome). We found that many chatbots had been evaluated through experimentation. However, most of the findings were based on observations from isolated experiments, and results have seldom been evaluated over again, irrespective of whether or not the chatbots were updated and improved. The second study reported that only one out of the 28 retrieved chatbot experiments [18] measured an improved version of the chatbot compared to the original version. The researchers designed a voice-activated chatbot that requires wake-up words. To get early feedback on the usability and the nature of any potential flaws, they conducted the first experiment with the first bot prototype that employed a simple heuristic to assess whether the user was addressing the bot. Following the enhancements to the early version of the bot, they conducted a second experiment with eight novices. These findings provide fair confidence that the second (improved) prototype bot is more useable. However, the researchers conducted experiments with different designs. In other words, to the best of our knowledge, most experiments on chatbot usability either have not been reproduced or have been reproduced according to the lesson they learned from the previous experiment.

It is pretty challenging to verify whether the results of independent experiments arise by chance, whether they are artificial, or whether the results conform to the regularities of the portion of reality under examination [5]. An effective validation method is to replicate the experiment to check that the results are reproducible [5]—this elucidated importance of replication in ESE.

A group of at least three replications could form a family of experiments to provide reliable validation [10]. Basili et al. [9] used the term family of experiments in 1999 to refer to a group of experiments pursuing the same goals whose results can be combined. Santos et al. further distinguishes the family of experiments through collections of experiments, either systematic literature reviews or replications of experiments [10]. Compared to individual experiments, Basili et al. [9] and Santos et al. [10] pointed out that a successful family of experiments has the advantage of increasing the validity and reliability of the outcomes of a single experiment.

However, we have not found any family or replication of experiments on chatbots following improvements. We regard this as being necessary in ESE in order to explore how to improve the usability of chatbots based on evidence. Therefore, we conducted a second family of experiments, reported in this article, with the improved version of the chatbot to explore how the usability of the chatbot was improved based on evidence.

IV. FAMILY DESIGN

This section describes the design of our family of experiments.

A. OBJECTIVES, HYPOTHESES AND VARIABLES

Based on findings from the previous study [15], we set out to investigate through replication within this family of experiments how to improve the usability of a chatbot by including usability characteristics in the application.

Note that our aim was to identify the application of usability characteristics in chatbot development rather than to help teams build a better UML diagram in academic settings. The null hypotheses that govern this research question are as follows:

H.1.0 There is no significant difference in efficiency using SOCIO V1 or improved Creately when building the class diagram.

H.2.0 There is no significant difference in effectiveness using SOCIO V1 or improved Creately when building the class diagram.

H.3.0 There is no significant difference in satisfaction using SOCIO V1 or improved Creately when building the class diagram.

H.4.0 There is no significant difference in the quality of the class diagram built using SOCIO V1 or improved Creately.

As mentioned above, we developed an updated version of the chatbot SOCIO called SOCIO V1 with context-sensitive help, and the control tool Creately was equipped with a better interface that did not rely on Adobe Flash. SOCIO V1 and the improved Creately were used to perform the family of experiments.

For each experiment run, the independent variable was the modeling tool, and the chatbot SOCIO V1 and the improved online application Creately were treatments. According to the above experimental setting [15], the response variables (dependent variables) within this family were three usability characteristics (i.e., efficiency, effectiveness, and satisfaction) and the quality of the outcome.

Based on definitions from ISO/IEC 25010:2011 [4], ISO 9241-11 [19], ISO/IEC/IEEE 29148 [20] and Hornbæk's guidelines [21], efficiency, effectiveness, and satisfaction are commonly measured characteristics for evaluating software usability. Precisely, we measure usability as follows:

Efficiency. Efficiency is measured in terms of time to complete a task and fluency.

Time. Once we completed the tutorial for the tool, participants were sent the task, and time was counted as of when the task was received. We manually recorded how many minutes each team took to complete each task. We recorded the start and stop times for remote experiments via Telegram chat. For offline, face-to-face experiments, we recorded when we asked participants to start on-site and when each team finished. Each team was given a maximum of 30 minutes to complete a task. If a team finished the task early, the time at which they finally submitted the outcome was recorded as the task completion time.

Fluency. Fluency was measured by the number of discussion messages generated by teammates. We counted the number of discussion messages manually. Discussion messages are generally about task performance, tool use, and team management topics. Any irrelevant communication or discussion messages were not counted, e.g., emotional expressions and questions put to the experimenter. Of the discussion messages, SOCIO V1 and Creately both share a common type of discussion message: messages regarding how to use the tool. To gain a better understanding of user opinions, we also analyzed this type of message in the experimental results of the discussion messages afterwards.

We measured *effectiveness* as *completeness*, based on the perceived success of each class diagram compared with the ideal class diagram (see lab package) that we (i.e., the experimenters) built to measure the solutions produced by teams [21], [22].

To calculate the completeness score, we counted how many elements were included compared to the ideal class diagram. We counted each class, relationship and attribute as one element. For instance, the ideal class diagram for Task 1 contains 32 elements. We counted the number of elements included by the teams and divided this number by the ideal number of elements (32) to calculate the completeness score for each team completing each task. Thus, the highest score for each team is 1. Note that when counting the included elements, the name and characteristic of the element does not necessarily have to be absolutely correct. At this point, we are measuring whether the participant managed to create the element, e.g., both "college" and " university" are counted as being correct.

We tailored the *System Usability Scale (SUS) questionnaire* to our experiments to assess *satisfaction* quantitatively and qualitatively. Each questionnaire included 10 five-point Likert scale SUS questions (1 for “Strongly Disagree” and 5 for “Strongly Agree”) and three to four open-ended questions about positive comments, negative comments, and tool suggestions. At the end of the second experimental session, we asked about participants' preferences for either of the two tools.

To calculate the numerical value of each participant's satisfaction score, we used Brooke's equation [23] below to calculate the quantitative SUS result. The team score was calculated using the median of the scores of the three team members for each question:

$$SUS\ score = [\textstyle\sum_{n=1}^{5}(P_{2n-1} - 1) + (5 - P_{2n})] \times 2.5. \qquad (1)$$

We also measured the *quality of the outcome* as the quality of the class diagrams generated by the teams used as a measure of effectiveness [21].

To gauge the quality of each team's class diagram, we used an ideal class diagram as a benchmark. However, a class diagram can have more than one solution, all of which are “correct.” Software engineering experts designed the ideal class diagram before the experiment was carried out. To assess quality, we employed the following metrics [24]:

Precision = TP / (TP + FP) (2)

Recall = TP / (TP + FN) (3)

Accuracy = (TP + TN) / (TP + TN + FP + FN) (4)

Error = (FP + FN) / (TP+ TN + FP + FN) (5)

Success = TP / (#Number of ideal class diagram elements) (6)

By comparing the ideal class diagram with the true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN) for each class diagram, the following formulas were computed:

TP (true positive): Number of elements found in the ideal and team class diagrams.

FN (false negative): Number of elements found in the ideal class diagram but not in the team class diagram.

FP (false positive): Number of elements found in the team class diagram but not in the ideal class diagram.

TN (true negative): There are no true negatives in the model comparison; hence, the value is always 0.

B. DESIGN OF THE EXPERIMENTS

A baseline experiment (EXP1) and two replications (EXP2 and EXP3) form the family of experiments in academic settings. Considering the relatively small sample size of the baseline experiment (i.e., 15 subjects) and the resulting potential for inaccurate and/or biased results [25], we followed the theoretical guidelines set out by Juristo and Gómez [5], employing an identical experimental design for all three experiments. Note that the experimental process of this second family is identical to the first family [15] in order to compare SOCIO and SOCIO V1 vertically. Each of the three experiments was structured as a two-sequence and two-period within-subject crossover design (see Table I).

TABLE I
EXPERIMENTAL DESIGN

Group	Task 1 Period 1	Task 2 Period 2
Group1	SOCIO V1	Creately
Group2	Creately	SOCIO V1

The two replications adhere to the baseline experiment with few variations. To assure that the replications are similar, and the results are comparable, researchers reuse the same experimental protocol and experimental material employed in the baseline experiment, and the replications are jointly run with the experimenter that conducted the baseline experiment.

Three experiments were run at three different sites. The baseline experiment (EXP1) took place at the *Universidad de las Fuerzas Armadas ESPE Extensión Latacunga* (ESPE-Latacunga) in Ecuador (UNIV-1), the first replication (EXP2) was conducted at the *Universidad Autónoma de Yucatán* (UADY) in Mexico (UNIV-2), and the second replication was run at the *Escuela Politécnica Superior of the Universidad Autónoma de Madrid* (EPS-UAM) in Spain (UNIV-3).

Due to COVID-19 lockdown in Mexico, Ecuador and Spain, test sessions for EXP1 and EXP2 were organized remotely via desktop sharing and video conferencing software. EXP3 was conducted in a face-to-face manner.

As both tools are collaborative, the experiments took place in a groupwork setting. The experiments were conducted using three-member teams, and each team was construed as an experimental subject. In each experiment, participants were randomly assigned to one of two groups (Group 1 or Group 2) and then grouped into three-member teams. Accordingly, each group applied the treatments differently (SOCIO V1-Creately/Creately-SOCIO V1). The experimental design is blocked by the period (i.e., the task).

At the beginning of the experiment, each participant was asked to complete a familiarity questionnaire and a consent form. After a 10-minute introduction to the tool that the participants would be using before each period, they were given a maximum of 30 minutes to complete the task using the tool. Group 1 carried out Task 1 with SOCIO V1 in the first period and Task 2 with Creately in the second (i.e., SOCIO V1-Creately sequence). On the other hand, Group 2 completed Task 1 in the first period using Creately and then completed Task 2 using SOCIO V1 in the second period (i.e., Creately-SOCIO V1 sequence). All participants were asked to complete a modified and validated SUS questionnaire connected with the tool they had just used following the completion of each experimental task (i.e., all participants had to fill in the modified SUS questionnaire twice with respect to two modeling tools). Additionally, participants were asked in the second SUS questionnaire whether they preferred SOCIO V1 or Creately.

Two distinct experimental tasks were designed (each assigned to a different experimental period). The first task was to create a class diagram for an online store that includes product and customer management. The second task was to create a class diagram for a college in order to facilitate the organization of courses and pupils. The complexity of the class diagrams was adapted to the duration of the experimental periods. Throughout the experiment, participants of the same team were only permitted to communicate via Telegram groups. This was done to ensure that all experimental data was captured. From the first family of experiments, we observed that most participants tended to run out of time. This may have affected their task completeness. Considering that (1) subject availability was limited and subject fatigue needed to be avoided and (2) we would not be able to measure the effectiveness variable if all the participants had had the option of completing each task, we did not extend the time limit in the following experimental series.

C. SAMPLE

The participants in our family of experiments were students recruited using the convenience sampling method at UNIV-1, UNIV-2, and UNIV-3. The sample in the family was composed of 96 participants: 45 students at UNIV-1, 27 students at UNIV-2, and 24 students at UNIV-3. All the participants were students completing a BSc in Computer Science degree or Joint BSc in Computer Science and Mathematics. Because SOCIO is a modeling chatbot, users had to be acquainted with UML in order to build the model (class diagram). In view of this, we recruited only students with a background in computer science or related fields to ensure that the participants would be able to complete the modeling tasks. To guarantee that all interactions between team members were conducted via Telegram, we made sure that the students' professors were present to oversee the process. In addition, each teammate was seated separately to make sure that there was no whispering.

For technical and methodological reasons (e.g., system failure, incomplete questionnaires, experiment withdrawals), teams 8 and 14 from EXP1 and 6 from EXP2 did not complete the experiment. The study was, therefore, limited to only 87 participants (see Table II). The sample included 14 women and 73 men who ranged in age from 20-27 (mean 22.54, SD 1.29).

TABLE II
OVERVIEW OF SUBJECTS

EXP	Time	Affiliation	#Participants	Teams
EXP1	Jan 2021	UNIV-1	45	15
EXP2	Jan 2021	UNIV-2	27	9
EXP3	Nov 2021	UNIV-3	24	8

A postal survey was carried out with 87 participants. Table III summarizes the analysis performed on aggregated familiarity results.

TABLE III
FAMILIARITY RESULT

Have you ever used Telegram?	
Yes	93%
No	7%
Have you ever used a chatbot?	
Yes	66%
No	34%
Which social networks do you use regularly? (Multiple choice)	
WhatsApp	98%
Telegram	66%
Twitter	52%
Facebook Messenger	59%
Instagram	78%
Rate your English level	
5	15%
4	24%
3	49%
2	11%
1	0%
Rate your knowledge of class diagrams	
5	5%
4	46%
3	38%
2	7%
1	0%
Rate your knowledge of chatbots	
5	5%
4	10%
3	36%
2	18%
1	26%

Considering that 93% of participants had experience with Telegram and 66% used Telegram regularly, we believe that the use of social networking platforms does not affect chatbot usability. However, 34% of participants had no experience with chatbots, and 26% had little knowledge of chatbots, which could detract from the sensitivity and credibility of the experimental results. In addition, we also asked about the level of English: 88% of the participants believed that they had at least an intermediate level of English. Because the task did not require complex English communication, we believe that their English proficiency was good enough to get the job done.

V. RESULTS AND AGGREGATION OF DATA

To answer the research questions, we provide a quantitative and qualitative description of the nature of this study for data synthesis and analysis.

For quantitative analysis, we performed a global analysis of the whole family of experiments and illustrated the individual experiments. The descriptive statistics and violin plots were used to provide readers or other researchers with a better understanding of the normality of each experimental data item. We analyzed the quantitative family result following Santos et al.'s guidelines [26]. The individual participant data (IPD) meta-analysis approach combined with a three-factor LMM was used to study the effect on the outcomes of multiple factors (e.g., period, treatment, and sequence) [10], [27]. We added a parameter to the LMM to account for differences in outcomes across experiments (i.e., Experiment). We then used the corresponding ANOVA table of the LMM to illustrate the statistical significance of the results.

Finally, we adopted the thematic analysis method [28] to gain further insight into user responses and analyze the qualitative data of the three open-ended questions.

A. QUANTITATIVE ANALYSIS

The following section analyzes each response variable (i.e., efficiency, effectiveness, satisfaction, and quality). We concentrate on their respective metrics (i.e., time and discussion messages for efficiency; completeness for effectiveness; satisfaction for satisfaction; and precision, recall, accuracy, error, and perceived success for quality).

For each metric, we provide: (i) descriptive statistics and violin plots divided by treatment (i.e., SOCIO V1, Creately) and by experiment (i.e., EXP1, EXP2, and EXP3), and (ii) the results of all the experiments pooled using a one-stage IPD meta-analysis and the contrast between treatments across the experiments [29].

1) FIRST VALIDATION FOR H.1.0: EFFICIENCY

Efficiency was measured in terms of time and fluency. Time is the amount of time taken to accomplish the tasks. Fluency refers to the number of discussion messages exchanged between teammates during class diagram development. Figs. 5 and 6 illustrate the violin plot for time and fluency across the experiments. The respective summaries of descriptive statistics are shown in Tables IV and VII, grouped by experiment and treatment.

Time. As shown in Fig. 5, the aggregate task completion time with SOCIO V1 appears to be similar to Creately in EXP1 and slightly less than Creately in EXP2 and EXP3. As the descriptive statistics (Table IV) show, time spent on task performance appears to be similar for both Creately and SOCIO V1. Besides, as shown in the ANOVA table (Table V) and the pairwise contrast between the treatments (Table VI), a negligible –and statistically non-significant– difference in the time was observed between Creately and SOCIO V1 (0.43 minutes). In sum, Creately and SOCIO V1 appear to perform similarly in terms of time.

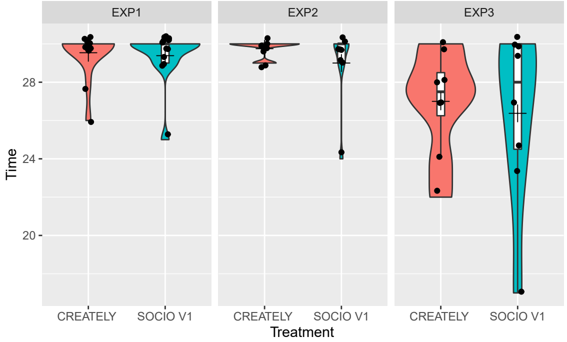


FIGURE 5. Violin plot for time spent on tasks (jitter added to the points).

TABLE IV
DESCRIPTIVE STATISTICS FOR TIME SPENT ON TASKS

EXP	Treatment	Team	Mean	Std. Dev.	Median
EXP1	Creately	13	29.54	1.20	30.0
EXP1	SOCIO V1	13	29.39	1.39	30.0
EXP2	Creately	8	29.75	0.46	30.0
EXP2	SOCIO V1	8	29.00	2.07	30.0
EXP3	Creately	8	27.00	2.78	27.5
EXP3	SOCIO V1	8	26.38	4.60	28.0

TABLE V
ANOVA TABLE FOR TIME

Measure	numDF	denDF	F-value	p-value
(Intercept)	1	27	7110.427	<.0001
Sequence	1	25	3.650	0.0676
Treatment	1	27	0.972	0.3329
Period	1	27	1.219	0.2792
Experiment	2	25	6.349	0.0059

TABLE VI
CONTRAST BETWEEN TREATMENTS FOR TIME

Contrast	Estimate	SE	df	t-radio	p-value
CR-SC	0.431	0.455	27	0.947	0.3519

Interestingly, we identified a trend where most teams in EXP1 and EXP2 spent as long as possible on completing and/or improving their class diagrams. Accordingly, we observed relatively lower task completeness than for EXP3. Based on these observations, a possible conclusion is that these participants needed more time to accomplish the task.

Discussion Messages. Bear in mind that people have different messaging styles: some prefer to send a variety of short messages in succession, and others prefer to send long messages. As mentioned in [15], we counted each sentence containing the complete subject, predicate, and object as one discussion message.

As the violin plot (Fig. 6) and descriptive statistics (Table VII) show, the participants appear to send more messages with

Creately than with SOCIO V1 in two out of our three experiments (EXP1 and EXP3). The opposite holds for EXP2. As ANOVA (Table VIII) and the contrast table (Table IX) show, a negligible –and statistically non-significant– difference in the discussion message was observed between SOCIO V1 and Creately (5.64). This suggests that **SOCIO V1 and Creately appear to perform similarly in terms of discussion messages.**

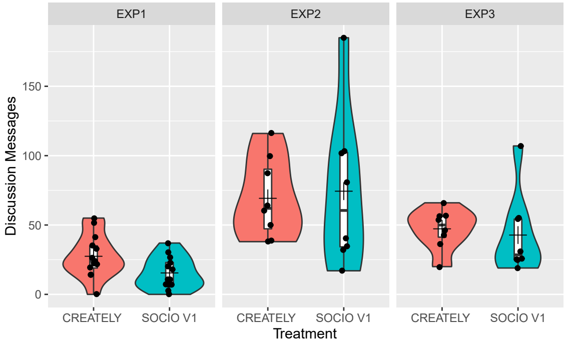


FIGURE 6. Violin plot for discussion messages (jitter added to the points).

TABLE VII
DESCRIPTIVE STATISTICS FOR DISCUSSION MESSAGES

EXP	Treatment	Team	Mean	Std. Dev.	Median
EXP1	Creately	13	27.46	15.53	24.0
EXP1	SOCIO V1	13	15.46	11.33	11.0
EXP2	Creately	8	69.25	28.84	62.0
EXP2	SOCIO V1	8	74.38	55.66	60.5
EXP3	Creately	8	47.25	14.43	50.0
EXP3	SOCIO V1	8	42.75	29.32	28.5

TABLE VIII
ANOVA TABLE FOR DISCUSSION MESSAGES

Measure	numDF	denDF	F-value	p-value
(Intercept)	1	27	86.05308	<.0001
Sequence	1	25	0.54658	0.4666
Treatment	1	27	1.43370	0.2416
Period	1	27	8.35061	0.0075
Experiment	2	25	10.85980	0.0004

TABLE IX
CONTRAST BETWEEN TREATMENTS FOR DISCUSSION MESSAGES

Contrast	Estimate	SE	df	t-radio	p-value
CR-SC	5.64	4.35	27	1.296	0.2058

2) SECOND VALIDATION FOR H.1.0: EFFICIENCY-TOOL USAGE MESSAGES

In the knowledge that there was a wide range of discussion messages, they were classified into the following types: task performance (e.g., how to divide labor), tool use, and discussions about UML knowledge. However, as we were researching chatbot usability, we were interested in discussions on tool usage, that is, how to use the tools properly. Therefore, we extracted discussion messages of this type and then performed an additional analysis.

As the plot (Fig. 7) and the descriptive statistics (Table X) show, the participants are more likely to send more messages on proper tool use with Creately than with SOCIO V1. Besides, as shown in the ANOVA table (Table XI), the difference between the number of tool usage messages is statistically significant (p-value <0.05). According to the pairwise contrast between the treatments in Table XII, **the participants using Creately sent up to 6.38 more tool usage messages than SOCIO V1 users**.

In sum, we cannot reject the null hypothesis H.1.0. SOCIO V1 and Creately appear to perform similarly regarding time and discussion messages. However, SOCIO V1 has the advantage of reducing the communication effort on tool usage for the participants with respect to the first experiment [15].

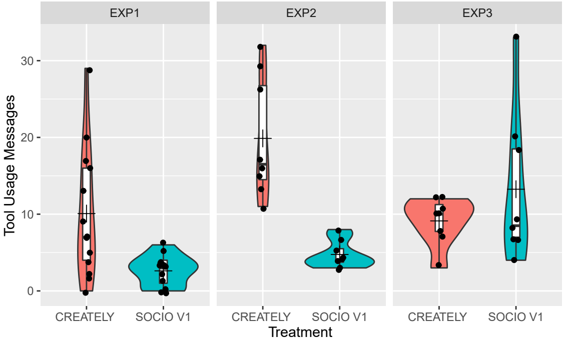


FIGURE 7. Violin plot for tool usage messages (jitter added to the points).

TABLE X
DESCRIPTIVE STATISTICS FOR TOOL USAGE MESSAGES

EXP	Treatment	Team	Mean	Std. Dev.	Median
EXP1	Creately	13	10.08	8.48	7.0
EXP1	SOCIO V1	13	2.62	1.94	3.0
EXP2	Creately	8	19.88	7.94	16.5
EXP2	SOCIO V1	8	4.75	1.83	4.0
EXP3	Creately	8	9.13	3.04	10.0
EXP3	SOCIO V1	8	13.25	9.77	8.5

TABLE XI
ANOVA TABLE FOR TOOL USAGE MESSAGES

Measure	numDF	denDF	F-value	p-value
(Intercept)	1	27	91.89526	<.0001
Sequence	1	25	0.03364	0.8560
Treatment	1	27	10.74951	0.0029
Period	1	27	0.00359	0.9527
Experiment	2	25	3.93138	0.0328

TABLE XII
CONTRAST BETWEEN TREATMENTS FOR TOOL USAGE MESSAGES

Contrast	Estimate	SE	df	t-radio	p-value
CR-SC	6.38	1.95	27	3.279	0.0029

The results of the first family [15] show that SOCIO is significantly more efficient than Creately. We acknowledge that this is mainly due to the fact that the previous version of Creately relied on Adobe Flash, which caused the software to be unstable and resulted in many participants having to quit and re-enter (this was also confirmed by the qualitative analysis, with many participants complaining about this). We noticed that the current version of Creately is no longer dependent on Adobe Flash, and we believe that, based on the experimental data, this has effectively improved Creately's efficiency.

3) VALIDATION FOR H.2.0: EFFECTIVENESS

Completeness. We measured effectiveness by the degree of task completeness.

As the violin plot (Fig. 8), descriptive statistics table (Table XIII), and contrast table (Table XV) show, SOCIO V1 has a slight (0.0752) edge over Creately in terms of completeness. As we can see in the ANOVA table (Table XIV), the treatment has a statistically significant impact on completeness. In sum, **SOCIO V1 outperforms Creately with respect to effectiveness**.

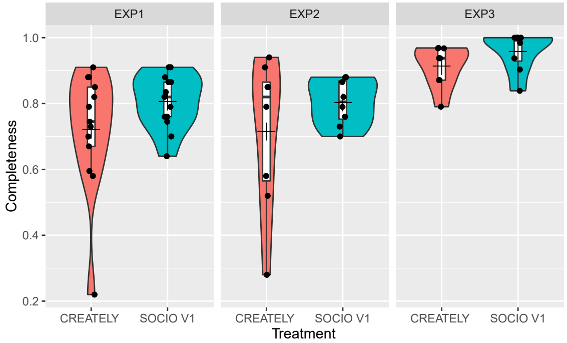


FIGURE 8. Violin plot for completeness (jitter added to the points).

TABLE XIII
DESCRIPTIVE STATISTICS FOR COMPLETENESS

EXP	Treatment	Team	Mean	Std. Dev.	Median
EXP1	Creately	13	0.72	0.18	0.75
EXP1	SOCIO V1	13	0.81	0.08	0.82
EXP2	Creately	8	0.72	0.23	0.82
EXP2	SOCIO V1	8	0.80	0.07	0.81
EXP3	Creately	8	0.91	0.064	0.94
EXP3	SOCIO V1	8	0.96	0.06	0.99

TABLE XIV
ANOVA TABLE FOR COMPLETENESS

Measure	numDF	denDF	F-value	p-value
(Intercept)	1	27	2028.5640	<.0001
Sequence	1	25	0.0000	0.9970
Treatment	1	27	4.4784	0.0437
Period	1	27	0.1387	0.7125
Experiment	2	25	9.3814	0.0009

TABLE XV
CONTRAST BETWEEN TREATMENTS FOR COMPLETENESS

Contrast	Estimate	SE	df	t-radio	p-value
CR-SC	-0.0752	0.0353	27	-2.128	0.0426

In sum, we reject the null hypothesis H.2.0. Compared with the results of the first family [15], completeness has improved by 7.52%, and this improvement is relevant for chatbot usability. After adding context-sensitive help to SOCIO, we observed that SOCIO V1 outperformed Creately on completeness.

4) VALIDATION FOR H.3.0: SATISFACTION

We adopted a modified SUS questionnaire to assess user satisfaction with SOCIO V1 and Creately. Each questionnaire consists of 10 SUS questions and three to four open-ended questions. In this section, we report a quantitative analysis of the responses to the SUS questions. The analysis of the responses to the open-ended questions will be reported in the qualitative analysis section.

Satisfaction Score. Fig. 9 shows the violin plot for the mean SUS scores across experiments. The respective summary of descriptive statistics is shown in Table XVI, grouped by experiment and treatment.

As the violin plot (Fig. 9) and descriptive statistics table (Table XVI) show, the satisfaction scores for SOCIO V1 are typically higher than for Creately. Besides, as the ANOVA table (Table XVII) shows, the difference between the satisfaction scores is statistically significant. In sum, SOCIO V1 appeared to consistently satisfy participants more than Creately in the second family and widened the gap in satisfaction from 6.16 to 8.9 (see Table XVIII). In sum, we rejected the null hypothesis H.3.0. Compared to the first family, an improvement of 8.9 in the second family indicates that satisfaction has improved by 8.9%, and this improvement is worthwhile from the point of view of chatbot usability. Additionally, the satisfaction score of the second family is statistically significant.

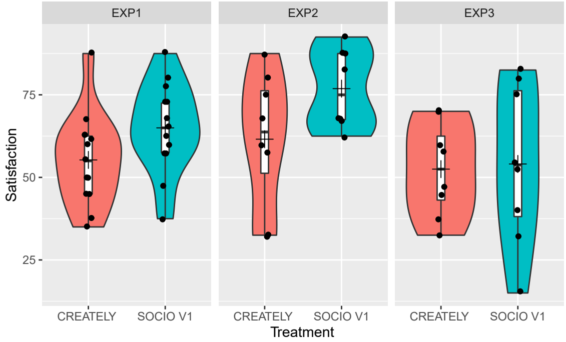


FIGURE 9. Violin plot for satisfaction (jitter added to the points).

TABLE XVI
DESCRIPTIVE STATISTICS FOR SATISFACTION

EXP	Treatment	Team	Mean	Std. Dev.	Median
EXP1	Creately	13	55.29	13.98	55.00
EXP1	SOCIO V1	13	65.00	13.58	65.00
EXP2	Creately	8	61.56	20.48	63.75
EXP2	SOCIO V1	8	76.88	11.78	75.00
EXP3	Creately	8	52.50	14.14	52.50
EXP3	SOCIO V1	8	54.06	24.24	53.75

TABLE XVII
ANOVA TABLE FOR SATISFACTION

Measure	numDF	denDF	F-value	p-value
(Intercept)	1	27	802.2736	<.0001
Sequence	1	25	1.3088	0.2634
Treatment	1	27	4.4109	0.0452
Period	1	27	0.4925	0.4888
Experiment	2	25	3.8526	0.0348

TABLE XVIII
CONTRAST BETWEEN TREATMENTS FOR SATISFACTION

Contrast	Estimate	SE	df	t-radio	p-value
CR-SC	-8.9	4.29	27	-2.075	0.0477

5) VALIDATION FOR H.4.0: QUALITY

We analyzed the quality of the class diagrams using five metrics (cf. equations (2) - (6)): precision, recall, accuracy, error, and perceived success.

The violin plots for these metrics are shown in Figs. 10, 11, 12, 13, and 14, respectively. The respective summary of descriptive statistics is shown in Table XIX, grouped by metric, experiment, and treatment. The summaries of the ANOVA test and contrast between treatments are shown in Tables XX and XXI, respectively.

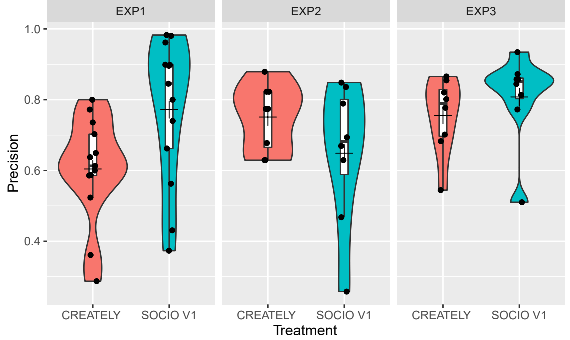


FIGURE 10. Violin plot for precision (jitter added to the points).

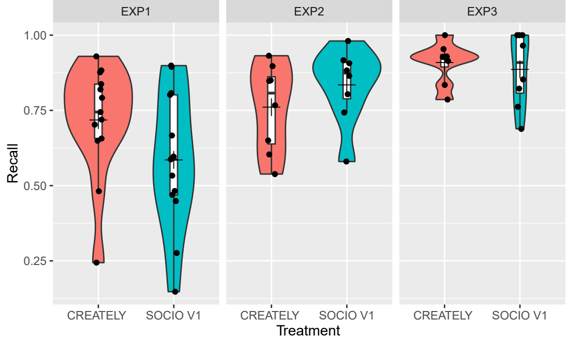


FIGURE 11. Violin plot for recall (jitter added to the points).

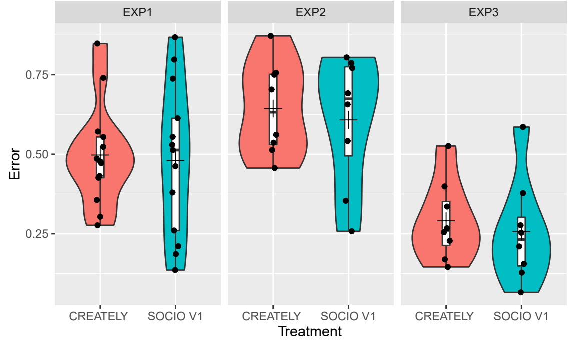


FIGURE 12. Violin plot for error (jitter added to the points).

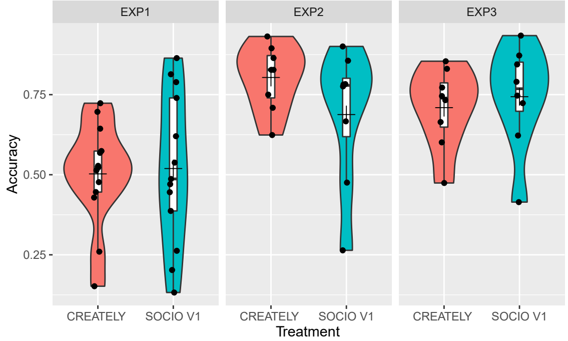


FIGURE 13. Violin plot for accuracy (jitter added to the points).

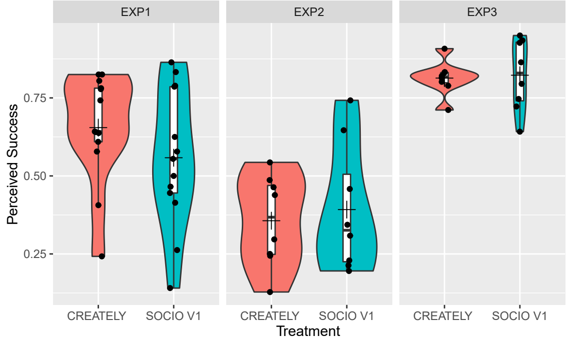


FIGURE 14. Violin plot for perceived success (jitter added to the points).

TABLE XIX
DESCRIPTIVE STATISTICS FOR QUALITY

Variable	EXP	Treatment	Team	Mean	Std. Dev.	Median
Precision	EXP1	Creately	13	0.60	0.15	0.61
	EXP1	SOCIO V1	13	0.77	0.21	0.85
	EXP2	Creately	8	0.75	0.09	0.77
	EXP2	SOCIO V1	8	0.65	0.20	0.68
	EXP3	Creately	8	0.76	0.11	0.79
	EXP3	SOCIO V1	8	0.81	0.13	0.85
Recall	EXP1	Creately	13	0.72	0.19	0.74
	EXP1	SOCIO V1	13	0.59	0.23	0.59
	EXP2	Creately	8	0.76	0.15	0.81
	EXP2	SOCIO V1	8	0.83	0.13	0.87
	EXP3	Creately	8	0.91	0.07	0.93
	EXP3	SOCIO V1	8	0.89	0.12	0.91
Accuracy	EXP1	Creately	13	0.50	0.16	0.52
	EXP1	SOCIO V1	13	0.52	0.24	0.49
	EXP2	Creately	8	0.80	0.10	0.83
	EXP2	SOCIO V1	8	0.69	0.22	0.78
	EXP3	Creately	8	0.71	0.13	0.74
	EXP3	SOCIO V1	8	0.74	0.16	0.77
Error	EXP1	Creately	13	0.50	0.16	0.48
	EXP1	SOCIO V1	13	0.48	0.24	0.51
	EXP2	Creately	8	0.64	0.15	0.63
	EXP2	SOCIO V1	8	0.61	0.21	0.67
	EXP3	Creately	8	0.29	0.13	0.26
	EXP3	SOCIO V1	8	0.26	0.16	0.23
Perceived Success	EXP1	Creately	13	0.65	0.17	0.64
	EXP1	SOCIO V1	13	0.56	0.22	0.55
	EXP2	Creately	8	0.36	0.15	0.37
	EXP2	SOCIO V1	8	0.39	0.21	0.33
	EXP3	Creately	8	0.81	0.06	0.82
	EXP3	SOCIO V1	8	0.82	0.11	0.83

TABLE XX
ANOVA TABLE FOR QUALITY

Variable	Measure	numDF	denDF	F-value	p-value
Precision	(Intercept)	1	27	1333.4514	<.0001
	Sequence	1	25	0.3530	0.5578
	Treatment	1	27	0.4316	0.1306
	Period	1	27	14.6171	0.0007
	Experiment	2	25	2.1285	0.1401
Recall	(Intercept)	1	27	1546.8766	<.0001
	Sequence	1	25	0.0675	0.7972
	Treatment	1	27	1.3758	0.2511
	Period	1	27	19.5103	0.0001
	Experiment	2	25	14.6336	0.0001
Accuracy	(Intercept)	1	27	907.8200	<.0001
	Sequence	1	25	0.0727	0.7897
	Treatment	1	27	0.1277	0.7236
	Period	1	27	15.7263	0.0005
	Experiment	2	25	14.2395	0.0001
Error	(Intercept)	1	27	468.7157	<.0001
	Sequence	1	25	0.0006	0.9813
	Treatment	1	27	0.3846	0.5403
	Period	1	27	12.3023	0.0016
	Experiment	2	25	1807748	<.0001
Perceived Success	(Intercept)	1	27	814.8571	<.0001
	Sequence	1	25	0.0814	0.7778
	Treatment	1	27	0.5399	0.4688
	Period	1	27	8.5950	0.0068
	Experiment	2	25	30.7001	<.0001

TABLE XXI
CONTRAST BETWEEN TREATMENTS FOR QUALITY

Variable	Contrast	Estimate	SE	df	t-radio	p-value
Precision	CR-SC	-0.056	0.0393	27	-1.427	0.1652
Recall	CR-SC	0.0512	0.0387	27	1.325	0.1964
Accuracy	CR-SC	0.0207	0.042	27	0.494	0.6254
Error	CR-SC	0.0215	0.0432	27	0.499	0.6219
Perceived Success	CR-SC	0.0352	0.0421	27	0.835	0.4108

Precision and Perceived Success. Regarding Precision and Perceived Success, the violin plots (Figs. 10 and 14) and descriptive statistics table (Table XX) show that SOCIO V1 slightly outperforms Creately in two out of three experiments.

Recall and Accuracy. Regarding Recall and Accuracy, the violin plots (Fig. 11, 13) and descriptive statistics table (Table XIX) show that Creately slightly outperforms SOCIO V1 in two out of three experiments.

Error. Regarding Error, the violin plots (Fig. 12) and descriptive statistics table (Table XIX) show that SOCIO V1 slightly outperforms Creately across all three experiments.

However, based on the analysis of these five quality metrics, we did not observe any statistically significant treatment. Summing up the above analysis, as the plots, descriptive statistics, ANOVA, and contrast table show, **Creately and SOCIO V1 both tend to return class diagrams of similar quality**.

In sum, we do not reject the null hypothesis H.4.0. In contrast to the result for the first family, which showed that Creately outperformed SOCIO in terms of recall and perceived success and SOCIO outperformed Creately on precision, we did not observe a statistically significant difference in the second family after both tools had been improved.

6) DISCUSSION OF ANALYSIS

From the aggregation of this family of experiments result, we observed that participants seemed to have higher task completeness with SOCIO V1 compared to Creately, and they appeared to be more satisfied with SOCIO V1 than Creately. However, the two tools perform similarly in terms of efficiency and quality of class diagrams.

B. QUALITATIVE ANALYSIS —THEMATIC ANALYSIS

We enacted the thematic analysis process as follows. After each experiment session, the participants were asked to complete a modified SUS questionnaire containing three or four open-ended questions regarding (i) three positive aspects, (ii) three negative aspects, (iii) three suggestions concerning the tool they had just used, and (iv) their preference for either tool (response required only after the second session). The response to open-ended questions was transcribed into English. Due to the need to identify recurring themes to identify interesting aspects, we coded features that were mentioned more than three times in the qualitative dataset.

As shown in Figs. 15 and 16, we identified six features based on satisfaction measures for SOCIO V1 and Creately [21]: content, task, collaboration, communication, user experience, and interface. We expected these results to contribute to the development of future real-time collaboration tools, particularly chatbots, and improve user-perceived usability. Figs. 17 and 18 are bar graphs that illustrate thematic analysis sub-themes, providing a more simplified and readable analysis. The orange bars represent user suggestions for the tool, the gray bars indicate negative comments, and the blue bars are positive comments.

In general, both tools received a similar number of reviews for each of the three open-ended questions. For example, SOCIO V1 received 245 positive comments, and Creately received 244. SOCIO V1 received 198 negative comments and Creately received 177. SOCIO V1 received 72 suggestions and Creately received 63.

1) CONTENTS

SOCIO V1 outperforms Creately in terms of contents, given that it receives more positive comments (26 vs. 16), fewer criticisms (7 vs. 23), and no suggestions for improvement, whereas Creately receives 3.

Users consider both SOCIO V1 and Creately to be helpful for integrated content and design implementation purposes (e.g., "useful for creating class diagrams", "useful tool for UML"). Moreover, this feature is more prominent in SOCIO V1, as it is mentioned by almost twice as many users than for Creately (11 vs. 6).

Content errors appear to be the most commonly reported faults with respect to content. Twice as many Creately users as SOCIO V1 users report errors (13 vs. 7), although they do not suggest respective improvements.

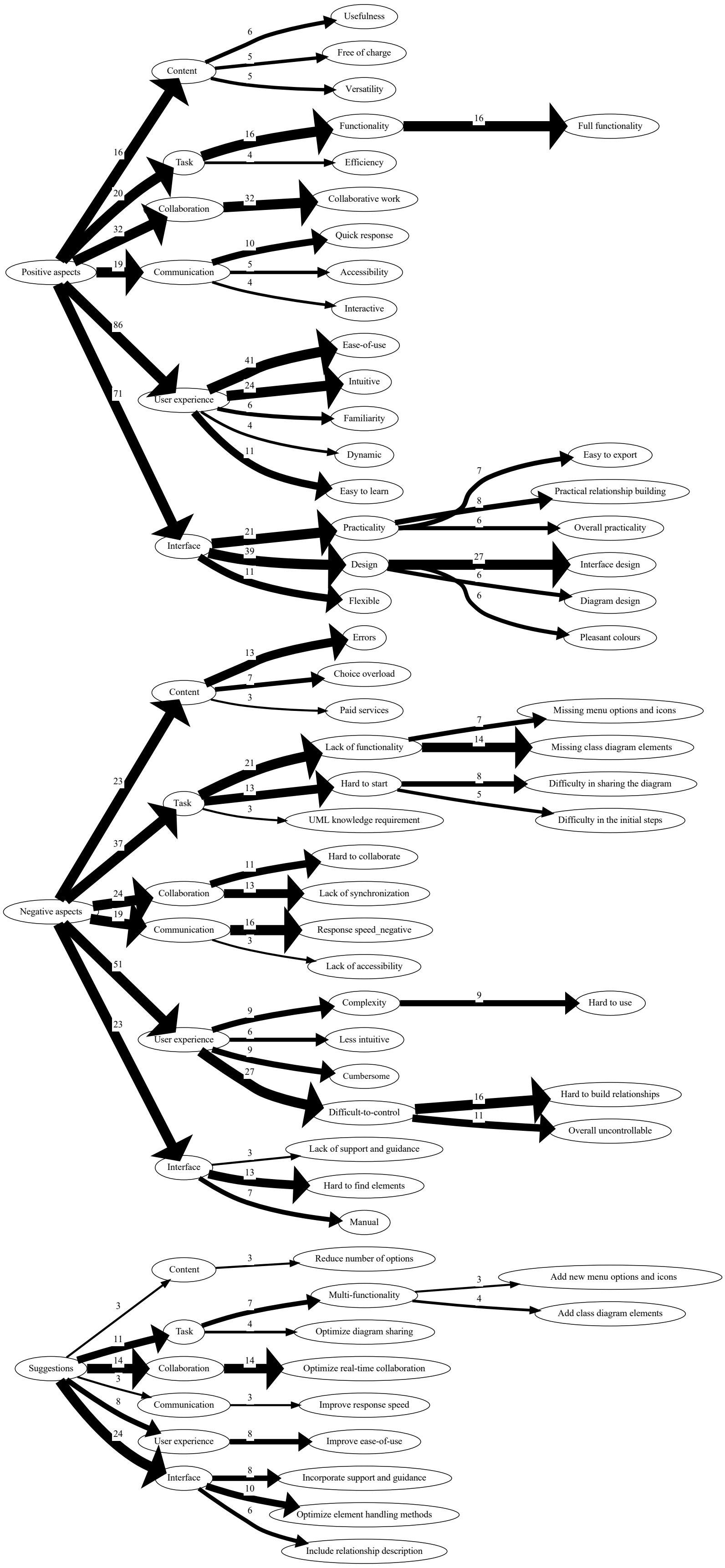


FIGURE 15. **Thematic analysis for Creately.**

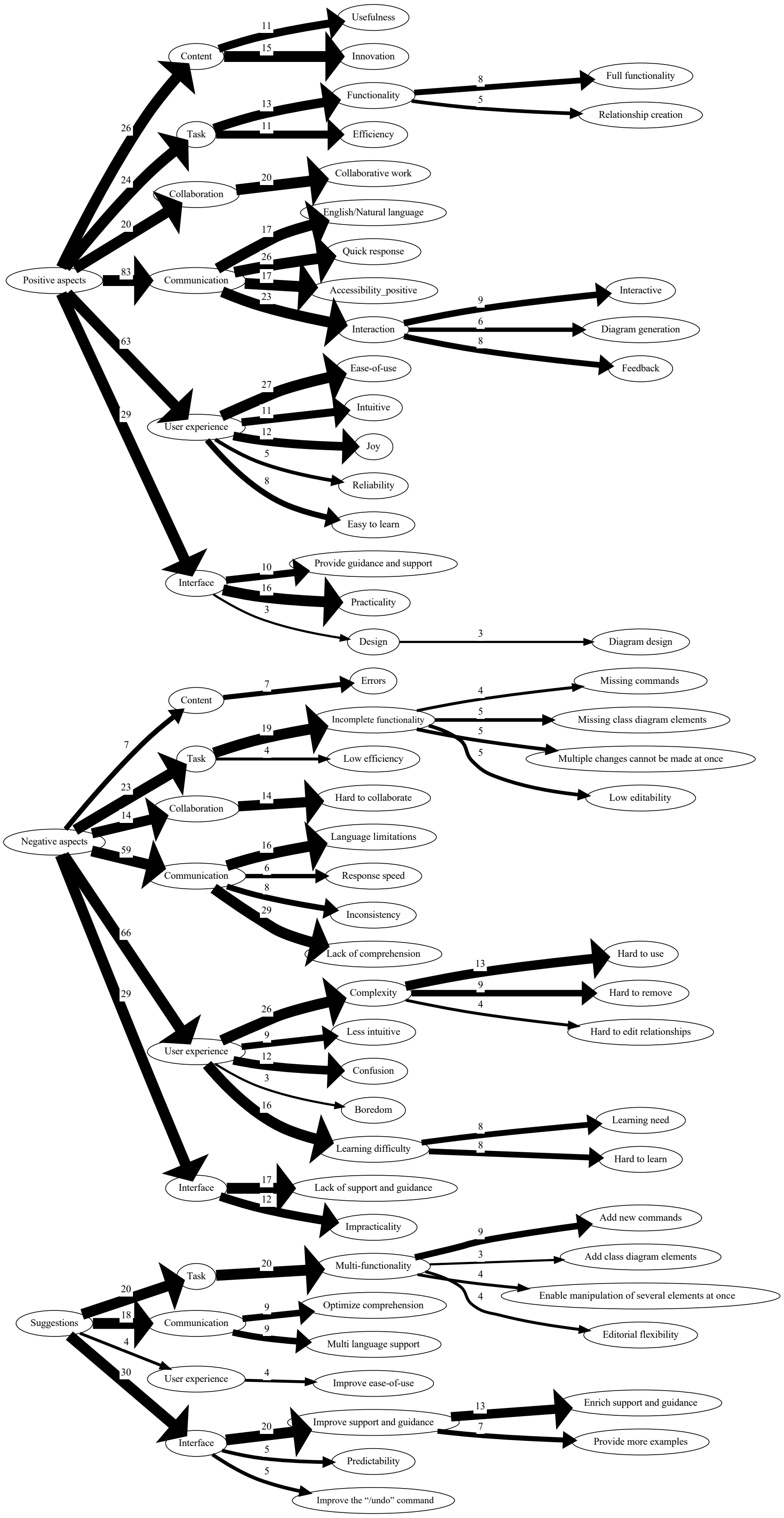


FIGURE 16. Thematic analysis for SOCIO.

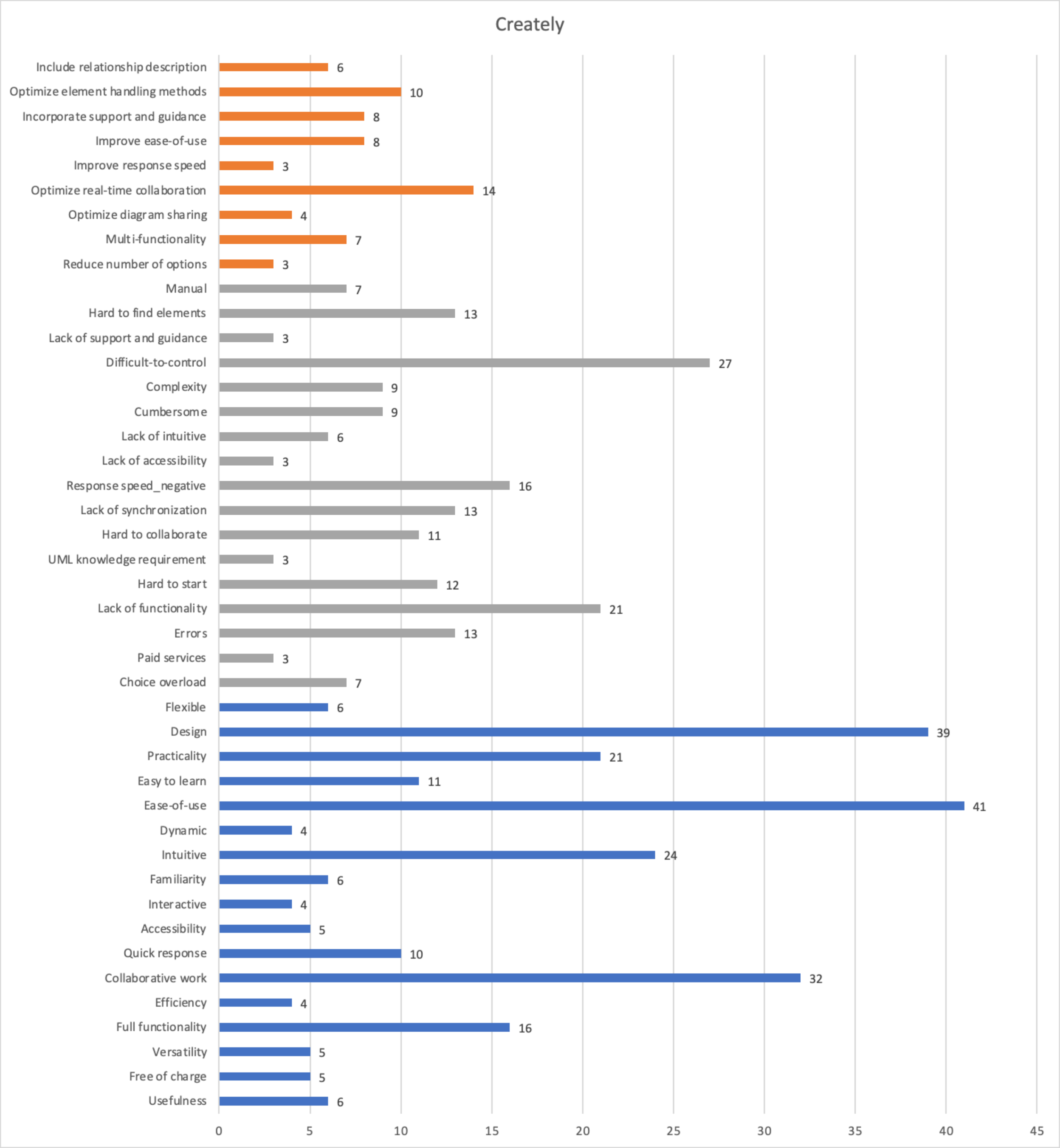


FIGURE 17. Bar graph of Creately's thematic analysis.

These bugs are mainly related to server and page response errors in both cases.

Apart from usefulness and errors, the remaining aspects described as positive and negative differ for the two tools. SOCIO V1 appears to be positively rated on innovation (15), as the experimental subjects regard a chatbot for building class diagrams as innovative and surprising (e.g., “innovation in creating UML”). By contrast, Creately's content design for commercial purposes caused controversy. On versatility (5), which provides additional content for UML diagram creation and is free of charge (5), it stands out slightly compared to other tools with similar features that offer paid services. In contrast, other users also consider these features to be a weakness.

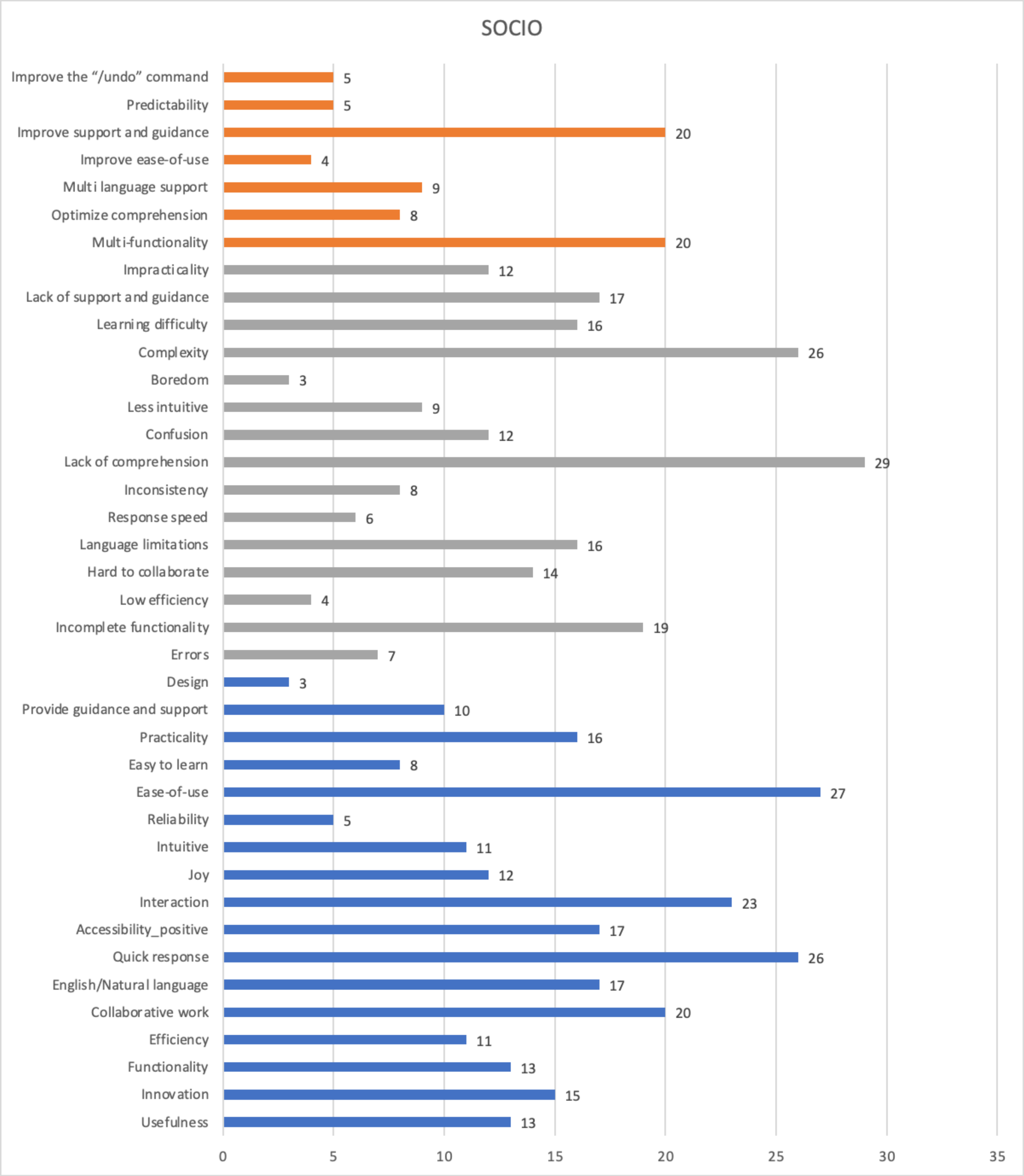


FIGURE 18. **Bar graph of SOCIO's thematic analysis.**

On the one hand, seven people indicate that it offers too many options that are not used for elaborating UML diagrams, and three participants even suggest reducing these options. On the other hand, three participants were also dissatisfied with the fact that, although the main functionality is free, it includes some paid features (3).

2) TASK

SOCIO V1 receives conflicting feedback on task completion, with 24 favorable and 23 negative comments, respectively. By contrast, Creately earns more negative comments (37) than positive ones (20). Compared to Creately, SOCIO V1 receives more positive feedback on task completion.

Regarding task completion, participants are most concerned with the tool's functionality and efficiency since they were mentioned most often. In terms of time efficiency, SOCIO V1 outperforms Creately, with 11 participants stating that the development of UML diagrams does not take very long (e.g., "Creating the diagram is extremely fast," "It works fast"), compared to only four comments for Creately.

However, when it came to evaluating the completeness of the tool functionality for task performance, both tools were found to have functional flaws in terms of missing class diagram elements and missing actions that need to be performed. Participants have mixed feelings about both tools; some believed the functionality was complete, while others reported flaws and suggested adding new functions.

Missing functionality was noteworthy in SOCIO V1. Whereas eight participants praised its comprehensive functionality, and five participants liked the fact that SOCIO V1 automatically and simply generates the relationships in the UML diagram ("Establishing relationships is easy," "Automatically creates links between classes"), 19 participants indicated that they missed functionality, and 20 suggested that new functionalities should be added. They mentioned, for instance, that (i) it is hard to edit class diagram elements (e.g., "You cannot modify class names," "Little modification of the diagram"), (ii) it is not possible to operate many diagram pieces at once (e.g., "I believe you cannot create several things at the same time," "It does not place the attributes in a group way, it places them one by one"), and (iii) data types are missing (e.g., "Limited attribute types," "No Singleton option"). Opinions vary widely with respect to functionality completeness in Creately, with 16 participants praising its functionality and 21 participants expressing dissatisfaction with missing functionality. For example, the participants were dissatisfied with missing relationships to link classes (e.g., "UML connector types," "Not all UML associations"). Also, 13 Creately users had trouble signing up, logging in, and creating the document, especially when sharing the project (8).

3) COLLABORATION

The collaboration feature refers to real-time collaboration, and both tools garnered more positive than negative feedback overall. Surprisingly, on the one hand, Creately's cooperation performance was complimented by more users (32) than SOCIO V1's (20). On the other hand, Creately earned more negative feedback in terms of collaborative capacity (24 vs. 14).

Both tools received a similar number of positive assessments for supporting real-time collaboration. However, Creately received 12 more positive reviews than SOCIO V1. We can conclude that SOCIO V1 outperforms Creately in terms of collaboration. Both tools have garnered criticism for being difficult to work with. Creately received 11 complaints (e.g., "It's challenging to cooperate", "It's confusing to work with numerous individuals"), whereas SOCIO V1 received 13 (e.g., "(It's) tough to utilize in teams," "It's confusing to work in groups"). Furthermore, we observed that Creately had a specific synchronization flaw, as 13 participants found it difficult to keep up with modifications made by their teammates (e.g., "Sometimes it takes a while to synchronize," "There is a little delay when collaborating"). Based on the above, Creately received 14 suggestions on how to improve cooperation, such as integrating chat.

4) COMMUNICATION

The interaction between users and tools is referred to as communication. SOCIO V1 receives significantly more positive and negative feedback, and suggestions, than Creately on communication.

Participants provide feedback on three themes common to both tools: response time, accessibility, and interaction. SOCIO V1 outperforms Creately on each of these aspects. Roughly three times as many users praise SOCIO V1 for quick reaction time than Creately (26 vs. 10). Furthermore, only six participants consider SOCIO V1 to have a slow response time as opposed to 16 for Creately. SOCIO V1 is more accessible than Creately as it benefits from being a social media-based tool. SOCIO V1 also outperforms Creately in terms of interaction. Although both tools receive positive feedback, Creately receives four positive comments while SOCIO V1 receives 23.

Of these 23 opinions, six participants appreciated the fact that SOCIO V1 returns the updated diagram after each action (e.g., "Shows the diagram after each command"), while eight highlighted SOCIO V1's help system in response to user errors (e.g., "If you make a mistake in a command, it corrects you instantly," "Provides good feedback").

In addition, the experimental participants expressed positive and negative thoughts on specific chatbot aspects. Seventeen participants positively rated communication with SOCIO V1 through natural language. However, it also received a disproportionately large amount of negative feedback on natural language comprehension:

- 29 participants stated that the chatbot does not understand sentences entered to build the diagram (e.g., "the chatbot sometimes does not understand what I enter," "Limited language").
- 16 participants complained that the chatbot only understands English sentences.
- Eight participants stated that communication with the chatbot is inconsistent because it sometimes responds differently to the same message.

These limitations are highlighted in the improvement suggestions: increase comprehension (9) and provide multi-language support.

5) USER EXPERIENCE

User experience refers to the user attitudes towards the interface and user interface experience. We observed that both tools received a lot of both positive and negative feedback in this regard.

The common sub-themes for both tools are ease of use and intuitiveness. Since it earned more positive and fewer negative comments in this respect, Creately is easier to use and more intuitive than SOCIO V1. On the other hand, SOCIO V1 was rated as more fun to use than Creately by 12 experimental participants, while Creately was rated as cumbersome or unmanageable by nine.

Both tools were praised for their wide-ranging capabilities (86 for Creately and 63 for SOCIO V1). Several people who claimed Creately is easy to use also mentioned that it is standard (6) and easy to understand (11). SOCIO V1 scores high for being fun to use (12), reliable (5), and easy to learn (8).

As already mentioned, both tools received a lot of negative feedback as well. Regarding SOCIO V1:

- Users were primarily disappointed because they found the chatbot confusing to use (12) and that it required a lot of learning (16). Since they were unfamiliar with SOCIO V1, they needed to learn how the chatbot worked (how to interact through commands and natural language sentences). Some users (8) found this taxing ("They must know the commands," "You need to learn every function of every command").

Regarding Creately:

- Some users (27) found the interface control and element management in the window frustrating ("I cannot change the position of the boxes," "The control of the application with the mouse is not easy").
- 16 subjects specifically stated that it is hard to control the elements adding relationships to link classes ("When joining or associating frames the task becomes a bit complicated," "The arrows chose paths overlapping with other elements").

When we asked participants for suggestions on how to improve the user experience for these tools, only eight and four people, respectively, suggested improving the ease of use of Creately and SOCIO V1.

6) INTERFACE

In general, Creately outperforms SOCIO V1 as it received more positive and less negative feedback. In particular, Creately was praised for being more visually appealing than SOCIO V1. Regarding interface design, 39 subjects found Creately's interface appealing, emphasizing that it is "minimalist" and "simple." Six of them specifically positively rated the visual attractiveness of the diagrams, and six emphasized the color range used. For SOCIO V1, however, only three people referred to the diagram's design in a positive light.

The interface of both tools is suitable for developing UML diagrams since SOCIO V1 received 16 favorable comments and Creately received 21. SOCIO V1 was credited for its command usage and automatic organization of diagram elements (e.g., "I like that all class diagram modification actions are done under the same command ((\talk)," "sort everything automatically"). Participants praised Creately in particular for the method of using a line to directly relate classes (8) and the ease with which diagrams can be exported using a button (7) (e.g., "It can be easily exported"). Despite the above, some people identified issues that detract from their usefulness (12 for SOCIO V1 and 7 for Creately). For SOCIO V1, for example, they mentioned (i) the continuous use of the \talk command in Telegram or (ii) the existence of too many commands. Task performance is entirely manual in Creately, which detracts from its practicality (e.g., "Classes and arrows are not reorganized to make it nice," "Everything is written letter by letter").

The number of suggestions for both tools for this issue (30 for SOCIO V1 and 24 for Creately) was greater than for the other five features. Participants suggested that SOCIO V1's assistance and documentation system might be improved and that Creately should incorporate help. SOCIO V1 features a help page and different responses to user input errors, which 10 participants liked, while 17 thought that the documentation was not complete enough (e.g., "A more complete manual is missing"). There were also 20 suggestions for improvement, seven of which refer to the addition of further instances (e.g., "More documentation," "Add more examples of use"). Furthermore, five users recommended introducing predictive support into SOCIO V1 to improve the help system by providing feedback for user input errors (e.g., "Error messages could be improved by including a hint of where the error might be in the message not understood"). On the other hand, Creately does not include any assistance or documentation, and some participants (8) requested that help be included (e.g., "Give a tutorial or walkthrough of any tool").

Because chatbot SOCIO V1 and web-based Creately interface interaction is different, both tools received feedback and suggestions for improving specific aspects of the interfaces. SOCIO V1 uses commands and natural-language statements, whereas Creately adopts drag and drop. With regard to the use of the \undo command in SOCIO, five people recommended that the user be allowed to specify which message to undo. Although 11 people praised Creately for its templates and the ease of diagram customization (e.g., "Several templates available," "Flexible"), 13 people complained about how hard it is to identify elements in Creately (e.g., "The components are not easy to find"). Similarly, 10 participants suggested making it easier to manage the interface elements to overcome the control challenge, and six participants suggested including a description of the relationships as only the name is displayed when they are added.

VI. THREATS TO VALIDITY

Although we considered the question of validity during the experimental design phase to assure the validity of the experiment results, we acknowledge that several threats to validity need to be discussed. In this section, we address the main threats to the validity of our family of experiments according to Cooke and Campbell's guidelines [30].

A. CONCLUSION VALIDITY

The first threat to conclusion validity is the limited sample size (29), which may lead to low statistical power. Although we could not recruit a large enough sample size, we did our best to recruit a sample of diverse participants from different countries and regions with different cultural backgrounds, which contributed to the validity of the results. Random subject heterogeneity rules out risk.

To ensure the transparency of the experimental result and encourage the external replication of the experiments, we uploaded the original data and additional analysis in the supplementary material. In the spirit of open science, we uploaded the experimental data and all the material used in this family of experiments to https://dx.doi.org/10.21227/qzdr-nj48.

B. INTERNAL VALIDITY

On the one hand, we acknowledge that students with similar backgrounds to our family from UNIV-1 and UNIV-3 were also recruited in the first family of experiments. In order to avoid learning effects as well as to alleviate threats to the internal validity, we made sure that the same participant only participated once in either the first or second family of experiments.

On the other hand, recognizing that subjects may react differently as time passes, we limited the duration of each session to 30 minutes. We set a 10-minute break between sessions to prevent subject and experimenter fatigue or boredom.

C. CONTRUCT VALIDITY

The first limitation that we observed is English language proficiency. Through the familiarity questionnaire, participants self-assessed their English language level with mean scores of 2.64, 3.11, and 3.98 for EXP1, EXP2, and EXP3, respectively. We observed that they did not express much confidence in their English level, and when they communicated with the chatbot in English, they also used some Spanish words (Spanish is their native language). However, the SOCIO V1 chatbot only supports natural language communication in English; participants had to use English to perform the task. On the other hand, Creately does not require a lot of English communication as it is a visual tool. This may threaten the quality of communication and the experiment results. To reduce this threat, we updated the help pages in Spanish (the native language of the subjects) and translated our materials and questionnaires into Spanish to reduce the communication effort.

The second limitation to construct validity was social threats. As mentioned before, we were forced to conduct two out of the three experiments remotely (i.e., EXP1 and EXP2) due to the COVID-19 pandemic. The remote experiment may prevent experimenters from solving misunderstandings timely. For example, two members of team 14 in EXP1 experienced network problems at the beginning of the second session of the experiment. They joined the experiment 11 minutes late. This meant that only one team member was working on task performance for the first 11 minutes. This invalidated the participation of this team, as this incident affected the experiment results.

D. EXTERNAL VALIDITY

Threats to external validity may materialize due to the use of students as experimental subjects and the adoption of toy tasks. As is common in SE experiments [6], we employed toy tasks and student subjects to measure the performance of two treatments. In addition, due to the characteristics of chatbots (using UML language), our participants had to be students of computer science or related fields. Although most of the subjects participating in the experiment were final-year computer science students and could be considered representative of novices in industry, the results of the study are applicable to an academic setting and may not be generalizable to industry.

VII. DISCUSSION OF RESULTS

Regarding the results on effectiveness and efficiency, it appears that 62.0% of the subjects tended to take as long as possible to complete and/or improve their class diagrams, while 38.0% of the subjects completed their class diagrams in as short a time as possible, i.e., they completed the task before the 30-minute time limit was up.

Of the abovementioned subjects, 62.0% completed the class diagram for the task that they were set close to the 30-minute time limit. If they had been given longer, they would have used up the allotted time. In this case, the average time taken would have been longer. However, we decided to set a time limit to be able to measure other variables, such as task completion rate.

As both tools were upgraded to varying degrees, neither of the treatments are the same as in the first family [15]. Although comparisons in data terms are meaningless, some differences should be pointed out. Creately's impressive improvements in terms of efficiency and precision referred to quality are due to a significant change: the operating environment upgrade, which no longer relies on Adobe Flash, has resulted in faster page refresh times, more efficient teamwork, and a resulting increase in the effectiveness of class diagram drawings. The changes to SOCIO mainly helped our users to create diagrams, which resulted in the improvement of satisfaction and completeness.

VIII. CONCLUSION AND FUTURE WORK

On the one hand, based on experimental results from previous work [15], (1) we updated the help page for SOCIO by providing more than one language and more examples, (2) we provided alternative context-sensitive help when SOCIO had difficulties understanding the command that the user sent. On the other hand, the control tool Creately was also upgraded replacing Adobe Flash and including an improved interface.

To understand how to improve the usability of chatbots based on evidence, we conducted a family of three experiments with a within-subject crossover experimental design. A total of 87 participants were recruited and divided into 29 teams. Finally, we reported pooled results, as well as quantitative and qualitative analyses.

In conclusion, we reject the null hypotheses H.2.0 and H.3.0, but not the null hypotheses H.1.0 and H.4.0.

The main results of the analysis of the data gathered in the family of experiments reveal that:

1. *With the observed quantitative results:* SOCIO V1 has better scores for effectiveness and satisfaction than updated Creately. Regarding the efficiency and quality of the class diagram, the difference between the two treatments at family level was not statistically significant.
2. *With the summary of the qualitative results:* SOCIO V1 appears to receive more positive comments and fewer criticisms than Creately regarding contents and interface. Regarding collaboration and communication, both treatments garner more positive than negative feedback. Both treatments receive conflicting feedback on task completion and user experience, with similar numbers of favorable and unfavorable comments.

This family of experiments consolidates the body of knowledge about chatbot usability improvement built on the results of the experiments. We hope our work will provide insights and different perspectives on usability evaluation for SOCIO chatbot and Creately developers.

In this research, we have found that some improvements to be implemented would be: (1) make it easier to remove/edit the elements, and (2) improve the natural language ability.

In the future, we will develop a second and third updated version (see background) to better understand how the usability of chatbots can be improved based on evidence.

ACKNOWLEDGMENT

This research was funded by the Spanish Ministry of Science, Innovation and Universities research grant PGC2018-097265-B-I00 and the FINESSE project (PID2021-122270OB-I00). This research was also supported by the Madrid Region R&D program (FORTE project, P2018/TCS-4314) and the SATORI-UAM project (TED2021-129381B-C21).

APPENDIX A

Apart from common changes, we created three different versions of the updated the SOCIO chatbot. Here we provide details of the changes that we made to versions 2 and 3.

Updated Version 2 (SOCIO V2): With added functionalities requested by users.

1. Add a /remove command. In response to the first suggestion on modifying the /undo command to be able to undo a participant's action, we developed a new command /remove. After sending the command, users can choose the type of elements (classes, attributes, or relations) that they want to remove. The full list of elements appears, and the user can select the exact element to be removed. For instance, if a user opts to remove a relationship, the chatbot lists all the relationships in the diagram, the user selects a relationship, and the relationship is automatically deleted.
2. Add two commands /undo+ and /redo+. The user can choose how many steps to cancel or redo instead of deleting or redoing one by one. Technically, there is no need to implement anything new inside SOCIO, we merely have to add commands to Telegram with a loop that performs undo or redo as many times as requested by the user. For instance, if a novice user realizes that there is something wrong with the elements he just created, instead of deleting them one by one and creating a new project, he merely has to use the /undo+ command to delete as many steps as he wants.

Updated Version 3 (SOCIO V3): Interface preference. In order to better understand if the change of appearance affects SOCIO chatbot usability, we changed the appearance of generated class diagram using a smaller font; for example, we set the monochrome font on a black background to six.

REFERENCES

[1] M. Franzago, D. Di Ruscio, I. Malavolta, and H. Muccini, “Collaborative model-driven software engineering: A classification framework and a research map,” *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1146-1175, 2017.

[2] T. Sutikno, L. Handayani, D. Stiawan, M. A. Riyadi, and I. M. I. Subroto, “Whatsapp, viber and telegram: Which is the best for instant messaging?” *International Journal of Electrical & Computer Engineering (IJECE)*, vol. 6, no. 3, pp. 909-914, 2016.

[3] 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. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE’17)*, Urbana, IL, USA, 2017, pp. 723-728.

[4] *Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*,” ISO/IEC 25010, 2011.

[5] N. Juristo, and O. S. Gómez, *Replication of Software Engineering Experiments*, Springer Berlin/Heidelberg, 2012, p. 60-88.

[6] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, Springer Science & Business Media, 2012.

[7] W. F. Tichy, “Should computer scientists experiment more?” *Computer*, vol. 31, no. 5, pp. 32-40, 1998.

[8] D. T. Lykken, “Statistical significance in psychological research,” *Psychological Bulletin*, vol. 70, no. 3, pp. 151-159, 1968.

[9] V. R. Basili, F. Shull, and F. Lanubile, “Building knowledge through families of experiments,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456-473, 1999.

[10] A. Santos, O. Gómez, and N. Juristo, “Analyzing families of experiments in SE: A systematic mapping study,” *IEEE Transactions on Software Engineering*, vol. 46, no. 5, pp. 566-583, 2018.

[11] K. Chung, H. Y. Cho, and J. Y. Park, “A chatbot for perinatal women’s and partners’ obstetric and mental health care: Development and

usability evaluation study," *JMIR Medical Informatics*, vol. 9, no. 3, article e18607, pp. 1-17, 2021.

[12] L. S. Nowell, J. M. Norris, D. E. White, and N. J. Moules, "Thematic analysis: Striving to meet the trustworthiness criteria," *International Journal of Qualitative Methods*, vol. 16, no. 1, pp. 1-13, 2017.

[13] M. Javadi, and K. Zarea, "Understanding thematic analysis and its pitfall," *Journal of Client Care*, vol. 1, no. 1, pp. 34-40, 2016.

[14] V. Braun, and V. Clarke, Thematic analysis. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), *APA handbook of research methods in psychology, Vol. 2. Research designs: Quantitative, qualitative, neuropsychological, and biological* (pp. 57-71). American Psychological Association, 2012.

[15] R. Ren, J. W. Castro, A. Santos, O. Dieste, and S. T. Acuna, "Using the socio chatbot for UML modelling: A family of experiments," *IEEE Transactions on Software Engineering*, PrePrints, 2022. Doi Bookmark: 10.1109/TSE.2022.3150720.

[16] M. Reeves, and J. Zhu, "Moomba – A collaborative environment for supporting distributed extreme programming in global software development," in: Eckstein, J., Baumeister, H. (eds) *Extreme Programming and Agile Processes in Software Engineering* (XP 2004 pp. 38-50). Lecture Notes in Computer Science, vol 3092. Springer, Berlin, Heidelberg.

[17] R. Ren, M. Zapata, J. W. Castro, O. Dieste, and S. T. Acuña, "Experimentation for chatbot usability evaluation: A secondary study," *IEEE Access*, vol. 10, pp. 12430-12464, 2022.

[18] R. R. Divekar, J. O. Kephart, X. Mou, L. Chen, and H. Su, "You talkin'to me? A practical attention-aware embodied agent," in: Lamas, D., Loizides, F., Nacke, L., Petrie, H., Winckler, M., Zaphiris, P. (eds) Human-Computer Interaction (INTERACT 2019 pp. 760-780). Lecture Notes in Computer Science, vol 11748. Springer, Cham.

[19] *Ergonomics of Human-System Interaction - Part 11: Usability: Definitions and Concepts*," ISO 9241-11, 2018.

[20] *Systems and Software Engineering – Life Cycle Processes – Requirements Engineering*," ISO/IEC/IEEE 29148, 2018.

[21] K. Hornbæk, "Current practice in measuring usability: Challenges to usability studies and research," *International Journal of Human-Computer Studies*, vol. 64, no. 2, pp. 79-102, 2006.

[22] R. Ren, J. W. Castro, A. Santos, S. Pérez-Soler, S. T. Acuña, and J. de Lara, "Collaborative modelling: Chatbots or on-line tools? An experimental study," in: *Proc. Evaluation and Assessment in Software Engineering (EASE'20)*, Trondheim, Norway, 2020, pp. 260-269.

[23] A. Iovine, F. Narducci, M. de Gemmis, and G. Semeraro, "Humanoid robots and conversational recommender systems: A preliminary study," in: *Proc. 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS'20)*, Bari, Italy, 2020, pp. 1-7.

[24] T. Fergencs, and F. Meier, "Engagement and usability of conversational search–A study of a medical resource center chatbot," in: Toeppe, K., Yan, H., Chu, S.K.W. (eds) *Diversity, Divergence, Dialogue* (iConference 2021 pp. 328-345). Lecture Notes in Computer Science, vol 12645. Springer, Cham.

[25] R. D. Riley, P. C. Lambert, and G. Abo-Zaid, "Meta-analysis of individual participant data: Rationale, conduct, and reporting," *BMJ*, vol. 340, pp. 1-7, 2010.

[26] A. Santos, S. Vegas, M. Oivo, and N. Juristo, A procedure and guidelines for analyzing groups of software engineering replications," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1742-1763, 2019.

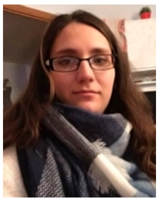
[27] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120-135, 2015.

[28] V. Braun, and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77-101, 2006.

[29] A. Whitehead, *Meta-Analysis of Controlled Clinical Trials*, 1st Edition, John Wiley & Sons, 2002.

[30] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*, Houghton Mifflin Boston, 1979.

RANCI REN received her MS in ICT Research and Innovation from the Universidad Autónoma de Madrid (UAM), Spain, in 2019. She is currently working toward her PhD in Software Engineering at UAM. Her main research interests include experimental software engineering, human-computer interaction, and chatbots. She is a member of the ACM.



SARA PÉREZ-SOLER received her MS in ICT Research and Innovation from the Universidad Autónoma de Madrid (UAM), Spain, in 2018. She is currently working toward her PhD in Software Engineering at UAM. She is currently assistant professor at Universidad Autónoma de Madrid's Computer Science Department. Her main research interests include chatbots, domain-specific language and model-driven engineering.

JOHN W. CASTRO received his PhD from the Universidad Autónoma de Madrid in 2015. He received his MS in Computer Science and Telecommunications, specializing in Advanced Software Development, from the Universidad Autónoma de Madrid in 2009. He has fifteen years of experience in the area of software system development. He is currently assistant professor at the University of Atacama (Chile). He worked as a research assistant at the Universidad Politécnica de Madrid. His research interests include software engineering, software development process, and the integration of usability in the software development process.

OSCAR DIESTE received his BS and MS in Computing from the Universidad da Coruña and his PhD from the Universidad de Castilla La Mancha. He is a researcher with the UPM's School of Computer Engineering. He was previously with the University of Colorado at Colorado Springs (as a Fulbright scholar), the Universidad Complutense de Madrid, and Universidad Alfonso X El Sabio. His research interests include empirical software engineering and requirements engineering.

SILVIA T. ACUÑA received her PhD from the Universidad Politécnica de Madrid in 2002. She is currently an associate professor of software engineering at Universidad Autónoma de Madrid's Computer Science Department. Her research interests include experimental software engineering, software usability, software process modeling, and software team building. She co-authored *A Software Process Model Handbook for Incorporating People's Capabilities* (Springer, 2005), and edited *Software Process Modeling* (Springer, 2005) and *New Trends in Software Process Modeling* (World Scientific, 2006). She was deputy conference co-chair on the organizing committee of ICSE 2021. She is a member of the IEEE Computer Society and a member of the ACM.

Appendix B

Full Text of Conference Papers

B.1 The Rise of the (Modelling) Bots: Towards Assisted Modelling via Social Networks

Title	The rise of the (modelling) bots: towards assisted modelling via social networks	
Publication	Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)	
Core	2017 Core A	
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
	Francisco Jurado	Universidad Autónoma de Madrid
Doi	10.1109/ASE.2017.8115683	
Date	October 2017	

Abstract

We are witnessing a rising role of mobile computing and social networks to perform all sorts of tasks. This way, social networks like Twitter or Telegram are used for leisure, and they frequently serve as a discussion media for work-related activities.

In this paper, we propose taking advantage of social networks to enable the collaborative creation of models by groups of users. The process is assisted by modelling bots that orchestrate the collaboration and interpret the users' inputs (in natural language) to incrementally build a (meta-)model. The advantages of this modelling approach include ubiquity of use, automation, assistance, natural user interaction, traceability of design decisions, possibility to incorporate coordination protocols, and seamless integration with the user's normal daily usage of social networks.

We present a prototype implementation called SOCIO, able to work over several social networks like Twitter and Telegram, and a preliminary evaluation showing promising results.

The Rise of the (Modelling) Bots: Towards Assisted Modelling via Social Networks

Sara Pérez-Soler, Esther Guerra, Juan de Lara, Francisco Jurado
Universidad Autónoma de Madrid (Spain)

***Abstract*—We are witnessing a rising role of mobile computing and social networks to perform all sorts of tasks. This way, social networks like Twitter or Telegram are used for leisure, and they frequently serve as a discussion media for work-related activities.**

In this paper, we propose taking advantage of social networks to enable the collaborative creation of models by groups of users. The process is assisted by *modelling bots* that orchestrate the collaboration and interpret the users' inputs (in natural language) to incrementally build a (meta-)model. The advantages of this modelling approach include ubiquity of use, automation, assistance, natural user interaction, traceability of design decisions, possibility to incorporate coordination protocols, and seamless integration with the user's normal daily usage of social networks.

We present a prototype implementation called SOCIO, able to work over several social networks like Twitter and Telegram, and a preliminary evaluation showing promising results.

***Index Terms*—Collaborative modelling; meta-modelling; social networks; natural language processing**

I. INTRODUCTION

According to recent studies [1], 70% of American citizens are users of social networks. People exploit social networks like Twitter, Telegram or Facebook, often on a daily basis, to be in contact with friends, share media, organize leisure activities, or discuss work-related issues in an agile manner.

The use of social networks for Software Engineering has been recognised as having high potential impact in practices and tools [2], [3]. One of the reasons is that they enable agile and lightweight means for coordination and information sharing. However, some identified open challenges include their use to increase community and end-user involvement, enhance project coordination, and improve development activities [3].

In this work, our goal is to profit from the benefits and potential of social networks to assist in a particular Software Engineering task: modelling and meta-modelling. For this purpose, we propose the collaborative modelling through social networks assisted by *modelling bots* that interpret the messages of users in order to construct a model or meta-model. User messages can be expressed either in natural language (NL), or they can be commands for model evolution. In the first case, the bot interprets the messages using NL processing techniques [4], [5] and derives update actions over the current model version. The approach is inherently collaborative, and integrates seamlessly with the normal usage of social networks to which users are familiar with. Moreover, it naturally leads to an accurate documentation of the traceability and provenance of the different design decisions incorporated into the model.

This paper motivates and presents usage scenarios for modelling bots, and presents a working prototype called SOCIO that is able to orchestrate model construction across both Twitter and Telegram. We have performed an initial user evaluation with promising results that encourage pursuing further research in this direction.

The rest of this paper is organized as follows. Section II motivates our approach, identifies envisioned scenarios, and elicits a set of requirements. Section III details the components of our proposal. Section IV describes architecture and tool support. Section V shows the results of our preliminary evaluation. Section VI compares with related work, and finally, Section VII ends with the conclusions and future work.

II. MOTIVATION AND USAGE SCENARIOS

The main motivation for this work is being able to benefit from the collaborative and ubiquitous nature of social networks – applications that people use on a daily basis – to perform assisted lightweight modelling. To this aim, we propose repurposing social networks based on micro-blogging (like Twitter or Telegram) as front-ends for the modelling activity, where dedicated bots interpret certain user messages to assist in the model construction. This way, the assisted modelling process seamlessly integrates with the normal use of social networks for discussion.

This approach enhances flexibility in modelling because it can be used in mobility and does not require installing new applications, but users can rely on apps they are already familiar with. When working on mobile devices, interacting via short messages can be easier and faster than using a diagramming tool, and can serve to quickly prototype models. Moreover, people with little or no background in computer science or modelling may be able to actively participate in modelling sessions. This may foster the collaboration of domain experts with teams of engineers. By recording the messages processed by the bot, the approach can trace information of the design decisions (*who* made *what*), so that every decision can be justified or rolled back.

This approach can be useful in several scenarios. First, to allow engineers quickly prototyping models *when* and *where* needed (e.g., in working meetings, but also when travelling home). Second, to assist teams of engineers collaborating with domain experts (who may lack a computer science background) to create domain models or meta-models. Third, in the educational domain, to enable groups of students the collaborative resolution of modelling exercises. In this scenario, bots could be configured for gamification activities or blended learning. Finally, being based on social networks,

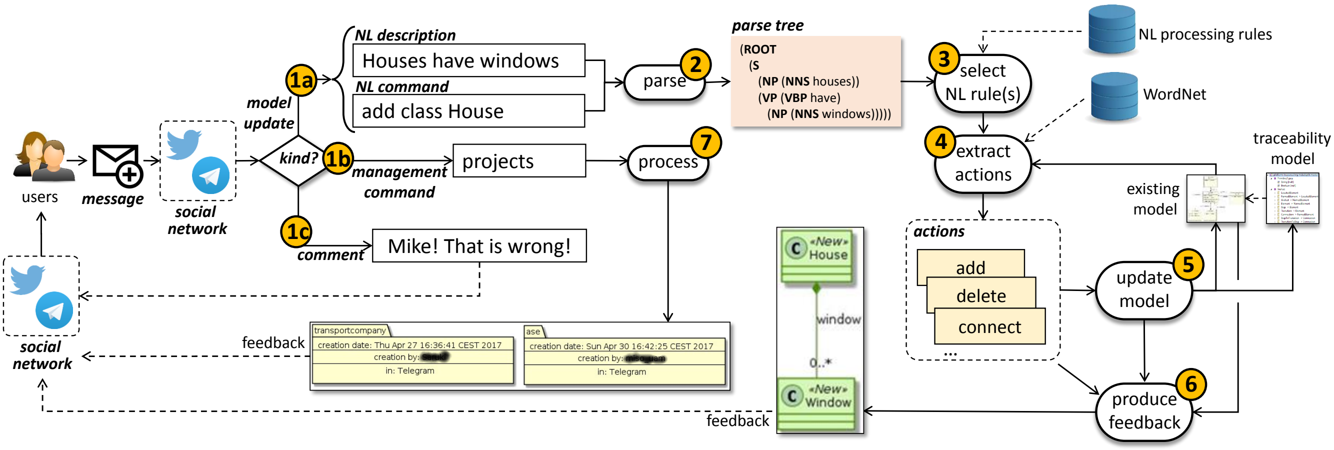


Fig. 1. Overview of our approach to assisted modelling via social networks

the modelling process can involve a large number of people. Hence, we foresee its use to crowdsource modelling decisions.

In order to give support to these scenarios, we identify the following requirements for our envisioned approach:

- Interaction through NL (to permit use by domain experts) and commands (more suitable for modelling experts). Anyhow, commands should have a flexible and natural syntax to minimize mistakes and user frustration.
- Traceability mechanisms able to justify design decisions and find their provenance.
- Integration of multiple social networks, so that users can use their preferred one.
- Support for both modelling and meta-modelling.
- Customizable collaboration protocols, e.g., supporting user roles and voting.
- Interoperability with accepted modelling frameworks, like the Eclipse Modelling Framework (EMF) [6].

Next, we detail our first steps towards realizing this vision.

III. Approach

Fig. 1 sketches our approach to modelling via social networks. Users can interact within the network of choice by sending messages directed either to the other partners (label 1c) or to the modelling bot (labels 1a-1b). The former messages permit discussing and coordinating, and they are handled by the network normally (i.e., they are regular text messages). Instead, messages directed to the bot are used for building models, and their format may depend on the particular social network. For instance, to send a message to the bot in Twitter, the message must mention the username of the bot (@modellingBot), while in Telegram, the message should start by /. The way of organising the collaboration also depends on the particularities of the social network. In Twitter, users of our system will normally be followers of the bot supervising the modelling process, so that every message sent to the bot will be received by its followers. In Telegram, users would create a group with the users interested in the modelling task, including the bot. In this way, every sent message will be received by all members of the group.

Messages directed to the bot are received by the users of the social network, just like any other message, but in addition, they are processed by the bot. We distinguish two kinds of such messages: *management commands* (label 1b) and *model update messages* (label 1a). The former allow performing project management tasks, like querying the existing modelling projects, creating a new model, or recovering the history of changes performed in a model. The bot processes such messages (label 7) and sends the result to the social network as a diagram embedded in an image file. The figure shows an example, where the set of available projects is returned as a package diagram (output of activity labelled 7). On their side, model update messages (label 1a) can be either *commands* or *descriptions*. The former are imperative actions to directly manipulate a model, e.g., to add a class or feature, change its type, or delete an element. The latter are descriptive statements of the domain concepts, like “houses have windows”.

Both commands and descriptions are expressed in NL. Hence, they are processed by a NL parser (label 2) which produces a parse tree of the sentences in the message. Supporting commands in NL provides flexibility because there is no need to adhere to a strict syntax. The system has an extensible library of NL processing rules, able to handle different kinds of NL phrases (label 3); currently, we support rules targeted to meta-model construction. From the current model state and the information inferred from the message, we synthesize a number of model update actions (label 4). As our approach is incremental, sometimes it is necessary to refer to existing model elements. In order to provide flexibility and avoid redundancies, we allow for synonyms which are sought using WordNet [7], a lexical database for the English language. We will describe our NL processing approach in Section III-A, and the extraction of model update actions in Section III-B.

The extracted actions are applied to the current version of the model to make it evolve. Moreover, the system maintains a traceability model to keep track of *why* the model was updated, and by *who* (label 5). We will explain model building and traceability in Section III-C. Finally, the bot emits a feedback message (label 6), which is received through the social network. The feedback to model update messages consists of an image of the updated model, with annotations indicating the last modifications. Section III-D will illustrate some steps in a model construction session, and the feedback obtained.

A. Natural Language Processing

We use the Stanford NL parser [5] to process model update messages, both commands and descriptions. The parser creates a parse tree with the grammatical relations of the words in the message, as Fig. 1 shows for message "houses have windows" (see output of activity labelled 2). This tree identifies the noun phrases (NP) that may provide information about the model, and the syntactical role of each part of the phrase. For example, in the tree of Fig. 1, "houses" and "windows" are tagged as common plural nouns (NNS), while "have" is tagged as a verb in present tense which is non-3rd person singular (VBP).

The parsed messages are interpreted according to a number of rules, which we currently base on the work of Arora and collaborators [4]. Each rule specifies the combination of word classes that activate the rule, usually based on the presence of certain verbs, as well as the model update actions that should be performed when the rule is matched. Our rules currently handle the construction of meta-models only, but since our approach is extensible, we plan to expand the set of rules to tackle the construction of models, likely instances of previously defined meta-models using this same approach. We consider the following rules:

Verb to be: When the main verb of the phrase is "to be", it can indicate either an inheritance relation between two classes (e.g., "Kitchen is a room", "Service may be premium service or normal service"), or the type of a feature (e.g., "Name is string", "The bank of the customer is BLUX"). If the phrase contains an expression of the form "A of B" or a genitive "B's A", then the rule infers that A is an attribute or reference of B.

Verb to have: When the main verb is "to have", or synonyms of it like "characterized by" and "identified by", this rule infers the subject of the sentence is a class, which has a feature. Deciding whether the feature is a reference or an attribute depends on the information there is in the meta-model about the feature. If there is not enough information, the feature is assigned an "open" type which may be refined by subsequent messages. As an example, the message "Bulky packages are characterized by their width, length and height" triggers the creation of features width, length and height with open type in class BulkyPackage.

Transitive verb: This rule handles all verbs with a subject and a direct object. It creates classes for the subject and direct object, and a reference whose name is the verb. For example, the phrase "The simulator shall send log messages" triggers the creation of classes Simulator and LogMessage, and the reference send from the former to the latter.

Contain: Verbs like "contain", "be made of", "include" and "be composed of" imply a composition relation between two classes. For example, the phrase "A delivery is made of packages" creates a composition relation between the classes Delivery and Package, and also creates the classes if they do not exist yet.

Add: This rule handles imperative sentences (with implicit subject) whose verb is "add", "create", "make", etc. These are interpreted as commands with a flexible syntax, resulting in the creation of classes, attributes or references. For example, "add house" will create the class House, while "create room in house" adds a feature room to class House.

Remove: This rule is similar to the previous one but for deletion. It considers imperative verbs synonyms of "remove", like "delete" and "erase".

Model update messages can include several sentences and more than one verb, like in "Add house and remove windows". Moreover, the processing of one message may trigger several NL rules, in which case, we apply the rule with higher priority. In particular, rules seeking for specific verbs have higher priority than the more general rule seeking transitive verbs.

B. Model Update Actions

As abovementioned, each NL rule specifies the model update actions to be applied when the rule is selected. The possible actions are the following:

Add class: This action is issued when the rule finds a common name that should be a class. The class is not created if one exists with the same or synonym name. Supporting synonyms provides flexibility and avoids redundancy. We apply accepted modelling styles for class names (i.e., singular, camel case).

Make class abstract/concrete: Classes can be set to abstract or concrete using their name or a synonym. If the class does not exist, then an *add class* action is issued as well.

Set parent class: This action sets an inheritance relation between two classes, creating the classes if they do not exist.

Remove parent: This action removes an inheritance relation. If the class does not exist, the action will make no changes.

Add attribute: This action is issued by the "verb to have" rule (e.g., "packages have weight") and in case of genitive cases (e.g., "package's name"). The attribute is added to the given class or to a synonymous one, creating a new class if it does not exist. If the class already owns a reference with same name, it is replaced by an attribute. The attribute's upper cardinality is set to 1 if the attribute name is singular, or to * if it is plural. At this point, the attribute type is left open, so that it can be refined later.

Add reference: This action is issued by the "transitive verb" and "contain" rules, and it works similarly to the previous action. If the owner class already defines an attribute with the same name, it is replaced by a reference.

Modify feature type: We support primitive data types like int, float, String, boolean and Date for attributes, while the type of references must be a class. The feature is created if it does not exist.

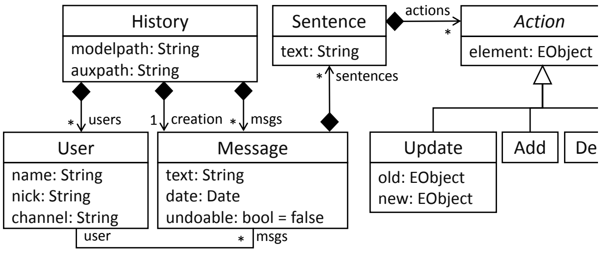


Fig. 2. Traceability meta-model

Remove class: It removes a class and its features.
Remove feature: It removes one or several features.

Any action can be undone and redone through comma

C. Model Update and Traceability

The actions derived from the messages are applied t current model version. In some cases, these actions lack some information. For instance, the action that an attribute may miss the specific type of the attribut which case, its type is left "open" so that it can be re later. Similarly, as removing a class would let the refere pointing to the class dangling, we add a provisional "gl class as target of these references. We foresee an auton mechanism that completes all these "open" design deci with sensible defaults in a final stage.

We also maintain traceability information of each mes sent to the bot, including the sender and the model update actions it triggers. We use a model-based approach to record the traceability data, building a traceability model conformant to the meta-model in Fig. 2 for each model being constructed. Class User keeps track of the participant users and the social network (channel) they employed to send the messages. We store all messages directed to the bot, and distinguish the message used to create the model. Actions point to the model elements affected by the action using reference element, whose type is EObject as this is the base class in EMF, the implementation technology we use. To keep track of the removed elements which are no longer in the model, we store them in an auxiliary model. Finally, Update actions point to the old and new versions of the updated element in the auxiliary model, and to the current version of the element in the model.

An image of the updated model diagram is sent to the collaborator users. The modified model elements are marked in the diagram, and a note explains the performed changes. This information is read from the traceability model. The trace can also be queried using the management command history, which sends a diagram with the traceability information to the users.

D. Example

Fig. 3 illustrates a typical modelling session. The rectangles labelled 1–4 contain NL messages that a user sends, while the diagrams are the feedback provided by the modelling bot.

The first message is handled by the "transitive verb" rule. This creates classes for "good transport company" and "delivery", and a reference for "handle". The cardinality of the reference is many as "deliveries" is in plural. The created classes have singular, camel case names. The newly created elements are shown in green. For space constraints, the figure omits the explanation of changes which the bot also produces as notes in the diagram.

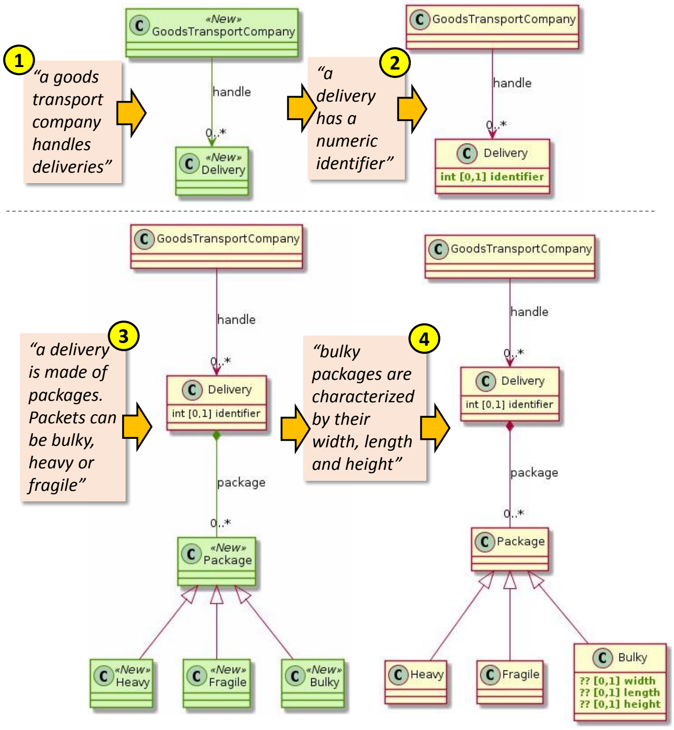


Fig. 3. Some steps in a model construction session

The second message is handled by the "verb to have" rule, which adds an integer attribute to class Delivery. The bot assigns an upper cardinality of 1, as there is no plural.

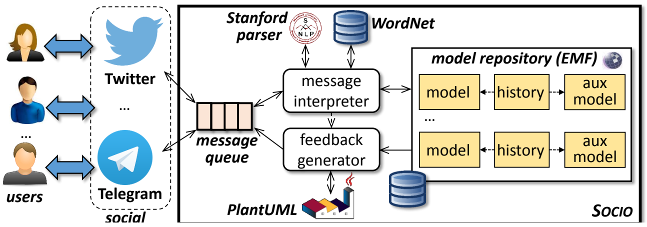
The third message contains two phrases. The first one is handled by the "contain" rule, which creates class Package and reference package. The second one is handled by the "verb to be" rule, which creates the inheritance hierarchy. This phrase makes use of the word "packet", which is as a synonym of "package", and hence, no new class is added for it.

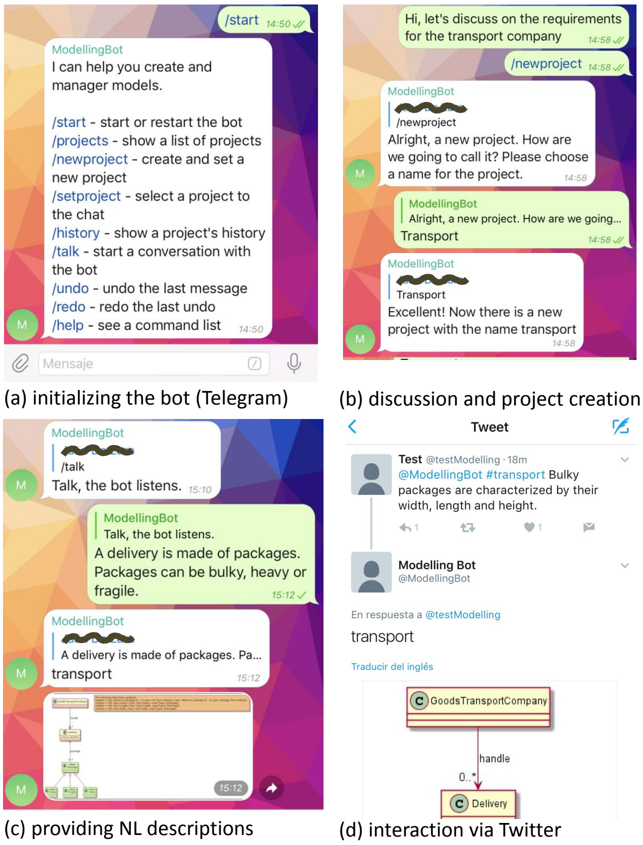
The fourth message, processed by the "verb to have" rule, creates three features in class Bulky. At this point, there is no information on whether they should be attributes or references, and hence, this is left open (shown with "??").

IV. Tool Support

We have developed a prototype tool for our approach called SOCIO (from assisted modelling through social networks). Fig. 4 shows its architecture. The tool supports Twitter and Telegram, though it can be extended with further social networks by implementing an interface. This means that users of different social networks can interact with each other.

Independently of their provenance, all bot-processable messages are enqueued and processed one-by-one. NL processing is performed using the Stanford parser and WordNet. The cre-





/start 14:50

ModellingBot
I can help you create and manager models.

/start - start or restart the bot
/projects - show a list of projects
/newproject - create and set a new project
/setproject - select a project to the chat
/history - show a project's history
/talk - start a conversation with the bot
/undo - undo the last message
/redo - redo the last undo
/help - see a command list 14:50

Mensaje

(a) initializing the bot (Telegram)

Hi, let's discuss on the requirements for the transport company 14:58

/newproject 14:58

ModellingBot
[redacted]
/newproject
Alright, a new project. How are we going to call it? Please choose a name for the project. 14:58

ModellingBot
Alright, a new project. How are we going...
Transport 14:58

ModellingBot
[redacted]
Transport
Excellent! Now there is a new project with the name transport 14:58

(b) discussion and project creation

ModellingBot
[redacted]
/talk
Talk, the bot listens. 15:10

ModellingBot
Talk, the bot listens.
A delivery is made of packages. Packages can be bulky, heavy or fragile. 15:12

ModellingBot
[redacted]
A delivery is made of packages. Pa...
transport 15:12

15:12

(c) providing NL descriptions

Tweet

Test @testModelling · 18m
@ModellingBot #transport Bulky packages are characterized by their width, length and height.
1 1

Modelling Bot
@ModellingBot

En respuesta a @testModelling
transport

Traducir del inglés

(d) interaction via Twitter

Fig. 5. Some steps in a modelling session via Telegram (a-c) and Twitter (d)

ated models are stored using EMF [6], the de-facto modelling standard nowadays. Socio currently supports the collaborative construction of meta-models, but we plan to give support to the creation of instance models in the future. Due to the message length limit of widespread social networks, the feedback is frequently given as a diagram image that may include the current model state, its history, or the set of available projects. These images are generated using PlantUML [8].

To illustrate the tool's capabilities, Fig. 5 shows a sample modelling session over Telegram and Twitter. It reflects the discussion of a set of engineers and the bot for building a meta-model for a transport delivery company.

Steps (a-c) correspond to the interaction in a Telegram group to which the engineers and the bot belong. The bot is started in step (a), and it replies with all available commands. This is a normal message that is received by all members of the group. Step (b) shows one discussion message between the engineers, which is interchanged using the social network normally. It also shows a message directed to the bot, asking the creation of a new model project. In step (c), a NL message describing requirements for the meta-model is directed to the bot. This is performed using the \talk command, after which the bot prompts the user to talk, and the user replies with a NL sentence. The bot processes the sentence, updates the meta-model, and returns an image of it. The image can be enlarged and shared with other Telegram groups and users via email or external services like Dropbox. Step (d) shows the interaction with the bot using Twitter. This requires mentioning the bot account and the model project. Interestingly, in addition to the traceability provided by Socio using the meta-model of Fig. 2, the organization of messages in a life-line which can be browsed at convenience, as provided by Telegram, is also an excellent means to track down design decisions.

V. Evaluation

To assess the suitability of our proposal, we have conducted a preliminary evaluation with 10 participants organized in 4 Telegram groups: 2 groups of 2 people, and 2 groups of 3 people. They were asked to create a meta-model for e-commerce in 15 minutes but with no other restriction, and then complete a questionnaire with 3 parts: two with Likert-scale questions, and a last one with free text questions.

All participants had a computer science background (post-graduate or last year degree students) and were non-native English speakers. The average declared modelling expertise was 62,5% (out of a maximum of 100%), and the level of English was 72,5%. Six conducted the task using the mobile, 2 the web browser, and 2 the desktop Telegram application.

The first part of the questionnaire consisted of the 10 questions of the System Usability Scale (SUS) [9], a de-facto standard to measure system usability. Socio obtained 74%, which indicates good usability. Interestingly, the users that gave the lowest SUS scores were those with less modelling expertise or level of English (the bot must be addressed in English). In particular, the Pearson correlation coefficient was 0,660 with a significance level 0,038.

The second part of the questionnaire comprised 8 questions evaluating four aspects: (1) suitability of NL to build models w.r.t. using an editor, (2) precision of the bot to interpret NL, (3) enough functionality in the command set, and (4) whether they liked embedding a modelling tool in a social network, or they would prefer a separate collaborative tool. We obtained around 75% for aspects 1 and 4, indicating that participants considered NL as a suitable interaction mechanism, and they appreciated the idea of collaborating through social networks. Aspect 3 was rated 60%, which is acceptable but leaves room for enriching the command set. Regarding aspect 2, the statement "bot-generated models agree with the provided NL phrases" was scored 62,5%. However, all participants observed some mismatch between the NL phrases and the obtained models, which suggests the need to improve the precision of the bot to interpret NL. Again, the users assigning a low score were those with less modelling or English expertise.

In the free-text questions, several participants identified as positive the possibility to use the tool on the phone, and being fun, easy-to-use and quicker than other modelling tools. They also suggested some improvements, like the need for coordination mechanisms, and commands to change the reference cardinalities. We will tackle this in future work.

Regarding interaction, participants used more often descriptive NL to interact with the bot (80%) than imperative, command-like messages (20%). This suggests that NL was found useful to complete the task. Overall, 50% were discussion messages and 50% were bot-directed messages. Talking to the bot was balanced in all groups, but in one group where a participant took the role of coordinator and basically sent messages to the other participants. This need for discussion justifies the inclusion of the modelling tool in a social network.

The study is preliminary, with several threats, like the low number of participants, the limited group size, the similar participant background, the fact that participants were non-native speakers, and the lack of a precise modelling goal which permitted evaluating the produced artefacts. However, the positive results encourage further research on this approach.

VI. Related Work

The impact and potential of collaborative and participatory modelling has been recognised by several disciplines – like water resources management and sustainable development [10], or smart product design [11] – to enhance their modelling and decision-making processes. However, typically these works do not propose a concrete method or tool.

In the context of Software Engineering, collaborative modelling has been used for model construction [12] and collaborative creation of domain-specific languages [13]. However, these approaches do not use social networks or NL processing. Instead, they rely on collaborative graphical model editors [12] or ad-hoc tools [13], with no assistant support.

Nowadays, people are used to social networks, and this fact has made organizations to adapt and introduce a social network perspective within their development processes [14], [15]. In this sense, an interesting example of a distributed problem-solving model that combines human and machine computation is crowdsourced software engineering [16]. Many commercial platforms like TopCoder (www.topcoder.com), Bountify (bountify.co) or uTest (www.utest.com) permit recruiting online labour to work on specific tasks, like coding and testing. However, as far as we know, there is no proposal on assisted collaborative modelling over social networks.

Our work proposes using NL to both human collaboration and bot interaction. NL processing techniques have been used within Software Engineering to derive UML diagrams/domain models from text [4], [17]. Our contribution in this context is to use an interactive, incremental approach, and the use of social networks to embed both assistance and collaboration.

Altogether, the use of social networks for collaborative modelling based on NL processing is a novel research direction.

VII. Conclusions and Future Work

In this paper, we have proposed a novel approach to collaborative modelling via social networks, with assistant bots able to process NL messages. We have created prototype tool support working over Telegram and Twitter, and performed an initial evaluation obtaining encouraging results.

In the future, we will incorporate customizable collaboration protocols for different styles of decision making, e.g., based on votings, or roles. We will develop support for building instances of meta-models, and for querying the model elements provenance. We will consider other types of bots, e.g., a quality assurance bot which monitors the current model state to suggest improvements, or gamification bots. We plan to use WordNet's super-subordinate relations to propose inheritance relations, as well as investigate the use of speech recognition for modelling. Finally, we foresee developing scalability mechanisms, e.g., by pruning the bot feedback to return only the modified model elements and their context.

Acknowledgment

Work funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

References

[1] Pew Research Center, http://www.pewinternet.org/fact-sheet/social-media/.

[2] A. Begel, R. DeLine, and T. Zimmermann, "Social media for software engineering," in *Proc. FoSER@FSE*. ACM, 2010, pp. 33–38.

[3] M. D. Storey, C. Treude, A. van Deursen, and L. Cheng, "The impact of social media on software engineering practices and tools," in *Proc. FoSER@FSE*. ACM, 2010, pp. 359–364.

[4] 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.

[5] M. Marneffe, B. Maccartney, and C. Manning, "Generating typed dependency parses from phrase structure parses," in *Proc. LREC*, 2006.

[6] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[7] G. A. Miller, "Wordnet: A lexical database for english," *Comm. ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[8] Plant UML, http://plantuml.com/.

[9] J. Brooke, "SUS: A retrospective," *J. Usability Studies*, vol. 8, no. 2, pp. 29–40, 2013.

[10] M. Hare, "Forms of participatory modelling and its potential for widespread adoption in the water sector," *Environmental Policy and Governance*, vol. 21, no. 6, 2011.

[11] T. Ahram, W. Karwowski, and B. Amaba, "Collaborative systems engineering and social-networking approach to design and modelling of smarter products," *Behav. Inf. Tech.*, vol. 30, no. 1, pp. 13–26, 2011.

[12] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *J. Network and Computer Applications*, vol. 35, no. 3, pp. 1086–1105, 2012.

[13] J. L. C. Izquierdo and J. Cabot, "Collaboro: a collaborative (meta) modeling tool," *PeerJ Computer Science*, vol. 2, p. e84, 2016.

[14] F. O. Zaffar and A. Ghazawneh, "Knowledge sharing and collaboration through social media - the case of IBM," in *Proc. MCIS*, 2012.

[15] C. Manteli, H. Vliet, and B. Hooff, "Adopting a social network perspective in global software development," in *Proc. ICGSE*, 2012, pp. 124–133.

[16] T. D. LaToza and A. van der Hoek, "Crowdsourcing in software engineering: Models, motivations, and challenges," *IEEE Software*, vol. 33, no. 1, pp. 74–80, 2016.

[17] 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*, vol. 22, no. 1, pp. 121–149, 2014.

B.2 Assisted modelling over social networks with SOCIO

Title	**Assisted modelling over social networks with SOCIO**	
Publication	Proceedings of MODELS 2017 Satellite Event co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017)	
Core	2017 Core B	
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Date	September 2017	

Abstract

Social networks are intensively used nowadays for both leisure and work. They have become a natural communication mechanism which helps users in coordinating and collaborating in their daily life activities.

To profit from their pervasive use, we propose SOCIO: a modelling assistant that seamlessly integrates across several social networks, like Telegram or Twitter. SOCIO is a modelling bot that can interpret natural language sentences and create meta-models out of them. It provides traceability of design decisions and statistics of user contributions. A video showcasing the tool is available at https://saraperezsoler.github.io/ModellingBot/.

Assisted modelling over social networks with SOCIO

Sara Pérez-Soler, Esther Guerra, Juan de Lara
Modelling & Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)
e-mail: {sara.perezs, esther.guerra, juan.delara}@uam.es

***Abstract*—Social networks are intensively used nowadays for both leisure and work. They have become a natural communication mechanism which helps users in coordinating and collaborating in their daily life activities.**

To profit from their pervasive use, we propose SOCIO: a modelling assistant that seamlessly integrates across several social networks, like Telegram or Twitter. SOCIO is a modelling bot that can interpret natural language sentences and create meta-models out of them. It provides traceability of design decisions and statistics of user contributions. A video showcasing the tool is available at https://saraperezsoler.github.io/ModellingBot/.

***Index Terms*—Meta-modelling, modelling bots, social networks, natural language processing**

I. INTRODUCTION

Social networks are becoming an important part of our daily digital lives, where we use them to keep in touch with friends and organize leisure activities. They are also gaining relevance as a mechanism to disseminate information and to cooperate and coordinate in work tasks. In particular, social media is increasingly used in software engineering, e.g., to support the formation of ecosystems around particular concepts and technologies; to participate in online development communities; and to disseminate technologies and crowdsource content [1], [2].

Given the widespread use of social networks, our goal is to exploit them for collaborative modelling. Hence, we introduce the concept of *modelling bot*, able to interpret natural language (NL), assist users in creating models, and which integrates with minimum disruption into the natural communication mechanism that micro-blogging based social networks – like Twitter or Telegram – offer. Our approach is lightweight, as it can be used in mobility and it does not require installing any dedicated modelling tool, but users can employ the social network client they are familiar with. It can be used by teams of engineers, likely distributed, to create models and enhance their global coordination. It fosters collaboration with domain experts, who might be unfamiliar with modelling tools, by interpreting domain requirements expressed in NL. Finally, it has the potential to facilitate the crowdsourcing of design decisions. While bots are starting to be used to help in software engineering activities like documentation or user support [3], to our knowledge, ours is the first proposal for a modelling assistant bot.

In this tool demo paper, we describe our modelling bot SOCIO [4] which works across several social networks like Twitter and Telegram. The tool interprets messages from the social network using the Stanford NL parser [5], the WordNet [6] lexical database to identify synonyms, and an extensible set of domain requirements extraction rules [7] that are responsible for growing the model to reflect the messages information. The bot keeps track of all model changes and their provenance, and provides statistics of the participation of each user and their percentage of model authorship. The constructed models can be validated, and exported to the EMF format. While SOCIO has been designed to help in both modelling and meta-modelling, the current version of the tool only supports the creation of meta-models.

The remaining of this paper is organized as follows. First, Section II describes our approach to NL processing, and Section III presents the architecture and main features of SOCIO. Next, Section IV compares with related research. Finally, Section V concludes with prospects for future work.

II. APPROACH

The goal of our tool is to provide modelling assistance integrated within widely used discussion and collaboration mechanisms such as social networks. For this purpose, our modelling bot integrates seamlessly in the social network as another participant, and the interaction with the bot is based on NL to keep a “sense of flow” as there is no need to switch tools for discussing and modelling. Figure 1 shows a scheme of our approach. At any moment, users can send messages directed to the bot. The way to address the bot varies slightly depending on the particularities of the social network: in Twitter, the message needs to mention the username of the bot (@ModellingBot), while in Telegram, the bot must belong to the modelling group and a command starting by “/” needs to be issued. The bot offers a suite of commands for model management (create a new model, show all existing models, update a model, validate a model, download a model) and to obtain statistics. We will explain these commands in Section III, while in the remainder of this section we will focus on the handling of NL messages expressing domain requirements. Messages not directed to the bot are just regular

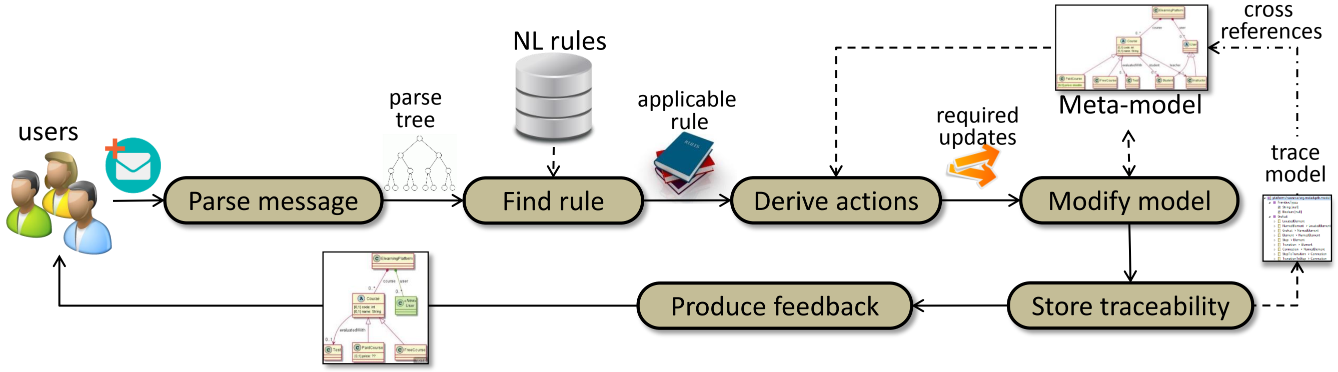


Fig. 1: Processing a NL message for model update

messages used for discussion and coordination within the modelling session.

When the bot receives a message, it uses the Stanford parser to produce a parse tree. Each word in this tree is tagged with its role in the sentence, like noun (singular or plural) or verb (past, gerund, past participle, 3rd person singular present, non-3rd person singular present). We have created a set of extraction rules that detect certain phrase structures of interest in the tree and deduce domain requirements that trigger model updates. The extraction rules are based on the work by Arora and colleagues [7], adapted to our context. The rule set is extensible, and we currently consider the following ones:

- **Verb to be**. These phrases may indicate a subclassing relation (e.g., "users can be students or teachers"), the type of a feature (e.g., "price is double") or the abstractness of a class (e.g., "course is abstract").
- **Verb to have (and synonyms)**. These phrases identify features (attributes or references) of a class. We also consider the Saxon genitive.
- **Transitive verbs**. When a phrase with a verb, subject and direct object is identified, this rule triggers the creation of classes for the subject and direct object (if they do not exist yet), and a reference for the verb.
- **Contain (and synonyms)**. These phrases signal the need for a composition relation between two classes (e.g., "an e-learning platform is made of courses").
- **Add (and synonyms)**. Imperative phrases using "add" or a synonym verb result in the creation of classes or features (e.g., "add name to user").
- **Remove (and synonyms)**. Similar to the previous rule, but for removing elements.

Each rule has a priority, equal to its position in the previous list. In this way, if several rules are applicable, only the one with the highest priority is selected. Then, the selected rule checks the state of the meta-model and produces the necessary meta-model update actions. In this step, rules make use of WordNet to detect synonyms, and take into account grammatical number to allow flexibility when referring to existing classes (e.g., an existing class "Teacher" can be referred later as "Instructors"). The actions are generated explicitly so that their effects can be undone and redone. Supported actions include creating and removing classes, making them abstract or concrete, adding and deleting features, and changing the type of a feature. After processing a message, the updated meta-model may lack some information, like the type of an attribute, or whether a feature is an attribute or a reference. These design decisions remain open and need to be resolved before the model is deemed valid.

The message, the user issuing the message and the performed actions are stored in a traceability model. This allows keeping accurate record of the provenance and rationale of every meta-model element, as well as generating different statistics. Once the meta-model and the trace model have been updated, the bot sends a picture of the meta-model to all users, where the impacted elements are highlighted in a different colour.

1) Example.: Figure 2 illustrates the processing of several NL messages used to build a meta-model for e-learning systems. The first sentence triggers the rule for the verb "to be", and results in the creation of two subclasses of Course. In the second sentence, the user uses a transitive verb ("courses are evaluated with a test"), which yields the creation of a new class Test and a reference evaluatedWith. The bot follows accepted naming conventions: upper camel case for classes and lower camel case for features. Moreover, it also checks the grammatical number of the words to assign an appropriate cardinality to features (e.g., [0..1] in reference evaluatedWith).

III. Tool Support

We have built a modelling bot called Socio to support the presented approach. It works on Telegram and Twitter, though it is designed to be extensible with further social networks and NL rules. The bot stores meta-models in Ecore format, and the traceability data as EMF models. The pictures of meta-models are generated with PlantUML. A video showcasing its use and some examples are available at https://saraperezsoler.github.io/ModellingBot/.

Figure 3 shows some screenshots in the interaction with Socio to build and validate a meta-model for e-learning systems. In the first place (not shown), a Telegram group that includes the participants and the bot needs to be created. In

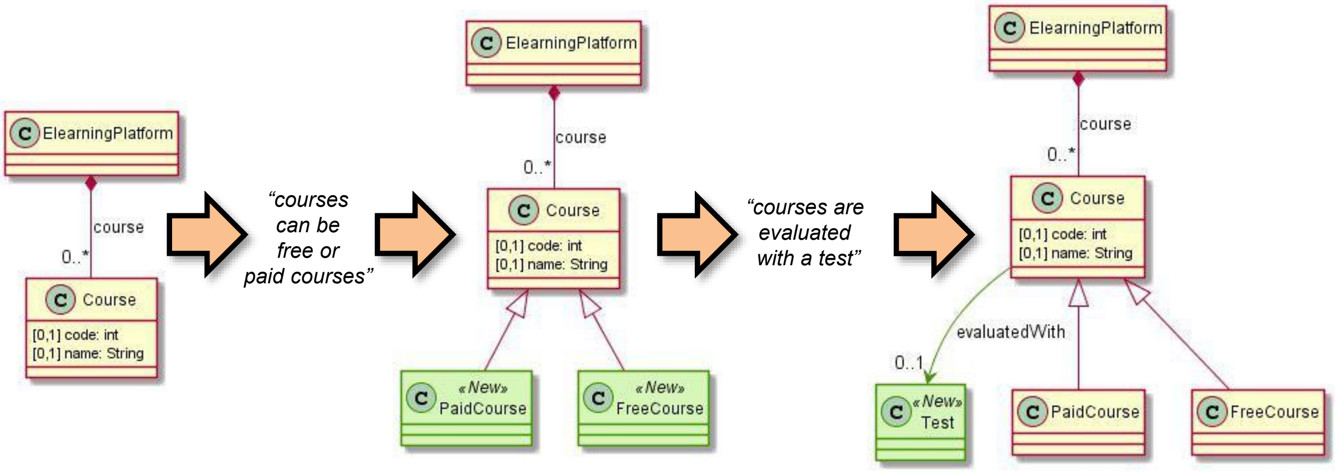


Fig. 2: NL interaction sequence, and corresponding meta-model updates

Figure 3a, the bot shows all available commands, and then, one participant creates a modelling project using the /newproject command.

In Figure 3b, a participant sends a NL message to the bot using the command /talk. The bot interprets the message to deduce domain requirements, updates the current meta-model version accordingly, and returns a picture of the updated meta-model with the created and updated elements highlighted in green. Figure 3c shows a similar interaction using Twitter. In this case, the bot username (@ModellingBot) and the project name (#learningplatform) need to be mentioned. The created attribute (code) is shown in green. After some interactions, one participant validates the model (Figure 3d), and the bot reports there is an error because the type of attribute PaidCourse.price is missing. At any moment, the meta-model can be downloaded in Ecore format using command /get (Figure 4a). The downloaded meta-model can then be used within Eclipse (Figure 4b).

The interaction with Socio is recorded in a traceability model. This can be used to understand the rationale of every decision and analyse user contributions. In particular, Socio offers the following statistics: messages sent by one or all users, meta-model update actions done by one or all users, and percentage of meta-model authorship. They are available through the /history command (see Figure 5a). As an example, Figure 5b shows the number of messages directed to the bot from all users along time, while Figure 5c shows the percentage of authorship. In addition, it is also possible to obtain a more detailed history of the messages sent by each user and their consequences.

IV. Related work

There are several approaches for collaborative modelling or meta-modelling [8]. Some recent examples include Collaboro [9] and SPACEclipse [10]. However, these works do not consider modelling assistants or NL as an interaction mechanism, and they do not rely on social networks but on platforms like Eclipse, which requires technical expertise. Instead, our approach integrates a modelling bot within a social network, so that users do not need to switch between discussion, coordination and modelling tools.

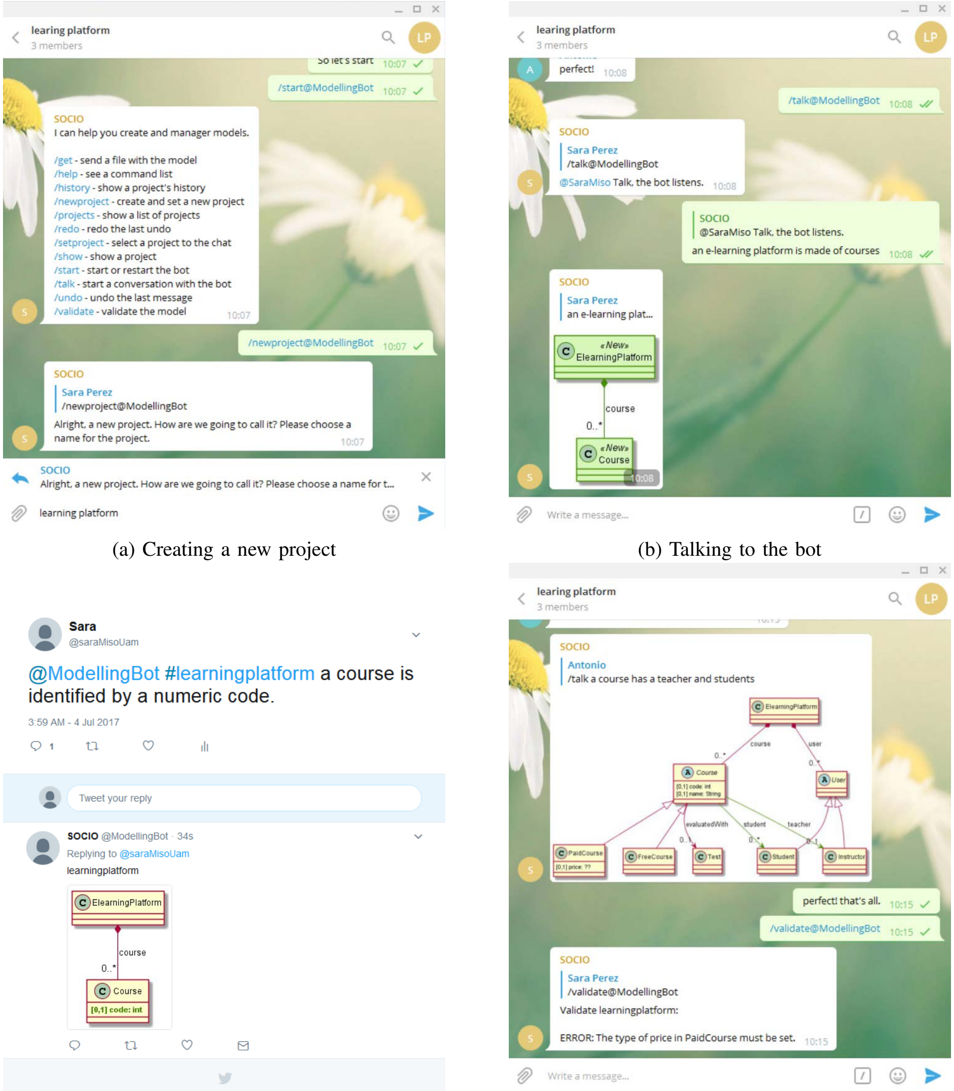
Social networks have been recognised to have a high-impact research potential in software engineering [1], [11]. In fact, we are witnessing an increasing use of bots to automate certain software engineering tasks, like DevOps activities or user support bots [3]. The general goal of their use is to improve efficiency (do things faster) and effectiveness (complete tasks towards meaningful goals). Socio targets improving effectiveness of meta-model creation, lowering the barrier of participation to non-technical people.

Several researchers have proposed different models of crowdsourcing for software design activities. For example, in [12], the authors use microtask crowdsourcing for parallelising the construction of morphological charts, a design technique to represent decision points and alternative solutions. Their experiments show the feasibility of using crowdsourcing to generate a wide range of design solutions, though of varying quality. More generally, Hoang et al. [13] provide a set of recommendations on when to crowdsource decision-making, namely, when tasks can be performed through the internet, do not require a significant level of communication, and can be partitioned into smaller tasks. Moreover, to avoid low quality outcomes, crowdsourced tasks should be easy to evaluate; in our case, live quality checks can be performed by both modelling and domain experts. Finally, platform availability is also key in crowdsourced tasks; our proposal based on social networks completely fulfils this requirement.

Altogether, to the best of our knowledge, our proposal is novel as modelling assistant bots have not been proposed up to now.

V. Conclusions and future work

This paper has described Socio, a bot for assisted modelling over social networks. It works over Twitter and Telegram,



learing platform
3 members

So let's start 10:07

/start@ModellingBot 10:07

SOCIO
I can help you create and manager models.

/get - send a file with the model
/help - see a command list
/history - show a project's history
/newproject - create and set a new project
/projects - show a list of projects
/redo - redo the last undo
/setproject - select a project to the chat
/show - show a project
/start - start or restart the bot
/talk - start a conversation with the bot
/undo - undo the last message
/validate - validate the model 10:07

/newproject@ModellingBot 10:07

SOCIO
Sara Perez
/newproject@ModellingBot
Alright, a new project. How are we going to call it? Please choose a name for the project. 10:07

SOCIO
Alright, a new project. How are we going to call it? Please choose a name for t...

learning platform

(a) Creating a new project

learing platform
3 members

perfect! 10:08

/talk@ModellingBot 10:08

SOCIO
Sara Perez
/talk@ModellingBot
@SaraMiso Talk, the bot listens. 10:08

SOCIO
@SaraMiso Talk, the bot listens.
an e-learning platform is made of courses 10:08

SOCIO
Sara Perez
an e-learning plat...

(b) Talking to the bot

Sara
@saraMisoUam

@ModellingBot #learningplatform a course is identified by a numeric code.

3:59 AM - 4 Jul 2017

Tweet your reply

SOCIO @ModellingBot · 34s
Replying to @saraMisoUam
learningplatform

(c) Interaction through Twitter

learing platform
3 members

SOCIO
Antonio
/talk a course has a teacher and students

perfect! that's all. 10:15

/validate@ModellingBot 10:15

SOCIO
Sara Perez
/validate@ModellingBot
Validate learningplatform:

ERROR: The type of price in PaidCourse must be set. 10:15

(d) Meta-model validation

Fig. 3: Interaction with Socio

interpreting NL sentences from different users to create a meta-model in a collaborative way. The assistant seamlessly integrates within the social network and keeps track of the model history and user contributions.

We are currently improving some aspects of Socio, e.g., enhancing NL processing for more accurate deduction of cardinalities, enabling meta-model instantiation, and defining user roles with support for collaboration protocols. We also plan to integrate other social networks and communication mechanisms, like speech recognition using Skype bots.

Acknowledgements. Work funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

References

[1] M. D. Storey, A. Zagalsky, F. M. F. Filho, L. Singer, and D. M. Germán, "How social and communication channels shape and challenge a participatory culture in software development," *IEEE Trans. Software Eng.*, vol. 43, no. 2, pp. 185–204, 2017.

[2] J. Whitehead, I. Mistrík, J. Grundy, and A. van der Hoek, "Collaborative software engineering: Concepts and techniques," in *Collab. Soft. Eng.* Springer, 2010, pp. 1–30.

[3] M. D. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *FSE*. ACM, 2016, pp. 928–931.

[4] S. Pérez-Soler, E. Guerra, J. de Lara, and F. Jurado, "The rise of the (modelling) bots: Towards assisted modelling via social networks," in *ASE*. IEEE, 2017.

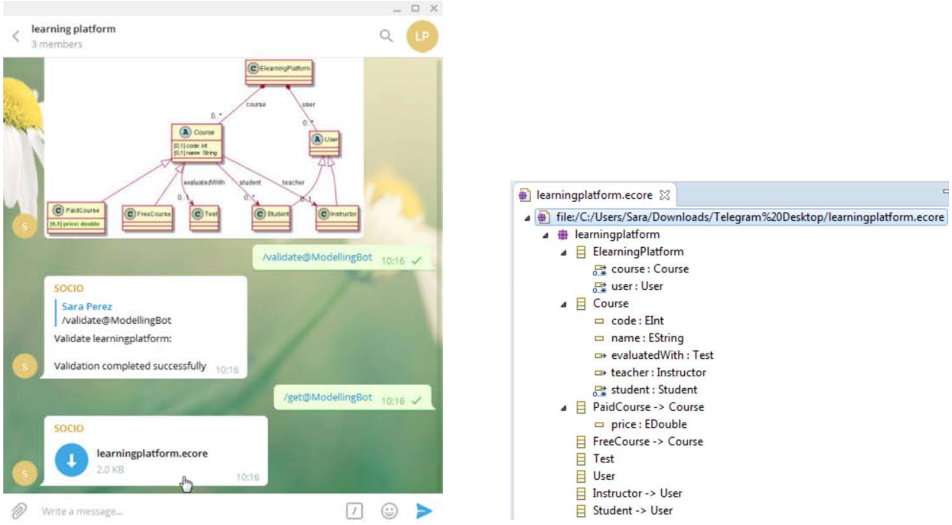
[5] M. Marneffe, B. Maccartney, and C. Manning, "Generating typed dependency parses from phrase structure parses," in *LREC*, 2006.

[6] G. A. Miller, "Wordnet: A lexical database for english," *Comm. ACM*, vol. 38, no. 11, pp. 39–41, 1995.

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

[8] M. Renger, G. L. Kolfschoten, and G. de Vreede, "Challenges in collaborative modelling: A literature review and research agenda," *IJSPM*, vol. 4, no. 3/4, pp. 248–263, 2008.

[9] J. L. C. Izquierdo and J. Cabot, "Collaboro: A collaborative (meta)



learning platform
3 members

/validate@ModellingBot 10:16 ✓

SOCIO
Sara Perez
/validate@ModellingBot
Validate learningplatform:

Validation completed successfully 10:16

/get@ModellingBot 10:16 ✓

SOCIO
learningplatform.ecore
2.0 KB 10:16

Write a message...

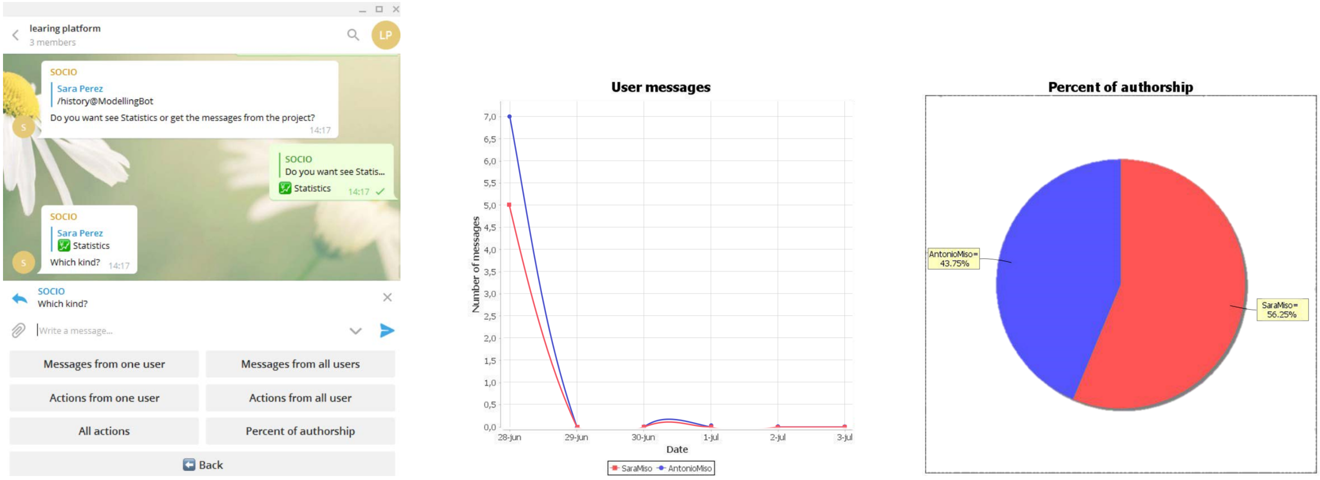
(a) Downloading Ecore meta-model

learningplatform.ecore ☒

- file:/C:/Users/Sara/Downloads/Telegram%20Desktop/learningplatform.ecore
  - learningplatform
    - ElearningPlatform
      - course : Course
      - user : User
    - Course
      - code : EInt
      - name : EString
      - evaluatedWith : Test
      - teacher : Instructor
      - student : Student
    - PaidCourse -> Course
      - price : EDouble
    - FreeCourse -> Course
    - Test
    - User
    - Instructor -> User
    - Student -> User

(b) Ecore meta-model in Eclipse

Fig. 4: Obtaining the final meta-model


learing platform
3 members

SOCIO
Sara Perez
/history@ModellingBot
Do you want see Statistics or get the messages from the project?
14:17

SOCIO
Do you want see Statis...
✓ Statistics 14:17 ✓

SOCIO
Sara Perez
✓ Statistics
Which kind? 14:17

SOCIO
Which kind?

Write a message...

(a) Selecting statistics

(b) Messages sent by users

(c) Percentage of authorship

Fig. 5: Some process statistics

modeling tool," *PeerJ Computer Science*, vol. 2, p. e84, 2016.

[10] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *J. Network and Computer Applications*, vol. 35, no. 3, pp. 1086–1105, 2012.

[11] A. Begel, R. DeLine, and T. Zimmermann, "Social media for software engineering," in *FoSER@FSE*. ACM, 2010, pp. 33–38.

[12] E. Weidema, C. Lopez, S. Nayebaziz, F. Spanghero, and A. van der Hoek, "Toward microtask crowdsourcing software design work," in *CSI-SE@ICSE*, 2016, pp. 41–44.

[13] N. H. Thuan, P. Antunes, and D. Johnstone, "Factors influencing the decision to crowdsource: A systematic literature review," *Information Systems Frontiers*, vol. 18, no. 1, pp. 47–68, 2016.

B.3 Collaborative modelling: chatbots or on-line tools? An experimental study

Title	Collaborative Modelling: Chatbots or On-Line Tools? An Experimental Study	
Publication	Proceedings of the Evaluation and Assessment in Software Engineering (EASE'20)	
Core	2020 Core A	
Authors	Ranci Ren	Universidad Autónoma de Madrid
	Adrián Santos	University of Oulu
	John W. Castro	Universidad de Atacama
	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Silvia T. Acuña	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1145/3383219.3383246	
Date	April 2020	

Abstract

Modelling is a fundamental activity in software engineering, which is often performed in collaboration. For this purpose, on-line tools running on the cloud are frequently used. However, recent advances in Natural Language Processing have fostered the emergence of chatbots, which are increasingly used for all sorts of software engineering tasks, including modelling. To evaluate to what extent chatbots are suitable for collaborative modelling, we conducted an experimental study with 54 participants, to evaluate the usability of a modelling chatbot called SOCIO, comparing it with the on-line tool Creately. We employed a within-subjects cross-over design of 2 sequences and 2 periods. Usability was determined by attributes of efficiency, effectiveness, satisfaction and quality of the results. We found that SOCIO saved time and reduced communication effort over Creately. SOCIO satisfied users to a greater extent than Creately, while in effectiveness results were similar. With respect to diagram quality, SOCIO outperformed Creately in terms of precision, while solutions with Creately had better recall and perceived success. However, in terms of accuracy and error scores, both tools were similar.

Collaborative modelling: chatbots or on-line tools? An experimental study

Ranci Ren
Universidad Autónoma de Madrid
Madrid, Spain
ranci.ren@estudiante.uam.es

John W. Castro†
Universidad de Atacama
Copiapó, Chile
john.castro@uda.cl

Adrián Santos
University of Oulu
Oulu, Finland
adrian.santos.parrilla@oulu.fi

Sara Pérez-Soler, Silvia T. Acuña, Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain
{sara.perezs, silvia.acunna, juan.delara}@uam.es

ABSTRACT

Modelling is a fundamental activity in software engineering, which is often performed in collaboration. For this purpose, on-line tools running on the cloud are frequently used. However, recent advances in Natural Language Processing have fostered the emergence of chatbots, which are increasingly used for all sorts of software engineering tasks, including modelling. To evaluate to what extent chatbots are suitable for collaborative modelling, we conducted an experimental study with 54 participants, to evaluate the usability of a modelling chatbot called *SOCIO*, comparing it with the on-line tool *Creately*. We employed a within-subjects cross-over design of 2 sequences and 2 periods. Usability was determined by attributes of efficiency, effectiveness, satisfaction and quality of the results. We found that *SOCIO* saved time and reduced communication effort over *Creately*. *SOCIO* satisfied users to a greater extent than *Creately*, while in effectiveness results were similar. With respect to diagram quality, *SOCIO* outperformed *Creately* in terms of precision, while solutions with *Creately* had better recall and perceived success. However, in terms of accuracy and error scores, both tools were similar.

CCS CONCEPTS

• Human-centered computing • Usability testing • *Software and its engineering* • *Software design engineering* • *Collaboration in software development*

KEYWORDS

Collaborative modelling, Usability, Chatbots, Effectiveness, Efficiency, Satisfaction, Quality.

ACM Reference format:

Ranci Ren, John W. Castro, Adrián Santos, Sara Pérez-Soler, Silvia T. Acuña and Juan de Lara. 2020. Collaborative modelling: chatbots or on-line tools? An experimental study. In *Proceedings of International Conference on Evaluation and Assessment in Software Engineering (EASE'20). ACM, New York, NY, USA, 10 pages.* https://doi.org/10.

1 Introduction

Modelling is an integral part of all engineering disciplines, like mechanical, electrical or software engineering. Often, modelling becomes a collaborative activity, requiring the participation of stakeholders with different backgrounds and technical expertise [7]. In software engineering, both asynchronous (i.e., based on version control) and synchronous (e.g., based on on-line tools) modelling mechanisms are typically used. For the latter purpose, a plethora of cloud-based platforms have recently emerged, supporting real-time collaboration. These include tools like *GenMyModel* (https://www.genmymodel.com/), *LucidChart* (https://www.lucidchart.com/), *Gliffy* (https://www.gliffy.com/), and *Creately* (https://creately.com/) among many others.

The advance in Natural Language Processing (NLP) techniques has favoured the emergence of chatbots [17]. These are software programs whose user interface is NL – either in text or speech forms – which are frequently embedded within social networks. Almost every industry is proposing chatbots to provide a more flexible access to their services – such as booking flights, perform bank operations, or checking traffic conditions – without the need to install dedicated apps [26]. The boost of chatbots is also partially due to the facilities offered by social networks – like Telegram, Twitter, or Slack – for their integration. Hence, any company can ride the wave of, e.g., Facebook Messenger's success and its huge audience to deploy a bot to engage with the customer. Tens of thousands of chatbots have been created for Facebook Messenger alone. According to forecasts and statistics from Gartner, the chatbot market is quickly growing, since 85% of customer relationships will be supported by artificial intelligence by 2020

†Corresponding Author.

EASE'20, April, 2020, Trondheim, Norway
© 2020 Copyright held by the owner/author(s). 978-1-4503-0000-0/18/06...$15.00
https://doi.org/10.1145/1234567890

[9]. Not only for leisure, but chatbots are increasingly being used to automate software engineering tasks as well. For example, developers use bots to automate deployment tasks, assign software bugs and issues, repair build failures, schedule tasks like sending reminders, integrate communication channels, or for customer support [17]. Recently, chatbots have been proposed for collaborative modelling. For example, in [20][21] a chatbot called *SOCIO* (saraperezsoler.github.io/ModellingBot/) was proposed. The chatbot is integrated within social networks, and interprets the NL phrases of groups of users to create a domain model. This approach lowers the entry barrier to modelling of non-technical experts, promoting a more active role of all the involved stakeholders in a project. In addition, the social network provides natural collaboration support via short messages.

Given the prominent position that chatbots are expected to take in software engineering, our objective is to assess to what extent a chatbot-based approach is suitable for collaborative modelling. For this purpose, we evaluate two alternative tools for collaborative modelling: *Creately* and *SOCIO*. The former is taken as representative of on-line tools, providing a baseline for comparison. The evaluation is based on a user study with 54 participants, and the assessment is made in terms of usability, efficiency, effectiveness, satisfaction and quality of the results. Overall, we found that *SOCIO* saved time and reduced communication effort over *Creately*. *SOCIO* satisfied users to a greater extent than *Creately*, while in effectiveness the results were similar. With respect to diagram quality, *SOCIO* outperformed *Creately* in terms of precision, while solutions with *Creately* had better recall and perceived success. However, in terms of accuracy and error scores, both tools performed similarly. On the one hand, the experiment findings validate an approach based on chatbots for collaborative modelling. This fact is relevant for builders of future modelling tools. On the other, the experiment advances our general understanding of usability of chatbots and provides directions for how to evaluate the usability of chatbots. As noted in [27], the construction of chatbots frequently neglects usability concerns. Hence, techniques for measuring their usability need to be investigated, to help improving the user experience. While experimentation is key in software engineering, there are still few experiments specifically targeting chatbot usability [23]. Our experiment can serve as a guide for the evaluation of chatbots in software engineering.

Paper organization. Section 2 analyses related work, while Section 3 introduces *SOCIO* and *Creately*. Section 4 describes the research method of the experiment, and Section 5 presents the results and their analysis. Section 6 discusses the results and the threats to validity. Finally, Section 7 concludes the paper.

2 Related Work

Next, we review collaborative modelling approaches, with emphasis on user studies; and on evaluations of chatbots' usability.

Collaborative modelling. Software engineering is a team activity, involving multiple engineers and stakeholders [30]. In the analysis and design phase of a project, collaboration involves sharing and working on a set of models. Our focus is on synchronous modelling. As mentioned in the introduction, a plethora of cloud-based tools have emerged, and traditional desktop based platforms, like Eclipse are targeting the web as well (see e.g. EMF.Cloud www.eclipse.org/emfcloud/, or the Graphical Language Server Protocol, www.eclipse.org/glsp/). Usability is one of the limiting factors of collaborative tools, as reported in [7]. Usability can be evaluated via experiments, and we now review some representative ones.

Some collaborative modelling tools can be used from within mobile devices. This is the case of NetSketcher, a tool to build process models [2]. The tool was evaluated informally on a task performed by 6 undergraduates. Also in the area of process modelling, the Cheetah Experimental Platform (CEP) is a collaborative, desktop-based tool with support for collaboration [6]. The tool was evaluated informally, with two engineers creating a simple model. The experiment analysed the collaboration process itself, e.g., observing change of roles of the users (active vs passive) during the collaboration. In [8], Eclipse GMF editors were incorporated collaboration capabilities. The tool was evaluated by 14 students, which defined both a modelling tool (using MDE techniques), and then evaluated the generated tool. Evaluation was performed using questionnaires. Hence, these are small-scale experiments, while the field would benefit from larger ones.

Usability of chatbots. In [23] a Systematic Mapping Study (SMS) is presented, analyzing the HCI mechanisms used to evaluate the usability of chatbots in different fields. In the health care domain, chatbots helped to self-control diseases such as diabetes [4][25], or offered therapy for patients suffering from post-traumatic stress disorders [28]. Other bots were designed to facilitate travel planning [19], help in e-commerce like buying shoes [13], search for information [22] or being the personal assistants, such as Apple Siri and Amazon Alexa [5][18]. The SMS selected 15 papers as primary studies, where 10 described experimental studies of chatbot usability. In most cases, the studies compared the chatbot with another application or system with same functionality or similar key characteristics [4][13][19][22][25]. For example, in [19], a website application and a chatbot are compared to investigate the differences in the levels of satisfaction. In most experiments, simple tasks are proposed, like using Siri to find an inexpensive hotel in Osaka [5] or search a flight ticket and hotel room via the chatbot [19]. A within-subject design was used in three experiments [13][19][22], in which subjects must apply all the treatments to be evaluated. However, each treatment was only used in a particular order. All experiments used questionnaires to collect data about user experience and satisfaction. These were typically provided at the end of the experiment, although in some cases, they were also filled after each task and/or at the beginning of the experiment to better understand basic information about the users [13][22].

With respect to chatbots for collaborative modelling, in [20][21] two small-scale evaluation experiments for *SOCIO* (with 19 and 8 participants) were presented. In [21] the suitability of this chatbot was assessed, while in [20], they evaluated a consensus mechanism for choosing different modelling alternatives. The tasks in both experiments were carried out in groups. All 10 participants in [21] performed the proposed task via Telegram, and were divided in 4 groups (of 2 and 3 people). In [20], all participants formed a single Telegram group. The research method used in both experiments was based on survey questionnaires, filled after finishing the tasks. In [21], the questionnaire was based on the System Usability Scale (SUS) [3], with a part to evaluate user satisfaction, the use of NL, the integration in social networks, and open questions. Questions in [20] focused on evaluating the consensus mechanism. The tasks proposed were relatively simple. In [21], the task was creating a class diagram for an electronic commerce system in 15 minutes. In [20], to select among modelling alternatives to measure the degree of agreement based on the group preferences. Participants chose the best of three options for two projects, the first without consensus mechanism and the second with it. The results in [21] were positive in terms of satisfaction, the suitability of using NL and the idea of collaborating in social networks. Even though the accuracy of interpreting NL was relatively good, results suggested the need to improve in this line. The consensus mechanism was considered useful for large groups and with an outcome that reflects the opinion of the majority [20].

However, those experiments focused on evaluating SOCIO in isolation (i.e., no comparison to a baseline), while the number of participants was small. Therefore, here we report on an experiment with larger number of users and compare with an alternative collaborative modelling approach, based on a traditional GUI.

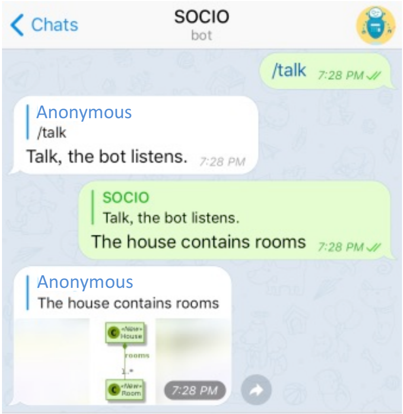
3 A brief overview of *SOCIO* and *Creately*

SOCIO is a chatbot that interprets NL (in English) to create class diagrams [21]. The chatbot is accessible from Twitter or Telegram (with nick *@ModellingBot*). Upon interpreting a NL phrase uttered by the user, it sends back an image with the current model state, with colors highlighting the changed parts. *SOCIO* supports commands to create new models, see user contributions, the percentage of authorship on the created models, among others.

SOCIO offers two types of interaction. The first one is similar to a casual task, based on descriptive phrases like "*the house contains rooms"* (cf. Figure 1). *SOCIO* identifies the relevant parts of a phrase (nouns, verbs, adjectives), to decide which actions to perform (creating or updating a class, an attribute, a relation). In the example of the Figure, it identifies two nouns (house, rooms), for which two classes are created. The "contains" verb is mapped to a containment reference, while the plural form of "rooms" suggests a *many* cardinality.

It can be noted that, in Telegram, the bot cannot directly listen to messages of the users in a group, which need to address the bot using the "*/talk*" command.

The second way to address the bot is more similar to using commands, like "*add class X*'', or "*set attribute size to int*''. Still, these commands have a flexible syntax, as illustrated in Figure 2 (where again the class names are added in singular).



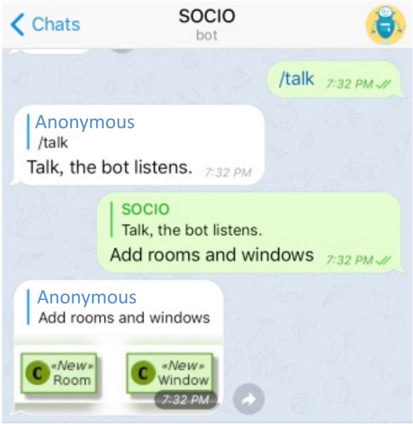
/talk 7:28 PM

Anonymous
/talk
Talk, the bot listens. 7:28 PM

SOCIO
Talk, the bot listens.
The house contains rooms 7:28 PM

Anonymous
The house contains rooms

Figure 1: Processing descriptive NL messages



/talk 7:32 PM

Anonymous
/talk
Talk, the bot listens. 7:32 PM

SOCIO
Talk, the bot listens.
Add rooms and windows 7:32 PM

Anonymous
Add rooms and windows

Figure 2: Processing command-like imperative messages

In contrast to *SOCIO, Creately* uses a traditional GUI, accessible through a web browser. The tool supports over 50 types of diagrams – including class diagrams – and real-time collaboration. In addition, the tool supports working offline, and re-synchronization when connectivity is available.

Figure 3 shows a screenshot of the tool. *Creately* is built on Adobe's Flex/Flash technologies and provides a visual communication platform for virtual teams. While *SOCIO* embeds modelling within a social network, *Creately* lacks an embedded chat. Hence, external ones, like Telegram should be used instead.

Since *Creately* is one of the most used online collaborative modelling tools[1] with friendly interface and learnability, we chose it as the control tool for comparing with *SOCIO*.

[1] according to modeling-languages.com/web-based-modeling-tools-uml-er-bpmn/

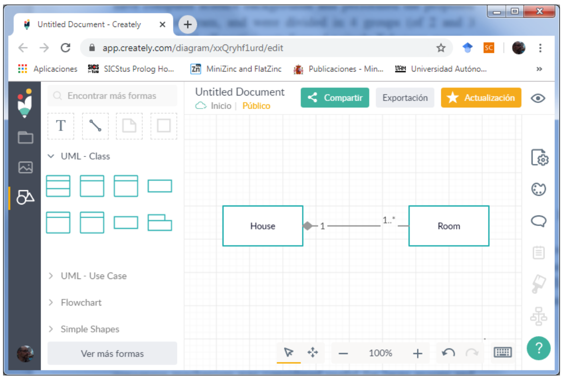


Figure 3: *Creately* being used for class diagram modelling

4 Research Method

The objective of the research is to evaluate the usability of the chatbot *SOCIO* by comparing it to the web tool *Creately* with respect to effectiveness, efficiency and satisfaction, from the point of view of users, and the quality of the class diagrams obtained. In particular, we make the following research question:

RQ: Compared to *Creately,* does the use of *SOCIO* positively affect the efficiency, effectiveness and satisfaction of the users when making class diagrams, and the quality of class diagrams?

The research hypotheses are:

H.1.0 There is no difference in efficiency between *SOCIO* and *Creately* when making a class diagram.

H.2.0 There is no difference in effectiveness between *SOCIO* and *Creately* when making a class diagram.

H.3.0 There is no difference in satisfaction between *SOCIO* and *Creately* when making a class diagram.

H.4.0 There is no difference in the quality of the class diagram made with *SOCIO* or *Creately*.

4.1 Experimental setting

The experiment was structured as a 2 **sequences** and 2 **periods within-subjects cross-over** design (see Table 1). Cross-over designs have the advantage of reducing variability – as subjects act as their own baseline – and require a smaller number of subjects than between-subjects designs – as subjects have as many measurements as periods [29].

The participants were grouped in *teams* of 3 members. The teams were randomly assigned to one out of two groups (Group 1 or Group 2, onwards), so each group applies the *treatments* in a different order (AB/BA). The *treatments* are two tools for creating class diagrams: the chatbot *SOCIO* and the web application *Creately*. Group 1 first applies *SOCIO* and then *Creately* (i.e., *SOCIO-Creately* sequence, SC-CR). Conversely, group 2 first applies *Creately* and then *SOCIO* (i.e., *Creately-SOCIO* sequence, CR-SC). Both groups implement the tasks in the same order (task 1 and task 2). Each task consists of a class diagram that needs implementing.

Table 1: Experimental Design

Tool	Task Period Sequence	Task 1 Period 1		Task 2 Period 2	
		SC	CR	SC	CR
Group 1:	*SC-CR*	X	__	__	X
Group 2:	*CR-SC*	__	X	X	__

Finally, participants in the same team are only allowed to communicate with each other in Telegram groups – so as to ensure that we record all the experimental data.

4.2 Participants

A total of 54 participants took part in the experiment. They all had a degree in Computer Science or a related degree from the *Universidad de las Fuerzas Armadas ESPE Extensión Latacunga* in Ecuador. All participants had studied or were studying a course on Software Analysis and Design. Thus, they had the necessary knowledge to make a class diagram.

4.3 Procedure

The 54 participants were split into two groups of 27 participants each. The participants in each group were further divided into 9 teams. The teams were randomly created. A total of 18 teams participated in the experiment (9 per group). To fit within the participants’ timetable, the experiment was run in four sessions over two days, with each participant attending one session.

The subjects did not undergo any preparatory or practice session before the experiment took place. All the subjects signed an informed consent form indicating that they granted us permission to record their data via Telegram. Then, subjects completed a familiarity questionnaire designed to help us collect their basic information (i.e., age, gender, level of English, preconceived ideas regarding their use of social media, and level of knowledge on class diagrams).

All the participants first received a brief tutorial about the tool they had to use. Then, they were required to perform the first task with the tool in a maximum of 30 minutes. We found such length appropriate so as not to fatigue the participants. We adapted the complexity of the class diagram to the experimental session length. In particular, it was a class diagram representing a store, including management of products and customers. At the end of the experimental session the subjects filled in a modified and validated satisfaction questionnaire System Usability Scale (SUS) associated with the tool [3].

Once the questionnaire was completed, participants received a tutorial of the second tool. Then, they performed the second task with the tool in a maximum of 30 minutes. The task consisted in designing the class diagram of a school supporting courses and students. At the end of the allowed time the participants filled in another modified SUS satisfaction questionnaire, with questions about the tool. In this last questionnaire, the participants were asked if they preferred *SOCIO* or *Creately*. Figure 4 shows the detail of each session. The experimental data and materials can be downloaded from: https://bit.ly/2vfYZNB.

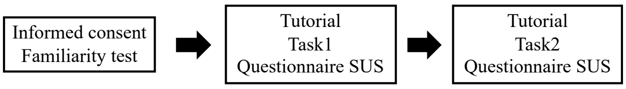


Figure 4: Experimental Procedure

4.4 Measure

The ISO/IEC 25010 [12] defines efficiency, effectiveness, satisfaction, and quality in use as common attributes for evaluating product usability. The response variables that we used and their respective metrics are outlined below.

We used the following metrics to measure efficiency:

- *Speed*: Time, measured in minutes, taken by a team to complete the task (with a maximum of 30 minutes).
- *Fluency*: Number of discussion messages generated by a team during the completion of the task via a Telegram group.

The metric we used to measure effectiveness was *completeness*, based on the *perceived success* in carrying out the task. Satisfaction was measured by the modified SUS questionnaire, including SUS questions, and three or four open-ended questions. The SUS questions are ordinal questions on a 5-point Likert scale – with a rating of 1 to 5, 1 representing "*strongly disagree*", and 5 representing "*strongly agree*". We select the median of the scores given by the three members of each team – to each question – as the score of the team. Finally, we calculate the average of the SUS scores of each team as their satisfaction score. We adopted Brook's equations [15][24] to derive the numerical value of each user's individual tool session score. The corresponding equations are shown below:

For questions 1, 3, 5, 7, 9:

$$\text{Sum1} = \text{score value} - 1 \tag{1}$$

For questions 2, 4, 6, 8, 10:

$$\text{Sum2} = 5 - \text{score value} \tag{2}$$

$$\text{SUS score} = 2.5 * (\text{sum1} + \text{sum2}) \tag{3}$$

Based on the values derived from this equation, we compared these two tools in matters of *satisfaction*. This calculation provided us with a way of quantifying satisfaction. We took an ideal class diagram as a reference to measure the **quality** of the teams' class diagrams. Such class diagram was designed by Software Engineering experts before the experiment took place. We used the following metrics to measure quality [10]:

$$\textbf{Precision} = TP / (TP + FP) \tag{4}$$

$$\textbf{Recall} = TP / (TP + FN) \tag{5}$$

$$\textbf{Accuracy} = (TN + TP) / (TP + FP + FN + TN) \tag{6}$$

$$\textbf{Error} = (FP + FN) / (TP + FP + FN + TN) \tag{7}$$

$$\textbf{Success} = TP / (\#\text{predicted diagram elements}) \tag{8}$$

The previous formulas can be computed by comparing the ideal class diagram with the class diagrams' true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN):

- TP (true positive): Number of elements that are found in both the ideal class diagram and the team' class diagram.
- FN (false negative): Number of elements that are found in the ideal class diagram, but not in the team' class diagram.
- FP (false positive): Number of elements that are found in the team class diagram, but not in the ideal class diagram.
- TN (true negative): In the comparison of models there are no true negatives, and hence the value is always 0.

This way, *precision* gives the percentage of correct classes in the solution of each team, based on the elements of the ideal diagram. *Recall* is a completeness metric, giving the percentage of classes of the ideal diagram present in the solution. *Accuracy* combines both metrics, and *error* reflects how many elements are redundant or missing in each solution. The perceived *success* refers to success rate of each team, compared with the ideal class diagram directly.

5 Data Analysis and Results

We analysed each of the four response variables (i.e., efficiency, effectiveness, satisfaction and quality) with a Linear Mixed Model following the advice of Vegas et al. [29]. In particular, we fitted a linear mixed model with the following factors: **(1) sequence** (either *Creately-SOCIO* or *SOCIO-Creately*), accounting for the assignment of teams to a combination of task and treatment; **(2) period** (either Session 1 or Session 2), confounded with task, accounting for the task that the teams had to implement; and **(3) treatment** (either *Creately* or *SOCIO*), accounting for the tool applied by the teams to implement the tasks.

We complement the results of the statistical analysis with Cohen's *d* for the treatments (*d*, hereinafter) and their standard errors (SEs). For this, we follow the formulae provided in the Cochrane Handbook for cross-over designs [11]. In the next subsections, we go over the data analysis.

5.1 Descriptive Data

According to the data gathered in the familiarity questionnaire, the participants have the following characteristics:

- From a total of 54 subjects, 44 are men and 10 are women.
- Subjects have a mean age of 22 and a standard deviation of 1.74. The highest concentration of participants is in the range 21-23 years.
- 66.7% of subjects use social media frequently. WhatsApp, Facebook, Instagram and Telegram are the most used social media applications.

- All the participants believe they are knowledgeable about class diagrams, and 90% of them relatively familiar with class diagrams.
- 87.1% of the participants have used or use Telegram frequently. 12.9% have no experience using Telegram.
- In relation to chatbots, all participants consider they understand them – at least at the conceptual level. Regarding their usage habits, 29.6 % have never used a chatbot, while 70.4% have some experience (55.6% have used chatbots at times and 14.8% are regular users). The fact of having subjects lacking previous experience with chatbots contributes to the greater sensitivity to the usability of the tool and the validity of the results.
- Although no subject is a native English speaker, all of them considered having a fluent level of English.

5.2 Efficiency

We measured efficiency in terms of *speed* and *fluency*. Speed corresponds to the time taken to complete the tasks. Fluency corresponds to the number of discussion messages exchanged between the team members during the tasks' implementation. Figures 5 and 6 show the box-plots corresponding to speed and fluency, respectively. Table 2 and 3 show the results of the linear mixed model fitted to analyse the data.

5.2.1 Speed. As we can see in Figure 5, time spent seems to be less on *SOCIO* than in *Creately*.

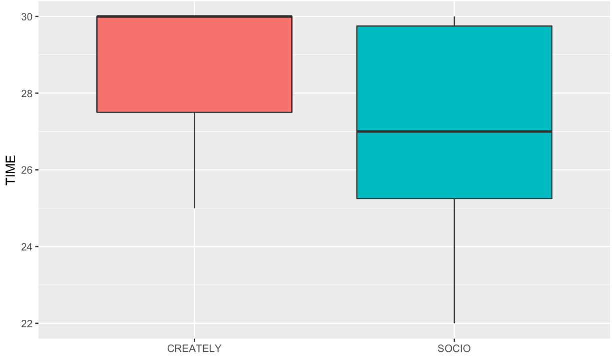


Figure 5: Time spent on completing the task

As we can see in Table 2, only the treatment has a statistically significant impact on time. In particular, the time spent with *Creately* is on average 1.78 minutes longer than that in *SOCIO*. Finally, d=0.80, SE(d)=0.41, suggesting that a large effect size – according to rules of thumb [1] – materialized for the treatment. This large effect size could be because: (i) *Creately* is built on Adobe Flash which caused errors, and at times users needed to re-enter the application during the experiment, and (ii) the delay in the collaborative process with *Creately* was noticeable, and sometimes users could not perform any operations while teammates were operating or (iii) Creately requires users to take care of layouting the diagram, which is not necessary in *SOCIO*.

Table 2: Linear Mixed Model for Time

	Estimate	Std. Error	*p*-value
(Intercept)	27.89	0.73	0.00
Seq	1.11	0.73	0.15
Treatment	-1.78	0.73	**0.03**
Period	0.78	0.73	0.30

5.2.2. Fluency. As we can see in Figure 6, the number of discussion messages seem smaller for *SOCIO* than for *Creately*.

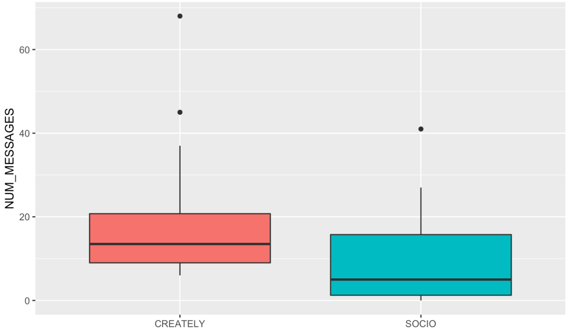


Figure 6: Number of Discussion Messages

As we can see in Table 3, only the treatment has a statistically significant impact on the number of discussion messages. Compared with *SOCIO*, the users sent 10 more messages with *Creately*. With d=0.70, SE(d)=0.22, a relatively large effect size materialized [1].

Table 3: Linear Mixed Model for Number of Discussion Messages

	Estimate	Std. Error	*p*-value
(Intercept)	22.72	4.78	0.0002
Seq	-3.17	6.10	0.61
Treatment	-9.94	2.92	**0.0036**
Period	-3.17	2.92	0.29

In sum, *SOCIO* saved more time in terms of communication effort than *Creately*. **In both aspects, *SOCIO* seems more efficient**.

5.3 Effectiveness

We used the degree of completeness of the tasks to measure effectiveness.

5.3.1 Completeness. Figure 7 shows a box-plot corresponding to the completeness scores of the teams per treatment. As we can see in Figure 7, the results for *completeness* seem similar – albeit more spread for *Creately*.

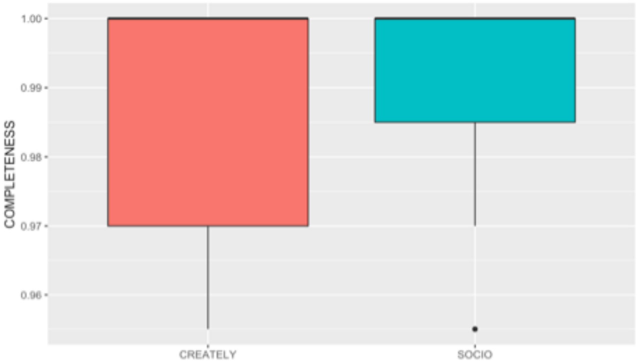


Figure 7: Completeness Scores

Table 4 shows the results of the linear mixed model fitted. As we can see in Table 4, none of the factors has a statistically significant impact on completeness. Finally, d=-0.21, SE(d)=0.34 suggesting a small effect size [1].

Table 4: Linear Mixed Model for Completeness

	Estimate	Std. Error	*p*-value
(Intercept)	0.985	0.00515	0.00
Seq	0.005	0.00518	0.34
Treatment	0.003	0.00512	0.52
Period	0.0083	0.00512	0.12

Thus, ***SOCIO* and *Creately* performed similar in terms of effectiveness**.

5.4 Satisfaction

We used a questionnaire to evaluate users' satisfaction towards *SOCIO* and *Creately*. Each questionnaire included the ten questions of the SUS and four open questions at the end.

5.4.1 Open-ended Questions. Both tools were reliable according to the participants. However, compared to *Creately*, *SOCIO* received more positive responses. Next, we analyse each question:

Q1: Please indicate three positive aspects that you want to highlight about the tool.

Both tools satisfied the participants because of their responsiveness, ease of use and collaboration capabilities. Besides, *Creately* was praised for its friendly interface. Some participants claimed that the chatbot was user-friendly and that it allowed them to have a more entertaining interaction. In other words, *SOCIO* was better suited to entertain the users.

Q2: Please indicate three negative aspects of the tool

Some participants complained that *SOCIO*'s documentation on its website were not sufficient. Additional answers mentioned that commands for *SOCIO* were not easy to learn, and some commands were lacking. Some of the participants expressed that *SOCIO* requires prior knowledge.

The biggest problems with *Creately* were related to real time collaboration, which produced some errors when loading on some of the user's computers. Some participants were not satisfied with the interface as it was too simple. Besides, some users claim that *Creately*'s functions were not comprehensive enough.

Q3: Do you have any suggestions for improvement?

For SOCIO, a number of participants suggested an improvement in its support for NLP. For *Creately,* participants suggested to improve its real time collaboration, and improve its user interface, which some participants considered too simple.

Q4: Which tool do you prefer?

Participants showed relatively positive emotions towards both tools, especially in the aspect of anticipation. Besides, they expressed more trust and joy for *SOCIO* than for *Creately*. Overall, 30 of the participants preferred *SOCIO*, while 24 expressed their preference towards *Creately*.

5.4.2 Questions of the SUS. We used the SUS score given by the participants to both tools and compared them side-by-side. Figure 8 shows the box-plot for the SUS scores.

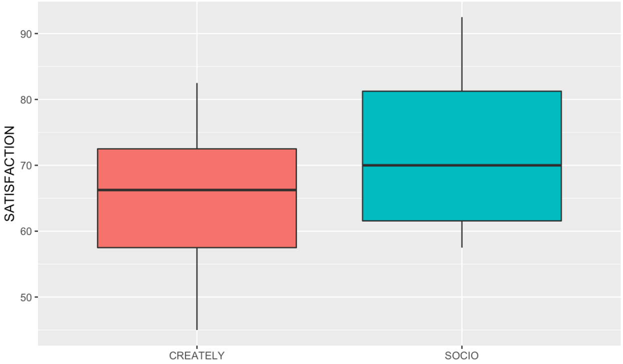


Figure 8: Satisfaction Scores

As Figure 8 shows, the satisfaction scores are typically higher for *SOCIO* than for *Creately*. As we can see in Table 5, the treatment has an almost statistically significant effect on satisfaction (p=0.1). In particular, d=0.58, SE(d)=0.35, thus, suggesting a medium effect size [1]. This indicates that ***SOCIO* satisfies users to a greater extent than *Creately***.

Table 5: Linear Mixed Model for Satisfaction

	Estimate	Std. Error	*p*-value
(Intercept)	64.51	3.88	0
Seq	1.69	3.97	0.69
Treatment	6.60	3.79	0.10
Period	-1.18	3.79	0.75

5.5 Quality

We analysed the quality of the class diagrams in various aspects: *precision*, *recall*, *accuracy*, *error* and *perceived success* (cf. equations 4-8). The box-plots for such metrics are shown in Figure 9-13, while the linear mixed models fitted are shown in Tables 6-10.

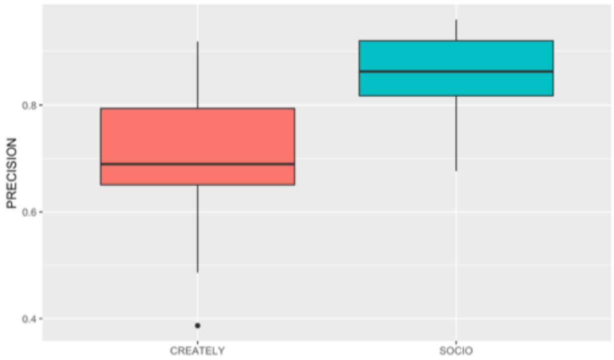


Figure 9: Precision Scores

Table 6: Linear Mixed Model for Precision

	Estimate	Std. Error	*p*-value
(Intercept)	0.731	0.055	0
Seq	0.008	0.066	0.911
Treatment	0.108	0.041	**0.018**
Period	-0.141	0.041	**0.003**

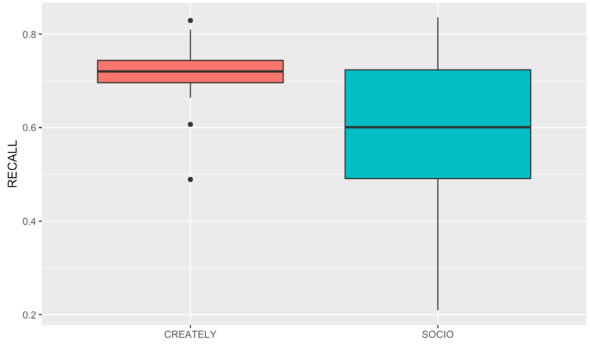


Figure 10: Recall Scores

Table 7: Linear Mixed Model for Recall

	Estimate	Std. Error	*p*-value
(Intercept)	0.729	0.051	0
Seq	-0.026	0.061	0.677
Treatment	-0.145	0.038	**0.001**
Period	-0.006	0.038	0.885

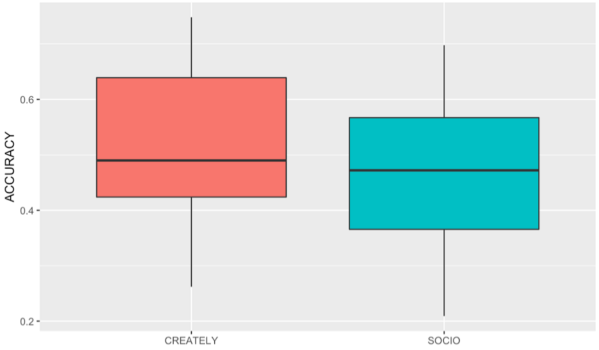


Figure 11: Accuracy Scores

Table 8: Linear Mixed Model for Accuracy

	Estimate	Std. Error	*p*-value
(Intercept)	0.546	0.048	0
Seq	0.015	0.067	0.81
Treatment	-0.048	0.031	0.13
Period	-0.069	0.031	**0.04**

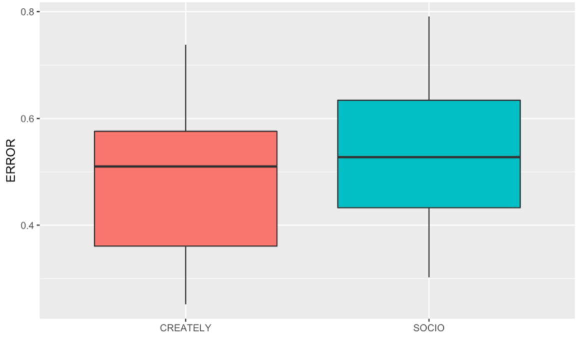


Figure 12: Error Scores

Table 9: Linear Mixed Model for Error

	Estimate	Std. Error	*p*-value
(Intercept)	0.453	0.048	0
Seq	-0.015	0.061	0.812
Treatment	0.048	0.031	0.135
Period	0.069	0.031	**0.039**

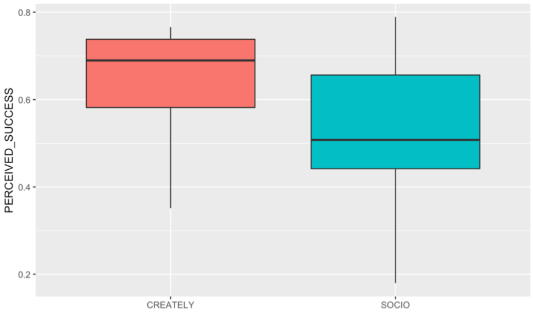


Figure 13: Perceived Success

Table 10: Linear Mixed Model for Perceived Success

	Estimate	Std. Error	*p*-value
(Intercept)	0.642	0.049	0
Seq	0.066	0.058	0.270
Treatment	-0.147	0.038	**0.001**
Period	-0.049	0.038	0.218

As shown above, the treatment has a significant impact on *precision* (where d=0.619, SE(d)=0.324, *SOCIO* outperforming *Creately*); *recall* (d=0.976, SE(d)=0.232, *Creately* outperforming *SOCIO*); and *perceived success* (d=0.996, SE(d)= 0.307, *Creately* outperforming *SOCIO*). However, in terms of accuracy and error scores both tools seem to perform similarly as indicated by the non-significance of the treatment factor, and the smallest effect sizes in such cases (d=0.334, SE(d)=0.240 for accuracy; d=-0.334, SE(d)=0.240 for error scores).

Summarizing, ***SOCIO*** **outperforms** ***Creately*** **in terms of precision, while** ***Creately*** **outperforms** ***SOCIO*** **in terms of recall and perceived success.**

6 Discussion and Threats to Validity

Overall, *SOCIO* seems superior to *Creately* in terms of efficiency and satisfaction, while in effectiveness they are similar. This suggests that *SOCIO* saved time and communication effort to the users. Also, that *SOCIO's* look and feel met the users' expectations to a greater extent than *Creately*. In addition, users created more precise class diagrams with *SOCIO* than with *Creately*. This means that a larger percentage of the classes created with *SOCIO* were also included in the ideal solution. This, and the observation that *Creately* was superior to *SOCIO* in terms of recall and perceived success, suggests that users made fewer classes with *SOCIO* than with *Creately* – albeit the diagrams were more complete with *Creately*. In plain words, users seemed to create more classes from the ideal solution with *Creately* than with *SOCIO* – despite it took longer to the users creating such classes with *Creately*, and *Creately's* interface seems not as appealing as *SOCIO's*.

Our take away from these results is that despite its greater precision, *SOCIO's* class diagrams may be lacking completeness due to the low training of the participants with the tool, and its English interface (as none of the participants was a native English speaker). In fact, participants highlighted the need of *SOCIO* to support more languages (namely, Spanish), social media platforms (e.g., Facebook Messenger rather than Telegram), and the need of more detailed examples in the manual. Also, they wished *SOCIO* helped auto-correcting spelling mistakes. Despite this, the satisfaction of the participants with *SOCIO* is relatively high. Notice that *SOCIO* scores better than *Creately* in some respects, but since it is not known or validated how good *Creately* is, this cannot be used as a basis for a comprehensive evaluation for *SOCIO*. However, the fact that *Creately* is one of the most used tools suggests that it is at least one of the best ones, and supports the conclusion that *SOCIO* is a good modelling tool.

Next we analyse threats to validity. **Internal validity** pertains to confounding factors that could influence our results. In the experiment, participants had to create two class diagrams. Although they already had the necessary knowledge for this task, the first task may have refreshed this knowledge. Therefore, the second treatment applied may provide better results. This can be mitigated by comparing the results in the two periods (two tasks), and studying any improvement observed. Although the sessions did not have an excessively long duration (an hour and a half) there could be a threat of tiredness or boredom. The subjects participated voluntarily, and their collaboration did not imply any impact on the grades of the course, so they might have suffered from a lack of motivation. An additional threat to the internal validity is related to the fact that participants were not English native speakers. Hence, the user experience and time spent may be affected by their English fluency.

Regarding **external validity** (generalizability of the results), our participants are university students with knowledge in computer science and class diagram design. Hence, the results are not generalizable to the industrial field, but can only remain in the academic realm. In addition, the evaluation has used *SOCIO* and *Creately*, therefore the results cannot be directly generalized to other modelling chatbots, or on-line modelling tools.

Besides, there is a threat to **conclusion validity** because we have performed many statistical tests, and hence, this has increased the risk of a statistical error for type I (saying there is an effect, when there is not). We decided not to apply any correction for multiple tests, like Bonferroni, due to the relatively small sample size of the experiment. However, we have complemented the statistical results with the effect sizes, to facilitate the interpretation of the practical relevance of the findings. All in all, we consider these results preliminary and proper sized experiments are still required to draw definite conclusions on the performance of SOCIO and *Creately*.

Finally, there is another threat to conclusion validity regarding the experimental tasks-because they may have impacted the experiment's results. To tackle this shortcoming, we plan to run more experiments assessing the performance of SOCIO and Creately with a different set of tasks.

7 Conclusion

Modelling is a team activity that is often performed in collaboration. Traditionally, collaborative modelling has been performed asynchronously in offline environments, or using online collaboration, sometimes in cloud-based tools. However, we have recently witnessed the emergence of chatbots, which are being used for all types of activities, including software engineering tasks like modelling. As usability of chatbots – in particular for modelling – is largely unexplored, in this paper we have reported on an evaluation comparing the *SOCIO* chatbot with the *Creately* on-line tool. Our aim was to answer the following research question:

> **RQ:** Compared to *Creately,* does the use of *SOCIO* positively affect the efficiency, effectiveness and satisfaction of the users when making class diagrams, and the quality of class diagrams?

We evaluated the usability of *SOCIO* from four aspects: efficiency, effectiveness, satisfaction and quality. Regarding efficiency, teams using *SOCIO* finished earlier than those using *Creately*. For collaboration, those using *SOCIO* showed high fluency, with an interaction-cost advantage over those using *Creately*. For effectiveness, *SOCIO* and *Creately* performed similarly in terms of completeness. For satisfaction, *SOCIO* satisfies users to a greater extent than *Creately* with respect to the results of the SUS score. More users expressed they preferred *SOCIO* rather than *Creately*. For quality *SOCIO* outperformed Creately in terms of precision, while solutions with *Creately* had better recall and perceived success. In sum, usability of *SOCIO* has a positive effect on most aspects, when taking *Creately* as a baseline.

In the future, we plan to conduct a second round of evaluations engaging more users to interact with the chatbot *SOCIO*, especially we will aim at English native speakers. Finally, we would like to enhance *SOCIO* with speech recognition, to enable design workshops using conversation, in the style of [14][16].

Acknowledgements

Work funded by the Spanish Ministry of Science (project MASSIVE, RTI2018-095255-B-I00) and the R&D programme of Madrid (project FORTE, P2018/TCS-4314).

REFERENCES

[1] Michael Borenstein, Larry V. Hedges, Julian P. T. Higgins, and Hannah R. Rothstein. 2009. *Introduction to meta-analysis*. West Sussex, Wiley, UK.

[2] Nelson Baloian, Gustavo Zurita, Flávia Maria Santoro, Renata Mendes de Araujo, S. Wolfgan, D. Machado, and José A. Pino. 2011. A collaborative mobile approach for business process elicitation. In *Proc. 2011 15th Int. Conf. Comp. Supp. Coop. W. Design (CSCWD0'11)*. Lausanne, Switzerland, 473-480.

[3] John Brooke. 1996. *SUS-a quick and dirty usability scale*. In P. W. Jordan, B. Thomas, B. A. Weerdmeester, & A. L. McClelland (Eds.). Usability Evaluation in Industry, Chapter 21, 189-194.

[4] Amy Cheng, Vaishnavi Raghavaraju, Jayanth Kanugo, Yohanes P. Handrianto, and Yi Shang. 2018. Development and evaluation of a healthy coping voice interface application using the google home for elderly patients with type 2 diabetes. In *Proc. 15th IEEE Ann. Consumer Communic. & Netw. Conf. (CCNC'18)*. Las Vegas, NV, USA, 1-5.

[5] Mei-Ling Chen and Hao-Chuan Wang. 2018. How Personal Experience and Technical Knowledge Affect Using Conversational Agents. In *Proc. 23rd Int. Conf. Intell. U. Interf. Comp. (IUI'18 - Comp)*. ACM. Tokyo, Japan. Article 53.

[6] Simon Forster, Jakob Pinggera, and Barbara Weber. 2013. Toward an understanding of the collaborative process of process modeling. *CAiSE Forum* 2013, 98-105.

[7] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. 2018. Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map. *IEEE Transact. Softw. Engin.* 44, 1146-1175.

[8] Jesús Gallardo, Crescencio Bravo, and Miguel A. Redondo. 2012. A model-driven development method for collaborative modeling tools. *J. Network and Comp. Applic.* 35(3), 1086-1105.

[9] Gartner. 2011. CRM Strategies and Technologies to Understand, Grow and Manage Customer Experiences. Gartner Customer 360 Summit 2011. Gartner, Los Angeles, CA.

[10] Fáber D. Giraldo, Sergio España, William Giraldo Orozco, and Oscar Pastor. 2018. Evaluating the quality of a set of modelling languages used in combination: A method and a tool. *Information Systems* 77, 48-70.

[11] Julian P. T. Higgins and Sally Green. 2011. *Cochrane handbook for systematic reviews of interventions*, (vol. 4). John Wiley & Sons.

[12] ISO/IEC 25010. 2011. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models. ISO. Geneva, Italy.

[13] Mohit Jain, Ramachandra Kota, Pratyush Kumar, and Shwetak N. Patel. 2018. Convey: Exploring the Use of a Context View for Chatbots. In *Proc. 2018 CHI Conf. Hum. Fact. Comp. Syst. (CHI'18)*. ACM, New York, USA, Paper 468.

[14] Rodi Jolak, Boban Vesin, Michel R. V. Chaudron. 2017. Using Voice Commands for UML Modelling Support on Interactive Whiteboards: Insights and Experiences. In *Proc. Iber. Conf. Soft. Eng* (*CIbSE'17*). Bs. As., Argentina, 85-98.

[15] Patrick W. Jordan, Bruce Thomas, Ian Lyall McClelland, and Bernard Weerdmeester. 1996. *Usability Evaluation in Industry* (1st. ed.). CRC Press Taylor & Francis Group.

[16] Samuel Lahtinen, Jari Peltonen. 2005. Adding speech recognition support to UML tools. *Journal of Visual Lang. Comput* 16(1-2), 85-118.

[17] Carlene Lebeuf, Margaret-Anne D. Storey, and Alexey Zagalsky. 2018. Software bots. *IEEE Software* 35(1), 18-23.

[18] Irene Lopatovska, Katrina Rink, Ian Knight, Kieran Raines, Kevin Cosenza, Harriet Williams, Perachya Sorsche, David Hirsch, Qi Li, and Adrianna Martinez. 2019. Talk to me: Exploring user interactions with the amazon alexa. *J. Librarianship and Information Science* 51(4), 984-997.

[19] Quynh N. Nguyen and Anna Sidorova. 2018. Understanding user interactions with a chatbot: A self-determination theory approach. In *Proc. Americas Conf. Inform. Syst. 2018: Digital Disrupt. (AMCIS'18).* New Orleans, LA, USA. 1-5.

[20] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2018. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Software* 35(6), 48-54.

[21] Sara Pérez-Soler, Esther Guerra, Juan de Lara, and Francisco Jurado. 2017. The rise of the (modelling) bots: towards assisted modelling via social networks. In *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE'17)*. IEEE Press, Piscataway, NJ, USA, 723-728.

[22] Joaquín Pérez, Yanet Sánchez, Francisco J. Serón, and Eva Cerezo. 2017. Interacting with a Semantic Affective ECA. In *Proc. Int. Conf. Int. Virt. Agents (IVA'17).* Lecture Notes in Computer Science, vol 10498. Springer. 374-384.

[23] Ranci Ren, John W. Castro, Silvia T. Acuña, and Juan de Lara. 2019. Usability of chatbots: A systematic mapping study. In *Proc. 31st Int. Conf. Soft. Eng. and Knowledge Eng. (SEKE'19)*. Lisbon, Portugal, 479-484.

[24] Julia Saenz, Walker Burgess, Elizabeth Gustitis, Andres Mena, and Farzan Sasangohar. 2017. The usability analysis of chatbot technologies for internal personnel communications. In *Proc. 67th Ann. Conf. and Expo of the Instit. of Industrial Engineers*. Pittsburgh, United States, 1357-1362.

[25] Claudia Sinoo, Sylvia van der Pal, Olivier A. Blanson Henkemans, Anouk Keizer, Bert P. B. Bierman, Rosemarijn Looije, and Mark A.Neerincx. 2018. Friendship with a robot: Children's perception of similarity between a robot's physical and virtual embodiment that supports diabetes self-management. *Patient Education and Counseling* 101(7), 1248-1255.

[26] Sam Suthar. 2019. 11 Chatbot Trends to Help Grow your Business in 2019. Retrieved November 11, 2019, from https://acquire.io/blog/chatbots-trends.

[27] The Interaction Design Foundation. 2019. What is usability? Retrieved November 11, 2019, from https://www.interaction-design.org/literature/topics/usability.

[28] Myrthe L. Tielman, Mark A. Neerincx, Rafael Bidarra, Ben Kybartas, and Willem-Paul Brinkman. 2017. A Therapy System for Post-Traumatic Stress Disorder Using a Virtual Agent and Virtual Storytelling to Reconstruct Traumatic Memories. *J. Medical Systems* 41, 125.

[29] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2016. Crossover designs in softw. eng. experiments: Benefits and perils. *IEEE Trans. Soft. Eng.* 42(2), 120-135.

[30] Jim Whitehead, Ivan Mistrík, John Grundy, and André van der Hoek. 2010. Collaborative software engineering: Concepts and techniques. Collaborative Software Engineering. In: Mistrík I., Grundy J., Hoek A., Whitehead J. (eds). *Collaborative Software Engineering* (pp. 1-30). Springer, Berlin, Heidelberg.

B.4 Towards Automating the Synthesis of Chatbots forConversational Model Query

Title	Towards Automating the Synthesis of Chatbots for Conversational Model Query	
Publication	In EMMSAD’2020 (CAISE), LNCS	
Core	2020 Core C	
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Gwendal Daniel	Universitat Oberta de Catalunya
	Jordi Cabot	Universitat Oberta de Catalunya
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1007/978-3-030-49418-6_17	
Date	May 2020	
Abstract	Conversational interfaces (also called chatbots) are being increasingly adopted in various domains such as e-commerce or customer service, as a direct communication channel between companies and end-users. Their advantage is that they can be embedded within social networks, and provide a natural language (NL) interface that enables their use by non-technical users. While there are many emerging platforms for building chatbots, their construction remains a highly technical, challenging task. In this paper, we propose the use of chatbots to facilitate querying domain-specific models. This way, instead of relying on technical query languages (e.g., OCL), models are queried using NL as this can be more suitable for non-technical users. To avoid manual programming, our solution is based on the automatic synthesis of the model query chatbots from a domain meta-model. These chatbots communicate with an EMF-based modelling backend using the Xatkit framework.	

Towards Automating the Synthesis of Chatbots for Conversational Model Query

Sara Pérez-Soler[1], Gwendal Daniel[2], Jordi Cabot[2,3], Esther Guerra[1] ✉, and Juan de Lara[1]

[1] Universidad Autónoma de Madrid (Spain)
[Sara.PerezS, Esther.Guerra, Juan.deLara]@uam.es
[2] Universitat Oberta de Catalunya (Spain)
gdaniel@uoc.edu
[3] ICREA (Spain)
jordi.cabot@icrea.cat

Abstract. Conversational interfaces (also called chatbots) are being increasingly adopted in various domains such as e-commerce or customer service, as a direct communication channel between companies and end-users. Their advantage is that they can be embedded within social networks, and provide a natural language (NL) interface that enables their use by non-technical users. While there are many emerging platforms for building chatbots, their construction remains a highly technical, challenging task.

In this paper, we propose the use of chatbots to facilitate querying domain-specific models. This way, instead of relying on technical query languages (e.g., OCL), models are queried using NL as this can be more suitable for non-technical users. To avoid manual programming, our solution is based on the automatic synthesis of the model query chatbots from a domain meta-model. These chatbots communicate with an EMF-based modelling backend using the Xatkit framework.

Keywords: Model-driven engineering, model query, automatic chatbot synthesis

1 Introduction

Instant messaging platforms have been widely adopted as one of the main technologies to communicate and exchange information. Most of them provide built-in support for integrating *chatbot applications*, which are automated conversational agents capable of interacting with users of the platform [10]. Chatbots have proven useful in various contexts to automate tasks and improve the user experience, such as automated customer services [23], education [9] and e-commerce [21]. However, despite many platforms have recently emerged for creating chatbots (e.g., DialogFlow [6], IBM Watson [7], Amazon Lex [1]), their construction and deployment remains a highly technical task.

Chatbots are also increasingly used to facilitate software engineering activities [5,12] like automating deployment tasks, assigning software bugs and issues, repairing build failures, scheduling tasks like sending reminders, integrating communication channels, or for customer support. In this context, we explored the use of chatbots for domain modelling in previous work [16,17]. Modelling chatbots can be embedded within social networks to support collaboration between different stakeholders in a natural way, and enable the active participation of non-technical stakeholders in model creation.

In the present work, we extend the previous ideas to support natural language (NL) conversational queries over the models. This is a more accessible and user-friendly way to query models than the use of technical languages like OCL (Object Constraint Language [15]). Moreover, we avoid the manual programming of the model query chatbots by their automatic synthesis. For this purpose, our solution is based on (i) the availability of a meta-model describing the structure of the models, (ii) its configuration with NL information (class name synonyms, names for reverse associations, etc.), and (iii) the automatic generation of a chatbot supporting queries over instances of the given meta-model. This approach is implemented on top of the Xatkit model-based chatbot development platform [4], which interprets the generated chatbot model and interacts with an EMF (Eclipse Modeling Framework) backend.

The rest of the paper is structured as follows. First, Section 2 provides motivation using a running example, and introduces background about chatbot design. Then, Section 3 explains our approach, and Section 4 describes the prototype tool support. Finally, Section 5 compares with related works, and Section 6 concludes.

2 Motivation and Background

In this section, we first provide a motivating example, and then introduce the main concepts behind chatbots.

2.1 Motivation

As a motivating example, assume a city hall would like to provide open access to its real-time traffic information system. Given the growth of the open data movement, this is a common scenario in many cities, like Barcelona[4] or Madrid[5].

We assume that the data provided includes a static part made of the different districts and their streets, with information on the speed limits. In addition, a dynamic part updated in real-time decorates the streets and their segments with traffic intensity values and incidents (road works, street closings, accidents or bottlenecks). Fig. 1 shows a meta-model capturing the structure of the provided information.

In this scenario, citizens would benefit from user-friendly ways to query those traffic models. However, instead of relying on the construction of dedicated front-ends with fixed queries, or on the use of complex model query languages like OCL, our proposal is the use of conversational queries based on NL via chatbots. Chatbots can be used from widely used social networks, like Telegram or Twitter, facilitating their use by citizens. Hence, citizens would be able to issue simple queries like *“give me all accidents with more than one injury”*; and also conversational queries like *“what are the incidents in Castellana Street now?”*, and upon the chatbot reply, focus on a subset of the results

[4] `https://opendata-ajuntament.barcelona.cat/`

[5] `https://datos.madrid.es`

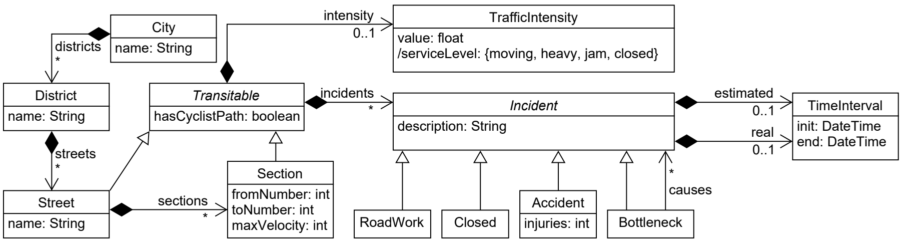


Fig. 1. A meta-model for real-time traffic information.

with *"select those that are accidents"*. Finally, for the case of dynamic models, reactive queries like *"ping me when Castellana Street closes"* would be possible.

Our proposal consists in the generation of a dedicated query chatbot given the domain meta-model. But, before introducing our approach, the next subsection explains the main concepts involved in chatbot design.

2.2 Designing a chatbot

The widespread interest and demand for chatbot applications has emphasized the need
to quickly build complex chatbots
edge base definition [18], and com
position. However, the developme
in several technical domains, ran
of the targeted instant messaging
To alleviate this situation, many
alogFlow [6], IBM Watson [7] or

Fig. 2 shows a simplification of the typical working scheme of chatbots. Chatbots are often designed on the basis of *intents*, where each intent represents some user's aim (e.g., booking a ticket). The chatbot waits for NL inputs from the user (label 1 in the figure); then, it tries to match the phrase with some intent (label 2), optionally calling an external service (label 3) for intent recognition or additional data collection; finally, it produces a response, which is often a NL sentence among a predefined set (label 4).

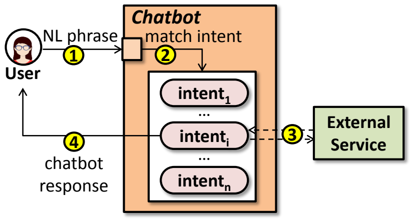


Fig. 2. Chatbot working scheme.

Intents are defined via training phrases. These phrases may include parameters of a certain type (e.g., numbers, days of the week, countries). The parameter types are called *entities*. Most platforms come with predefined sets of entities and permit defining new ones. Some platforms permit structuring the conversation as an expected flow of intents. For this purpose, a common mechanism is providing intents with a *context* that stores information gathered from phrase parameters, and whose values are required to trigger the intent. In addition, there is normally the possibility to have a *fallback* intent, to be used when the bot does not understand the user input.

3 Approach

Fig. 3 shows the scheme of our approach. First, the chatbot designer needs to provide a domain meta-model (like the one in Fig. 1) defining the structure of the models to be queried, and complemented with NL hints on how to refer to its classes and features (synonyms). From this information, an executable chatbot model that can be used to query model instances is generated. The next subsections explain these two steps.

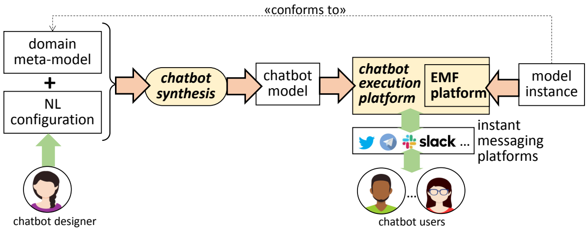


Fig. 3. Scheme of our approach.

3.1 Chatbot generation: Intents and entities model

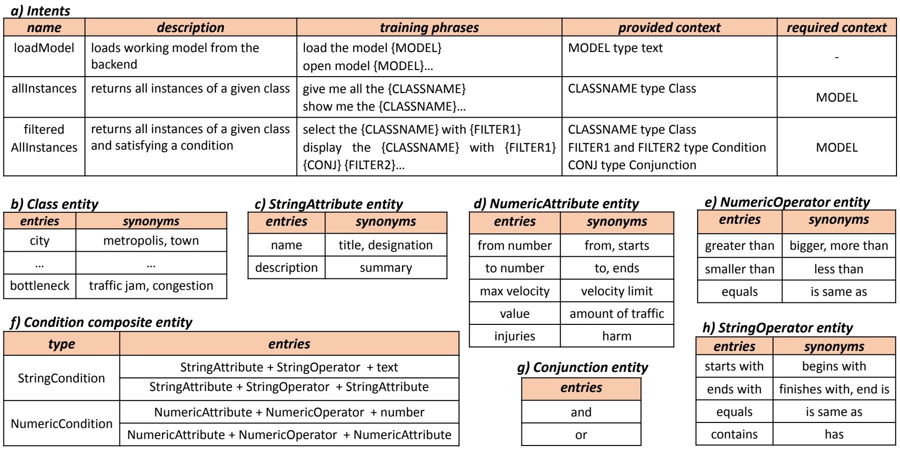
The chatbot designer has to provide a domain meta-model and optionally, a NL configuration model. The latter is used to optionally annotate classes, attributes and features with synonyms, and the source of references with a name to refer to its backward navigation. From this information, we generate the chatbot intents and entities.

Table(a) of Fig. 4 captures the generation of intents. We create an intent per query type, plus an additional intent called loadModel to select the model to be queried. The second row of the table shows the intent allInstances, which returns all objects of a given class. The intent is populated with training phrases that contain the class name as parameter. The possible class names are defined via an entity Class (see Table(b)). This intent would be selected on user utterances such as *"give me all cities"* or *"show every incident"*. The intent requires having a loaded model, which the table indicates as the intent requiring a model as context.

In the same table, intent filteredAllInstances returns all instances that satisfy a given condition. The intent is populated with training phrases that combine a class name and a condition made of one or more filters joined via logical connectives. We provide an entity Condition for the filters, explained below. This intent would be selected upon receiving phrases like *"give me all accidents with more than one injury"* (please note the singular variation w.r.t. the attribute name injuries).

In addition to intents, we create several entities based on the domain meta-model and the NL configuration. Specifically, we create an entity named Class (Table(b)) with an entry for each meta-model class name. These entries may have synonyms, as provided by the NL configuration, to refer to the classes in a more flexible way. Likewise, we create an entity for each attribute name attending to their type: String (Table(c)), Numeric (Table(d)), Boolean and Date (omitted for space constraints). For example, the StringAttribute entity (Table(c)) has an entry for all String attributes called name. Just like classes, these entries may have synonyms if provided in the NL configuration.

The Condition entity (Table(f)) is a composite one, i.e., its entries are made of one or more entities. This entity permits defining filter conditions in queries, such as *"name starts with Ma"* or *"injuries greater than one"*.



a) Intents

name	description	training phrases	provided context	required context
loadModel	loads working model from the backend	load the model {MODEL} open model {MODEL}...	MODEL type text	-
allInstances	returns all instances of a given class	give me all the {CLASSNAME} show me the {CLASSNAME}...	CLASSNAME type Class	MODEL
filtered AllInstances	returns all instances of a given class and satisfying a condition	select the {CLASSNAME} with {FILTER1} display the {CLASSNAME} with {FILTER1} {CONJ} {FILTER2}...	CLASSNAME type Class FILTER1 and FILTER2 type Condition CONJ type Conjunction	MODEL

b) Class entity

entries	synonyms
city	metropolis, town
...	...
bottleneck	traffic jam, congestion

c) StringAttribute entity

entries	synonyms
name	title, designation
description	summary

d) NumericAttribute entity

entries	synonyms
from number	from, starts
to number	to, ends
max velocity	velocity limit
value	amount of traffic
injuries	harm

e) NumericOperator entity

entries	synonyms
greater than	bigger, more than
smaller than	less than
equals	is same as

f) Condition composite entity

type	entries
StringCondition	StringAttribute + StringOperator + text
	StringAttribute + StringOperator + StringAttribute
NumericCondition	NumericAttribute + NumericOperator + number
	NumericAttribute + NumericOperator + NumericAttribute

g) Conjunction entity

entries
and
or

h) StringOperator entity

entries	synonyms
starts with	begins with
ends with	finishes with, end is
equals	is same as
contains	has

Fig. 4. Intents and entities generated for the running example chatbot.

Regarding the complexity of the chatbot, the number of intents is fixed, and it depends on the primitives of the underlying query language that the chatbot exposes. Fig. 4 exposes two primitives of OCL: allInstances, and allInstances()→select(cond). Other query types can be added similarly, which would require defining further intents. The number of generated entities is also fixed, while the number of entries in each entity depends on the meta-model size and the synonyms defined in the NL configuration.

3.2 Chatbot generation: Execution model

The generated chatbot also contains actions, required to perform the query on a modelling backend, which we call the *execution model*. This execution model contains a set of *execution rules* that bind user intentions to response actions as part of the chatbot behaviour definition (cf. label 4 in Fig. 2). For each intent in the *Intent* model, we generate the corresponding execution rule in the execution model using an event-based language that receives as input the recognized intent together with the set of parameter values matched by the NL engine during the analysis and classification of the user utterance.

All the execution rules follow the same process: the matched intent and the parameters are used to build an OCL-like query to collect the set of objects the user wants to retrieve. The intent determines the type of query to perform (e.g., allInstances, select, etc.), while the parameters identify the query parameters, predicates, and their composition. The query computation is delegated to the underlying modelling platform (see next section), and the returned model elements are processed to build a human-readable message that is finally posted to the user by the bot engine.

As an example, Listing 1 shows the execution rule that handles an allInstances operation. The class to obtain the instances of is retrieved from the context variable (available in every execution rule) and passed to our EMF Platform, which performs the query. Next, the instances variable holding the results is processed to produce a readable string (in this case a list of names), and the Chat Platform is called to reply to the user.

```
on intent GetAllInstances do
    val Map<String, Object> collectionContext = context.get(”collection”)
    val instances = EMFPlatform.GetAllInstances( collectionContext.get(”class”) as String )
    val resultString = instances.map[name].join(”, ”)
    ChatPlatform.Reply(”I found the following results” + resultString)
```

Listing 1. Execution rule example

4 Proof of Concept

As a proof of concept, we have created a prototype that produces Xatkit-based chatbots [4], following the two phases depicted in Fig. 3. Xatkit is a model-driven solution to define and execute chatbots, which offers DSLs to define the bot intents, entities and actions. The execution of such chatbots relies on the Xatkit *runtime* engine. At its core, the engine is a Java library that implements all the execution logic available in the chatbot DSLs. Besides, a connector with Google's DialogFlow engine [6] takes care of matching the user utterances, and a number of platform components enable the communication between Xatkit and other external services.

In the context of this paper, we have developed a new EMF Platform that allows Xatkit to query EMF models in response to matched intents. The first version of our prototype platform[6] provides actions to retrieve all the instances of a given class, and filter them based on a composition of boolean predicates on the object's attributes or references. These predicates are retrieved from the context parameter defined in the intents (see Section 3.1), and mapped to Java operations (e.g., the StringComparison "contains" is translated into ((String)value).contains(otherValue). The query result is returned as a list of EObjects, which is processed using the bot expression language to produce the response message. Listing 1 showed an example of the use of this EMF Platform.

We have also developed a web application, where domain meta-models (in .ecore format) can be uploaded, and then (optionally) configured with synonyms. Once the configuration is finished, the application synthesizes a Xatkit chatbot model, which then can be executed using the Xatkit runtime engine.

Fig. 5(a) shows the web application on the left, where the running example meta-model (cf. Fig. 2) is being configured. Fig. 5(b) shows a moment in the execution of the generated Xatkit chatbot, and the result returned by the bot when processing the example utterance *"show all accidents with more than one injury"*.

5 Related Work

Next, we review approaches to the synthesis of chatbots for modelling or data query.

Our work relies on NL as a kind of concrete syntax for DSLs [17]. NLP has been used within Software Engineering to derive UML diagrams/domain models from text [2,11]. However, the opposite direction (i.e., generating chatbots from domain models) is largely unexplored. Almost no chatbot platform supports automatic chatbot generation from external data sources. A relevant exception is Microsoft QnA Maker [14], which generates bots for the Azure platform from FAQs and other well-structured textual information.

[6] https://github.com/xatkit-bot-platform/xatkit-emf-platform

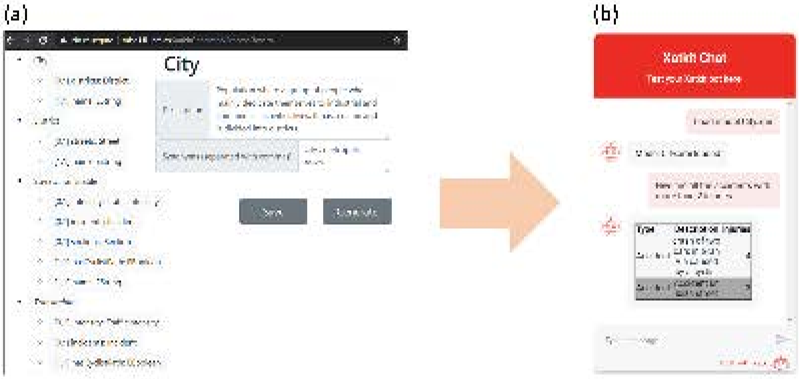


Fig. 5. (a) Web application to configure the chatbot. (b) A query in the generated chatbot.

Closest approaches to ours are tools like ModelByVoice [13] and VoiceToModel [20], which offer some predefined commands to create model elements for specific types of models. In contrast, our framework targets model queries and not model creation, which was pursued in our previous work [17]. None of those two approaches support queries. Castaldo and collaborators [3] propose generating chatbots for data exploration in relational databases, but requiring an annotated schema as starting point, while in our case providing synonyms is an optional step. Similarly, [19] integrates chatbots to service systems by annotating and linking the chatbot definition to the service models. In both cases, annotations and links must be manually created by the chatbot designer to generate the conversational elements. In contrast, our approach is fully automatic. In [22], chatbots are generated from OpenAPI specifications but the goal of such chatbots is helping the user in identifying the right API Endpoint, not answering user queries.

Altogether, to our knowledge there are no automatic approaches to the generation of flexible chatbots with model query capabilities. We believe that applying classical concepts from CRUD-like generators to the chatbot domain is a highly novel solution to add a conversational interface to any modelling language.

6 Conclusion

Conversational interfaces are becoming increasingly popular to access all kind of services, but their construction is challenging. To remedy this situation, we have proposed the automatic synthesis of chatbots able to query the instances of a domain meta-model.

In the future, we aim to support more complex queries, including the conversational and reactive ones mentioned in Section 2.1. Our approach could be used to query other types of data sources (e.g., databases or APIs) via an initial reverse engineering step to build their internal data model and translate the NL query into the query language of the platform. Finally, we would like to add access control on top of the bot definition to ensure users cannot explore parts of the model/system unless they have permission.

Acknowledgments. Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00 and TIN2016-75944-R) and the R&D programme of Madrid (P2018/TCS-4314).

References

1. Amazon: Amazon lex. `https://aws.amazon.com/lex/` (2019)
2. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F.: Extracting domain models from natural-language requirements: approach and industrial evaluation. In: Proc. MoDELS. pp. 250–260. ACM (2016)
3. Castaldo, N., Daniel, F., Matera, M., Zaccaria, V.: Conversational data exploration. In: Proc. ICWE. pp. 490–497 (2019)
4. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: A multimodal low-code chatbot development framework. IEEE Access 8, 15332–15346 (2020)
5. Erlenhov, L., de Oliveira Neto, F.G., Scandariato, R., Leitner, P.: Current and future bots in software development. In: Proc. BotSE@ICSE. pp. 7–11. IEEE / ACM (2019)
6. Google: DialogFlow. `https://dialogflow.com/` (2019)
7. IBM Watson Assistant: `https://www.ibm.com/cloud/watson-assistant/` (2019)
8. Jackson, P., Moulinier, I.: Natural language processing for online applications: Text retrieval, extraction and categorization, vol. 5. John Benjamins Publishing (2007)
9. Kerlyl, A., Hall, P., Bull, S.: Bringing chatbots into education: Towards natural language negotiation of open learner models. In: Applications and Innovations in Intelligent Systems XIV, pp. 179–192. Springer (2007)
10. Klopfenstein, L., Delpriori, S., Malatini, S., Bogliolo, A.: The rise of bots: a survey of conversational interfaces, patterns, and paradigms. In: Proc. DIS. pp. 555–565. ACM (2017)
11. Landhäußer, M., Körner, S.J., Tichy, W.F.: From requirements to UML models and back: How automatic processing of text can support requirements engineering. Software Quality Journal 22(1), 121–149 (2014)
12. Lebeuf, C., Storey, M.D., Zagalsky, A.: Software bots. IEEE Software 35(1), 18–23 (2018)
13. Lopes, J., Cambeiro, J., Amaral, V.: ModelByVoice - towards a general purpose model editor for blind people. In: Proc. MODELS Workshops. CEUR Workshop Proceedings, vol. 2245, pp. 762–769. CEUR-WS.org (2018)
14. Microsoft: QnA Maker. `https://www.qnamaker.ai/` (2019)
15. OCL: `http://www.omg.org/spec/OCL/` (2014)
16. Pérez-Soler, S., Guerra, E., de Lara, J.: Collaborative modeling and group decision making using chatbots in social networks. IEEE Software 35(6), 48–54 (2018)
17. Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J.: Towards conversational syntax for domain-specific languages using chatbots. JOT 18(2), 5:1–21 (2019)
18. Shawar, A., Atwell, E., Roberts, A.: Faqchat as in Information Retrieval System. In: Proc. LTC. pp. 274–278. Poznań: Wydawnictwo Poznańskie (2005)
19. Sindhgatta, R., Barros, A., Nili, A.: Modeling conversational agents for service systems. In: On the Move to Meaningful Internet Systems, Proc. OTM. pp. 552–560 (2019)
20. Soares, F., Araújo, J., Wanderley, F.: VoiceToModel: an approach to generate requirements models from speech recognition mechanisms. In: Proc. SAC. pp. 1350–1357. ACM (2015)
21. Thomas, N.: An e-business chatbot using AIML and LSA. In: Proc. ICACCI. pp. 2740–2742. IEEE (2016)
22. Vaziri, M., Mandel, L., Shinnar, A., Siméon, J., Hirzel, M.: Generating chat bots from web API specifications. In: Proc. ACM SIGPLAN Onward! pp. 44–57 (2017)
23. Xu, A., Liu, Z., Guo, Y., Sinha, V., Akkiraju, R.: A new chatbot for customer service on social media. In: Proc. CHI. pp. 3506–3510. ACM (2017)

B.5 Model-driven chatbot development

Title	Model driven chatbot development.	
Publication	39th International Conference on Conceptual Modelling (ER'2020)	
Core	2020 Core A	
Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1007/978-3-030-62522-1_15	
Date	November 2020	

Abstract

Chatbots are software services accessed via conversation in natural language. They are increasingly used to help in all kinds of procedures like booking flights, querying visa information or assigning tasks to developers. They can be embedded in webs and social networks, and be used from mobile devices without installing dedicated apps. While many frameworks and platforms have emerged for their development, identifying the most appropriate one for building a particular chatbot requires a high investment of time. Moreover, some of them are closed – resulting in customer lock-in – or require deep technical knowledge.

To tackle these issues, we propose a model-driven engineering approach to chatbot development. It comprises a neutral meta-model and a domain-specific language (DSL) for chatbot description; code generators and parsers for several chatbot platforms; and a platform recommender. Our approach supports forward and reverse engineering, and model-based analysis. We demonstrate its feasibility presenting a prototype tool and an evaluation based on migrating third party Dialogflow bots to Rasa.

Model-driven chatbot development

Sara Pérez-Soler[0000-0002-4558-7111], Esther Guerra[0000-0002-2818-2278], and Juan de Lara[0000-0001-9425-6362]

Universidad Autónoma de Madrid (Spain)
(Sara.PerezS, Esther.Guerra, Juan.deLara)@uam.es

Abstract. Chatbots are software services accessed via conversation in natural language. They are increasingly used to help in all kinds of procedures like booking flights, querying visa information or assigning tasks to developers. They can be embedded in webs and social networks, and be used from mobile devices without installing dedicated apps. While many frameworks and platforms have emerged for their development, identifying the most appropriate one for building a particular chatbot requires a high investment of time. Moreover, some of them are closed – resulting in customer lock-in – or require deep technical knowledge.
To tackle these issues, we propose a model-driven engineering approach to chatbot development. It comprises a neutral meta-model and a domain-specific language (DSL) for chatbot description; code generators and parsers for several chatbot platforms; and a platform recommender. Our approach supports forward and reverse engineering, and model-based analysis. We demonstrate its feasibility presenting a prototype tool and an evaluation based on migrating third party Dialogflow bots to Rasa.

Keywords: Chatbots · Model-Driven Engineering · DSLs · Migration

1 Introduction

Chatbots are software programs that interact with users via natural language (NL) conversation. Their use is booming because they can be used within webs and social networks – like Telegram, Twitter or Slack – without having to install dedicated apps [23]. Many companies are developing chatbots to offer 24/7 customer service while reducing costs, and their presence is percolating a wide range of areas such as education [26, 29, 30] or civic engagement [27].

The success of chatbots has led to the emergence of a plethora of technologies for their creation. Not only big software companies have made available chatbot creation tools, like Google's Dialogflow [9], IBM's Watson Assistant [28], Microsoft's bot framework [17] or Amazon's Lex [15], but many other proposals exist, like Rasa [21], FlowXO [10] and Pandorabots [18]. Among them, we find a variety of approaches. For example, Dialogflow and Watson offer low-code cloud development platforms that support the creation and deployment of bots, while Rasa is a framework that requires Python programming for bot development.

Overall, these chatbot creation tools are indisputably powerful (e.g., some provide NL processing, speech recognition, etc.). However, since there are so many options, choosing the most appropriate one to develop a chatbot with certain features is not easy. There may also be operational factors to consider in the decision, as for example, some options may imply vendor lock-in, and migrating chatbots between tools is not generally supported. Last but not least, some approaches have a steep learning curve and require expert knowledge.

To overcome these problems, we propose a model-driven engineering (MDE) approach [22] to chatbot development. This relies on a meta-model with core primitives for chatbot design, and a domain-specific language (DSL) to define bots independently of the implementation technology. Chatbots defined with the DSL can be analysed for "smells" of defects, and a ranked list of appropriate bot creation tools is recommended based on the chatbot definition and other requirements. Our DSL can be used for *forward engineering*, to produce the chatbot implementation from its specification; and for *reverse engineering*, to produce a model out of a chatbot implementation, which can then be analysed, refactored and migrated to other platforms. Currently, we provide code generators and parsers from/to Dialogflow and Rasa, but our architecture is extensible. We evaluate our approach migrating third-party Dialogflow chatbots to Rasa.

In the rest of the paper, Section 2 introduces chatbot design and motivates our work. Section 3 outlines our proposal. Section 4 describes the meta-model and the DSL. Section 5 details our platform recommender. Section 6 presents tool support. Section 7 reports an evaluation based on migration. Section 8 compares with related works, and Section 9 concludes and outlines future work.

2 Building a chatbot: background and limitations

Chatbots (also called cor
versational user interface
converse on any topic wit
(e.g., bookings flights or

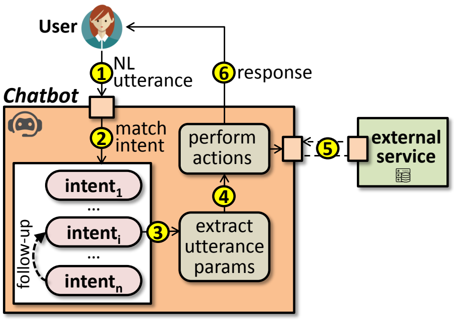


Fig. 1: Chatbot working scheme.

Fig. 1 shows the typical working scheme of task-specific chatbots. They are designed around a set of *intents* that users may want to accomplish. Given a user utterance (e.g., *"I'd like to buy a flight ticket from Madrid to Vienna"*, label 1 in the figure), the chatbot tries to identify the corresponding intent (label 2). The approach for this depends on the particular chatbot creation tool. Some of them – like Pandorabots – permit defining patterns or regular expressions upon which the utterance is matched, while others – like Dialogflow, Lex or Rasa – require declaring training phrases and apply NL processing (NLP) techniques. If the chatbot does not find any matching intent, some approaches allow having a default fallback intent. In addition, the conversation flow can be structured into expected sequences of intents (relation *follow-up* in the figure).

After matching an intent, the chatbot extracts the *parameters* of interest from the utterance (e.g., the origin and destination of the flight, label 3). Parameters may be typed by *entities*, which can be either predefined (e.g., date, number) or specific to a chatbot (e.g., flight class). If the utterance lacks some expected parameters (e.g., date of flight), the chatbot can be configured to ask for them.

As a last step, the chatbot can perform different actions depending on the intent, such as calling an external service (e.g., a booking information system, label 5) or replying to the user (label 6). The simplest response format is text, but some chatbot deployment platforms (e.g., Telegram, Twitter) also support images, URLs, videos or buttons.

There are numerous tools for creating chatbots that follow this scheme. These tools use different approaches, ranging from low-code form-based platforms (e.g., Dialogflow, Lex, Watson, FlowXO) to frameworks for programming languages (e.g., Rasa, Botkit [4]), libraries (e.g., Chatterbot [6]) and services (e.g., LUIS [16]). Such a variety makes it difficult to ascertain which tool is suitable to build a specific chatbot, as not every tool supports every possible feature (e.g., only a few provide NLP or multi-language support). Moreover, the conceptual model of the chatbot might be difficult to attain, as the chatbot definition frequently includes tool-specific accidental details. As a consequence, reasoning, understanding, validating and testing chatbots independently from the implementation technology becomes challenging. Finally, some platforms are proprietary which hinders chatbot migration and results in vendor lock-in.

In the following section, we present our proposal to overcome these problems.

3 Model-driven engineering of chatbots

Fig. 2 shows a scheme of our proposal. It provides a technology-agnostic DSL called Conga (ChatbOt modelliNg lanGuAge) to design chatbots. This is built on the basis of a neutral, platform-independent meta-model resulting from an analysis of the existing approaches. The DSL permits modelling chatbots independently of any development platform, and validating quality criteria and well-formedness rules on the chatbot models. Section 4 introduces this DSL.

To facilitate the task of selecting a development tool for implementing a given chatbot model, we provide an extensible recommender that analyses the chatbot model as well as other requirements, to provide a ranked list of suitable tools. Section 5 explains the recommender system and its extensible architecture.

In addition, the DSL is complemented with code generators that synthesize chatbot implementations from chatbot models for specific development tools (e.g., JSON configuration files in the case of Dialogflow, or Python programs and configuration files in the case of Rasa). The chatbots so generated can be deployed in different platforms (e.g., Telegram, Slack or Twitter) to make them available to users. Likewise, the DSL facilitates chatbot migration by the provision of parsers from several development platforms into the DSL. Our tool

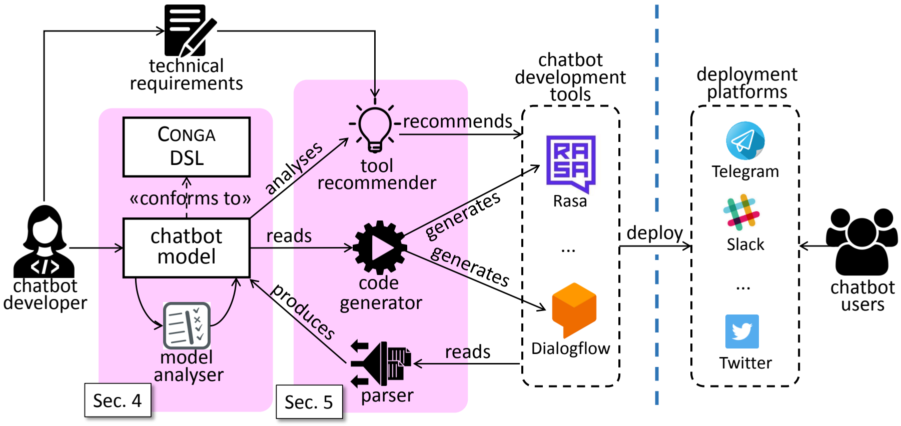


Fig. 2: Overview of our proposal.

support for these scenarios is explained in Section 6, while its evaluation based on migration scenarios is presented in Section 7.

Overall, the advantages of our proposal are the following: it keeps the design of the chatbot independent of the specific development technology; it provides analyses applicable at the design level (i.e., prior to the implementation); it assists in the selection of an appropriate development tool; it enables both forward and backward engineering; and it reduces the risk of vendor lock-in.

4 Conga: a DSL for chatbot design

Our DSL Conga enables the design of chatbots conformant to the neutral meta-model of Fig. 3. This is a platform-independent meta-model which gathers recurrent concepts in chatbot development approaches. Table 1 summarizes the main concepts of the 15 approaches that we have revised to design our meta-model.

The main meta-model class is *Chatbot*, which has a *name* and a list of supported *languages* to allow the definition of multi-language chatbots. Chatbots can define *intents*, *entities*, *actions* and structure the dialogue via *flows*.

Most analysed approaches (10 out of 15) rely on the notion of intent. In our meta-model, an *Intent* has a *name*, can be a *fallback* intent, and defines one set of regular expressions or NL training phrases per supported language. As Table 1 shows (3^{rd} and 4^{th} columns), all approaches support at least one of these two definition mechanisms, while 6 approaches can combine regular expressions with NL phrases. An example of a training phrase in English to query the price of a cake can be *"How much does a chocolate cake cost?"*.

Intents may need to collect information, like the cake flavour in the previous sentence. This information is stored in *Parameter*s, which most approaches support (see 5^{th} column of Table 1). In our meta-model, *Parameter*s have a *name*, a *type*, can be a *list*, can be *required*, and may define a list of *prompts* to

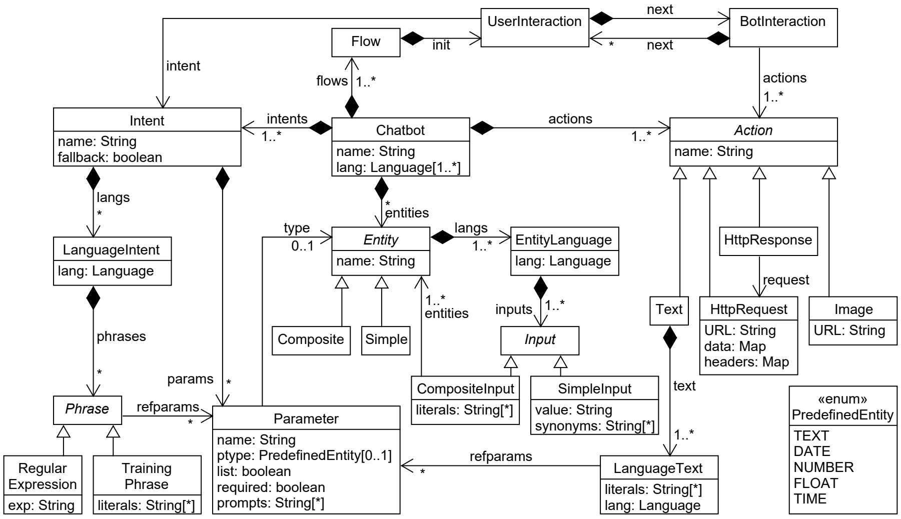


Fig. 3: Platform-independent chatbot design meta-model (simplified excerpt).

Table 1: Recurrent concepts of representative chatbot creation approaches.

Approach	Intent	NLP	Regular expr	Phrase params	Entities	Answ. Text	Answ. Image	Http Rq/Rs	Dialogue structure
Botkit [4]	no	no	yes	no	no	no	no	no	programm.
Bot framework [17]	yes	yes	yes	yes	yes	yes	yes	yes	tree
Chatfuel [5]	no	yes	no	no	no	yes	yes	yes	tree
Chatterbot [6]	no	yes	no	no	yes	no	no	no	context
Dialogflow [9]	yes	yes	no	yes	yes	yes	yes	yes	context
FlowXO [10]	yes	no	yes	no	yes	yes	yes	yes	tree
Landbot.io [14]	no	no	yes	no	no	yes	yes	yes	tree
Lex [15]	yes	yes	no	yes	yes	yes	yes	yes	session
LUIS [16]	yes	yes	yes	yes	yes	no	no	no	tree
Pandorabots [18]	yes	no	yes	yes	no	yes	yes	no	DSL
Rasa [21]	yes	yes	no	yes	yes	yes	yes	yes	tree
SmartLoop [24]	yes	yes	yes	yes	yes	yes	yes	no	context
Watson [28]	yes	yes	yes	yes	yes	yes	yes	yes	context
Xatkit [8]	yes	yes	yes	yes	yes	yes	yes	yes	context
Xenioo [31]	no	yes	yes	yes	yes	yes	yes	no	tree

ask for a value when the parameter is required but the user utterance does not include its value. Parameters are typed by entities (6^{th} column in the table). Our meta-model supports both predefined entities (enumeration *PredefinedEntity* with values *text*, *date*, *number*, *float* and *time*) and chatbot-specific ones (class *Entity*).

Chatbot-specific entities can be *Simple* entities, defined as a list of words with their synonyms, or *Composite* entities, made of other entities and text. For example, in our bakery example, we may define simple entities for the products (cake, cupcake, biscuit...) and flavours (chocolate, strawberry, vanilla...), and a composite entity combining both (⟨product⟩ with ⟨flavour⟩ flavour, ⟨flavour⟩ ⟨product⟩, ⟨flavour⟩ flavoured ⟨product⟩...).

Chatbots can perform different *Action*s. The most common ones are the following (see 7^{th} to 9^{th} columns in Table 1): sending a *Text* response to the user, which requires specifying the actual text for each chatbot language; sending an *Image* which is identified by its *URL*; performing an *HttpRequest* to a given *URL*, optionally providing some *headers* and *data*; and sending to the user an *HttpResponse* for a previous *http* request.

Finally, a chatbot can define conversation *Flow*s. As the last column of Table 1 shows, all approaches provide some way to structure the dialogue, and in particular, the meta-model has primitives to cover conversation trees and intent activation based on contexts and sessions. Pandorabots supports a richer mechanism based on a DSL – the Artificial Intelligence Markup Language (AIML)[1] – which our meta-model does not include due to its specificity. A flow is made of *UserInteraction*s associated to an intent, and *BotInteraction*s comprising one or more actions. A flow must start with a user interaction followed by a bot interaction, after which there may be other user interactions, and so on.

To facilitate the instantiation of this meta-model, we have designed a textual concrete syntax for it. Listing 1 illustrates its usage by showing an excerpt of the definition of a chatbot for a bakery to which users can consult prices and order different products like bread or cakes. The first line defines the chatbot name and the supported languages (English and Spanish). Lines 4–18 define an intent named *Price*, which declares a set of training phrases for each language of the chatbot. If a set of phrases does not specify a language (as is the case in line 5), then they are assumed to be in the first language declared by the chatbot (English in this example). The intent defines four parameters in lines 15–18. The training phrases can refer to them (e.g., [count_param] in line 6) and assign them a value in the context of the phrase (e.g., three in line 6). The parameters type can be a predefined entity, like *number*, or a user-defined one, like *flavour*.

Lines 21–29 show the definition of the simple entity *flavour*. This declares the admissible flavours for each language supported by the chatbot, together with their synonyms.

Lines 31–42 illustrate the definition of actions, specifically, a text response called *PriceResponse*. As in the training phrases, text responses can be in different languages, and use parameter values (e.g., [Price.bread_param] in line 34).

Finally, lines 44–49 define the conversation flow (i.e., sequences of user and chatbot interactions). The listing configures two flows, which always must start with a user interaction and the corresponding intent. Flows are defined once, independently of the language. The flow in line 45 takes place when the user utterance matches the *Price* intent, in which case, the chatbot performs the action *PriceResponse* defined in lines 32–42. The second flow (lines 46–49) corresponds to the intent *Buy*. In this case, the chatbot asks for the product type to buy, and the flow is split depending on the user answer (cake or bread). This branching can be recursively nested to enable a compact representation of alternative flows.

The DSL includes model validation rules of two kinds. The first ones are *integrity constraints* that ensure the well-formedness of chatbot models. For ex-

[1] http://www.aiml.foundation/

ample, some of these rules forbid equally named elements (e.g., two *Action*s with the same name) and validate that each *Intent* has exactly one *LanguageIntent* for each language of the chatbot (attribute *Chatbot.lang*). The second kind of rules performs a static analysis of the chatbot definition to assess whether it adheres to *best practices* for chatbot design. Violating these rules may be a "smell" of a bad chatbot design. Currently, the DSL validates the following aspects: there is a fallback intent; text responses only use parameters of intents appearing in the conversation flow; there are no two intents with the same training phrase; all intents define either one regular expression or at least three training phrases; and training phrases do not start by a parameter typed by the predefined entity *text*, as this would match any user utterance which can be problematic.

```
chatbot Bakery language: en, es

intents:
  Price:
    inputs {
      "How much are" (three)[count_param] (bread)[bread_param] "?",
      "How much is a" (cake)[cake_param] "?",
      "How much is a" (chocolate)[flavour_param](cake)[cake_param] "?"
    }
    inputs in es {
      "¿Cuanto cuesta el" (pan)[bread_param] "?",
      "¿Cuanto cuesta una" (tarta)[cake_param] "?",
      "¿Cuanto cuesta un" (pastel)[cake_param] "de" (chocolate)[flavour_param] "?"
    }
    parameters:
      bread_param, cake_param: entity product;
      flavour_param: entity flavour;
      count_param: entity number;

entities:
  simple entity flavour:
    inputs in en {
      chocolate synonyms choco, cocoa, truffle;
      ...
    }
    inputs in es {
      chocolate synonyms choco, cacao, trufa;
      ...
    }

actions:
  text response PriceResponse:
    inputs {
      "The" [Price.bread_param] "costs 1 euro per unit",
      "The" [Price.flavour_param] [Price.cake_param] "costs 10 euro per unit",
      "The" [Price.cake_param] "costs 10 euro per unit"
    }
    inputs in es {
      "El" [Price.bread_param] "cuesta 1 euro por unidad",
      "Las" [Price.cake_param] "de" [Price.flavour_param] "cuestan 10 euros por unidad",
      "Las" [Price.cake_param] "cuestan 10 euros por unidad"
    }

flows:
  − user Price => chatbot PriceResponse;
  − user Buy => chatbot Type {
    => user Cake => chatbot Quantity => user num => chatbot BuyCakeHttp, buyCakeResponse;
    => user Bread => chatbot Quantity => user num => chatbot BuyBreadHttp, buyBreadResponse;
  }
```

Listing 1: Excerpt of chatbot model definition with the CONGA DSL.

5 Recommending a chatbot creation tool

Due to the large amount of tools and approaches for chatbot creation (cf. Table 1), selecting the best option to build a particular chatbot becomes complex. To assist in this task, we provide a recommender that receives a chatbot model specified with Conga and the answers to a questionnaire relative to other aspects of the chatbot (e.g., technical, organizational or managerial requirements), and from this information, it recommends an appropriate tool to implement the chatbot. The recommender builds on a model-based extensible architecture that enables the addition of new chatbot creation tools and the customization of the questions and model features the recommendation builds on.

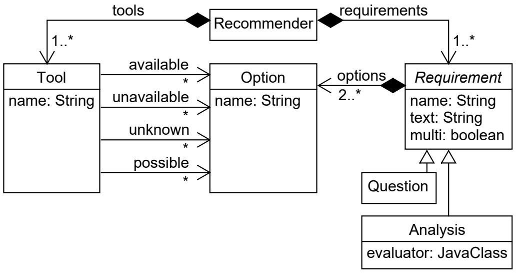


Fig. 4: Recommender meta-model.

Fig. 4 shows the meta-model our *Recommender* relies on. To make a recommendation, it considers a list of chatbot *Requirements*, whose value can be retrieved either by means of a *Question* to the developer, or automatically via an *Analysis* of the chatbot model. Both kinds of requirements have a *name*, a *text*, a list of admissible *Option*s, and can be *multi-response* or not. In addition, *Analysis* requirements define an *evaluator*, which is the (Java) class in charge of analysing the chatbot model. This latter class must extend the built-in abstract class *Evaluator* and implement its abstract method *evaluate*, which receives a chatbot model and returns the *Option*s that this model fulfils. The recommendation consists of a list of *Tools*. For each tool, the recommender stores the requirement options that are *available*, *unavailable*, *unknown* or are ultimately *possible* (i.e., not natively supported but achievable using a workaround).

The recommender currently considers the requirements in Table 2, and new ones can be added if needed. The table also shows the coverage of these requirements by two chatbot creation tools: Dialogflow and Rasa. Regarding analysis requirements, we check whether the chatbot model is multi-language (like in Listing 1), the targeted languages[2], and whether it uses predefined or chatbot-specific entities, calls to external services, parameters, training phrases or regular expressions. Rasa does not support multi-language bots, but a workaround is generating one bot per language, hence the value *possible* in the table.

Questions are chatbot requirements explicitly asked to the developer as they cannot be inferred from the chatbot model. The first seven questions in Table 2 deal with technical aspects. Specifically, we ask for the following issues: the social network the chatbot is to be deployed in (Dialogflow supports 16, and Rasa 8); the hosting server of the chatbot, since some platforms (e.g., Dialogflow) can host the chatbot themselves, but others (e.g., Rasa) require an external server; the level of support for version control, which is built-in in platforms like Di-

[2] For brevity, Table 2 shows the number of languages supported, not the list of them.

Table 2: Requirements that the recommender currently takes into consideration.

Text	**Multi-response**	**Options**	**Dialogflow**	**Rasa**
Analyses				
Is the chatbot multi-language?	false	Yes	avail.	possib.
		No	avail.	avail.
Which are the chatbot languages?	true	-	21	all
Does the chatbot use new or predefined entities?	true	Predefined	avail.	avail.
		New entities	avail.	avail.
		None	avail.	avail.
Does the chatbot call to external services?	false	One	avail.	avail.
		Multiple	possib.	avail.
		None	avail.	avail.
Does the chatbot use phrase parameters?	false	Yes	avail.	avail.
		No	avail.	avail.
Does the chatbot need persistent or volatile parameter storage?	true	Persistent	avail.	avail.
		Volatile	avail.	avail.
		None	avail.	avail.
Does your chatbot need natural language processing or pattern matching?	true	NLP	avail.	avail.
		Pattern	unavail.	unavail.
Questions				
Which social networks do you want to deploy the chatbot in?	true	-	16	8
Do you want to deploy the chatbot on your own host?	false	Tool host	avail.	unavail.
		Own host	unavail.	avail.
Do you want to use a built-in version control system?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require native support for chatbot analytics?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require native support for utterance persistence?	false	Yes	avail.	avail.
		No	avail.	avail.
Do you require the chatbot to support speech recognition?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require the chatbot to support sentiment analysis?	false	Yes	avail.	unavail.
		No	avail.	avail.
Do you require to use an open-source tool?	false	Yes	unavail.	avail.
		No	avail.	avail.
Which price model do you plan to use?	true	Free	avail.	avail.
		Pay as you go	avail.	unavail.
		Quota	unavail.	unavail.
		Pay advanced feats.	unavail.	avail.
What's the level of expertise of the development team?	false	Low	avail.	unavail.
		High	avail.	avail.

alogflow, while programming-based approaches like Rasa need to use an external version control system like *github*; the need to monitor the chatbot performance (e.g., Dialogflow provides some chatbot analytics); the persistence of utterances for their subsequent analysis; and the need to support speech recognition or sentiment analysis.

The last three questions in Table 2 tackle organizational and managerial aspects concerned with open-source and price model requirements, and the level of expertise of the development team. For example, the expertise for using Rasa is higher than for Dialogflow, since the former requires programming.

Since some requirements may be more important than others depending on the project, we assign an importance level to each requirement, which the developer can customize. The supported levels are: *irrelevant*, *relevant*, *double relevant* and *critical*. Irrelevant requirements are not considered for the recommendation, and critical ones are breaking factors (i.e., tools that do not comply with the requirement will not be recommended). For each tool, the recommender computes a score based on the supported requirements and their importance level. *Available* requirements add 1 to the score of a tool, *unavailable* ones add 0, *unknown* ones add 0.5, and *possible* ones add 0.75. In all cases, double relevant requirements score double. Then, the recommender orders the tools according to their score, and produces a report with the ranking of tools and how each requirement contributes to this ranking.

Incorporating a new chatbot creation tool (e.g., Watson) into our framework requires: (i) informing the tool options for every requirement in the recommender; (ii) providing a code generator from Conga to the tool; (iii) optionally, providing a parser if reverse engineering is required. Our framework prevents the code generation for a tool whenever the chatbot requirements are *unavailable* in that tool. There may be some *possible* requirements though, meaning that their support is not native in the tool but they can be implemented. For instance, Rasa does not support multi-language chatbots, but this can be emulated by generating one chatbot per language. As another example, Dialogflow only supports one external service call per intent, and so, the generator only considers the first call and warns the developer.

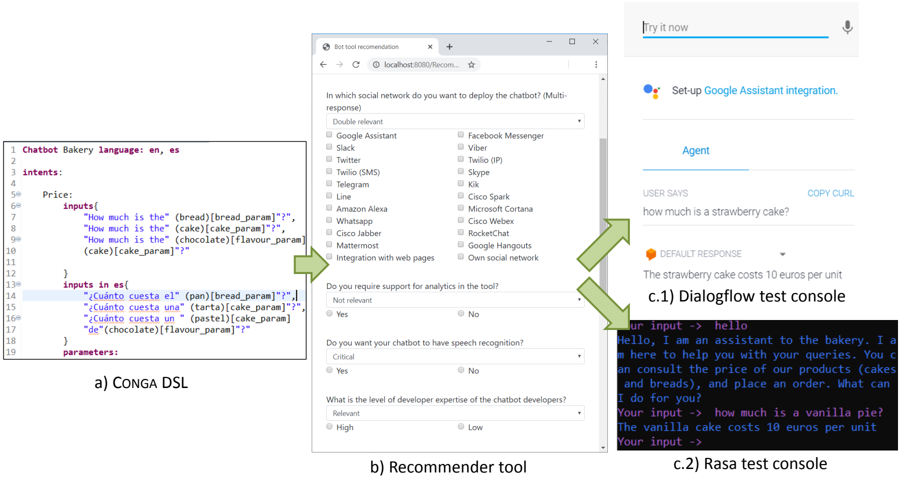
6 Tool support

We have built tool support for our approach. Fig. 5(a) shows the developed editor for the Conga DSL, which uses the Eclipse Modeling Framework (EMF) [25] and Xtext. The editor provides syntax highlighting, autocompletion, and informs of errors and warnings found in the chatbot models.

Upon uploading a chatbot model to a web server, we can apply the recommender (Fig. 5(b)) and generate code for a specific chatbot creation tool. Currently, the recommender considers 14 up-to-date tools, and we provide generators and parsers from/to Dialogflow and Rasa. Anyhow, as previously explained, both aspects are extensible. Figs. 5(c.1) and 5(c.2) show two generated chatbots for Dialogflow and Rasa in their respective development environments, from where the chatbots can be deployed into a social network.

7 Evaluation

This section reports on an evaluation of our approach on a migration scenario which involves both backward and forward engineering. The goal is to answer two research questions (RQs): **RQ1**: *Is* Conga *expressive enough to capture the details of existing chatbots?* **RQ2**: *Can the migration process be fully automated?* For this purpose, we have migrated four Dialogflow agents developed by third parties (three from github, one built by Google) into Rasa. Table 3 summarizes the experiment results.



```
Chatbot Bakery language: en, es

intents:

    Price:
        inputs{
            "How much is the" (bread)[bread_param]"?",
            "How much is the" (cake)[cake_param]"?",
            "How much is the" (chocolate)[flavour_param]
            (cake)[cake_param]"?"

        }
        inputs in es{
            "¿Cuánto cuesta el" (pan)[bread_param]"?",
            "¿Cuánto cuesta una" (tarta)[cake_param]"?",
            "¿Cuánto cuesta un " (pastel)[cake_param]
            "de"(chocolate)[flavour_param]"?"
        }
        parameters:
```

a) CONGA DSL

In which social network do you want to deploy the chatbot? (Multi-response)

Double relevant

□ Google Assistant	□ Facebook Messenger
□ Slack	□ Viber
□ Twitter	□ Twilio (IP)
□ Twilio (SMS)	□ Skype
□ Telegram	□ Kik
□ Line	□ Cisco Spark
□ Amazon Alexa	□ Microsoft Cortana
□ Whatsapp	□ Cisco Webex
□ Cisco Jabber	□ RocketChat
□ Mattermost	□ Google Hangouts
□ Integration with web pages	□ Own social network

Do you require support for analytics in the tool?

Not relevant

○ Yes ○ No

Do you want your chatbot to have speech recognition?

Critical

○ Yes ○ No

What is the level of developer expertise of the chatbot developers?

Relevant

○ High ○ Low

b) Recommender tool

USER SAYS

how much is a strawberry cake?

DEFAULT RESPONSE

The strawberry cake costs 10 euros per unit

c.1) Dialogflow test console

```
ur input ->  hello
Hello, I am an assistant to the bakery. I a
m here to help you with your queries. You c
an consult the price of our products (cakes
 and breads), and place an order. What can
I do for you?
Your input ->  how much is a vanilla pie?
The vanilla cake costs 10 euros per unit
Your input ->
```

c.2) Rasa test console

Fig. 5: Our tool in action for forward engineering. (a) CONGA editor. (b) Recommender. (c.1) Generated bot for Dialogflow. (c.2) Generated bot for Rasa.

Table 3: Assessment metrics.

	Dialogflow					CONGA		**Rasa**			
	No. intents	No. ents.	Http req.	No. files	Lang	No. objects	No. lines	No. chatbots	No. Python lines	No. Markd. lines	No. yaml lines
Game	11	0	yes	30	en/fr	541	268	2	378	242	362
Room reservation	7	1	no	17	en	717	196	1	253	166	137
Coffee shop	21	8	no	60	en	931	393	1	657	394	269
Nutrition	4	7	no	23	en	833	610	1	802	81	99

Game[3] is a conversational agent for a numeric guessing game. It has 11 intents, no entities, one *http* request, and supports English and French. Its Dialogflow specification is made of 30 JSON files. From this specification, our parser creates a model with 541 objects and 268 lines of CONGA code. Since Rasa does not support multi-language chatbots, two Rasa chatbots are generated from the CONGA model, one for each language. These have 378 lines of Python code (to define parameters and actions), 242 lines of Markdown code (to define intents and flows) and 362 lines of YAML code (to configure the chatbot).

Room reservation[4] is a chatbot to book hotel rooms. It has 7 intents and one entity, and works in English. The migration produces a Rasa chatbot with 253 lines of Python code. Since the original Dialogflow chatbot has button actions, which are unsupported by CONGA, we had to add them manually in Rasa.

Coffee shop is a Dialogflow pre-built agent to order food to a coffee shop. Its specification is the most complex of the four chatbots, spanning 60 JSON files. These are parsed into a CONGA model with 931 objects.

[3] https://github.com/actions-on-google/dialogflow-number-genie-nodejs

[4] https://github.com/dialogflow/dialogflow-java-client-v2/tree/master/samples/resources

Nutrition[5] is a chatbot to query the nutritional value of meals. Although it is a small chatbot with 4 intents and 7 entities, it generates many lines of Python code because the entities have many entries.

Overall, we were able to migrate all Dialogflow chatbots but the button actions on the room reservation bot, which confirms the expressiveness of CONGA (RQ1). Except for that bot, migration was fully automatic (RQ2). These results are very promising, but more case studies are needed to strengthen the confidence in the capabilities of CONGA. Moreover, we manually checked that the produced Rasa chatbots preserved the original Dialogflow behaviour, but we plan to automate this check in future work (e.g., using tools like Botium[6]).

8 Related work

The popularity of chatbots has promoted the appearance of many tools for their construction. In this section, we revise works built atop these tools to simplify some aspect of chatbot development.

Xatkit [8] (formerly known as Jarvis [7]) is a model-driven solution for developing chatbots. Similar to our approach, it proposes a meta-model and a textual DSL. However, differently from us, Xatkit has its own bot execution engine that builds on Dialogflow to identify the user intent using NLP, and does not generate code for existing chatbot development tools. Moreover, even though Xatkit is model-based, it does not address the recommendation of suitable chatbot platforms, nor reduces the risk of vendor lock-in by supporting chatbot migration.

In [3], Baudat et al. facilitate the definition of Watson chatbots by means of an OCaml library which produces the necessary JSON files, and the use of ReactiveML to orchestrate the dialog. While this approach is generative, it is limited to Watson and does not support reverse engineering.

There are some recent model-based proposals to automate the construction of chatbots for a specific task. For example, the framework in [1] permits creating chatbots for video game development; in [20], we generate Dialogflow chatbots to allow instantiating meta-models using a NL syntax; and in [19], we generate model query chatbots. Other works do not rely on models for automating chatbot creation, such as [13], where the authors enable a black-box reuse of components for creating chatbots for FAQ exploration. All these approaches are not general-purpose, but they produce chatbots for a specific task (creating video games, creating models, querying models, or exploring FAQs).

Conversely, in [2], the authors envision a reverse engineering process called *botification* to produce a conversational interface for existing web sites. The process parses a web page to produce a domain model, which serves to configure the allowed NL interactions. Botified webs improve the user experience for visually impaired users, and the development cost is low. We believe that our architecture could serve as a reference to implement this scenario.

[5] https://github.com/Viber/apiai-nutrition-sample

[6] https://www.botium.ai/

Another related line of research concerns crowd-powered conversational assistants [11,12]. While they are not auto-generated, as we do in this paper, they can auto-evolve by learning appropriate responses from previous ones.

Finally, some development tools are specific for voice-user interfaces. For example, tortu[7] supports the visual creation of conversation flows, but it does not allow code generation or bot migration. In a similar vein, VoiceFlow[8] offers a graphical DSL to create voice-based conversation flows that can be deployed on Google home or Alexa, but does not provide recommendation or migration facilities, and the deployment platforms are fixed.

Overall, our approach is novel as it provides a complete MDE solution comprising a unifying DSL for chatbot design, a recommender of up-to-date chatbot development tools according to given design and technical chatbot requirements, and supporting forward and backward engineering, including migration.

9 Conclusion and future work

Nowadays, we can find many tools for building chatbots. While these tools accelerate chatbot development, the chatbot design can become obscured under technical tool details. Moreover, selecting the most appropriate tool, or chatbot migration, require a high investment of time. To alleviate these problems, we have proposed an MDE approach to chatbot development that includes a textual DSL, a platform recommender, code generators and parsers. Our approach supports both forward and reverse chatbot engineering, and has been evaluated by migrating four Dialogflow chatbots developed by third parties to Rasa.

In the future, we plan to extend our framework with more chatbot creation tools, facilities for model-based testing, quick-fixes for violations of chatbot best-practices, and mechanisms to make Conga extensible with platform-specific concepts, like buttons. We are currently migrating our editor of Conga models to a web environment, and later we plan to perform a user study with developers to assess the advantages of our approach. Finally, we plan to create higher-level DSLs to define domain-specific chatbots (e.g., for education or commerce) which can be transformed into our framework for validation and code generation.

Acknowledgments. Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

References

1. Baena-Pérez, R., Ruiz-Rube, I., Dodero, J.M., Bolivar, M.A.: A framework to create conversational agents for the development of video games by end-users. In: OLA. CCIS, vol. 1173, pp. 216–226. Springer (2020)
2. Báez, M., Daniel, F., Casati, F.: Conversational web interaction: Proposal of a dialog-based natural language interaction paradigm for the web. In: CONVERSATIONS. LNCS, vol. 11970, pp. 94–110. Springer (2019)

[7] https://tortu.io/ [8] https://www.voiceflow.com/

3. Baudart, G., Hirzel, M., Mandel, L., Shinnar, A., Siméon, J.: Reactive chatbot programming. In: REBLS@SPLASH. p. 2130. ACM (2018)
4. Botkit: `https://botkit.ai/`, last access in 2020
5. Chatfuel: `https://chatfuel.com/`, last access in 2020
6. Chatterbot: `https://chatterbot.readthedocs.io/`, last access in 2020
7. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Multi-platform chatbot modeling and deployment with the Jarvis framework. In: Proc. CAiSE. LNCS, vol. 11483, pp. 177–193. Springer (2019)
8. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: A multimodal low-code chatbot development framework. IEEE Access **8**, 15332–15346 (2020)
9. Dialogflow: `https://dialogflow.com/`, last access in 2020
10. FlowXO: `https://flowxo.com/`, last access in 2020
11. Huang, T.K., Chang, J.C., Swaminathan, S., Bigham, J.P.: Evorus: A crowd-powered conversational assistant that automates itself over time. In: UIST. pp. 155–157. ACM (2017)
12. Jonell, P., Fallgren, P., Dogan, F.I., Lopes, J., Wennberg, U., Skantze, G.: Crowdsourcing a self-evolving dialog graph. In: CUI. pp. 14:1–14:8. ACM (2019)
13. de Lacerda, A.R.T., Aguiar, C.S.R.: FLOSS FAQ chatbot project reuse: How to allow nonexperts to develop a chatbot. In: OpenSym. ACM (2019)
14. Landbot.io: `https://landbot.io/`, last access in 2020
15. Lex: `https://aws.amazon.com/en/lex/`, last access in 2020
16. LUIS: `https://www.luis.ai/`, last access in 2020
17. Microsoft Bot Framework: `https://dev.botframework.com/`, last access in 2020
18. Pandorabots: `https://home.pandorabots.com/`, last access in 2020
19. Pérez-Soler, S., Daniel, G., Cabot, J., Guerra, E., de Lara, J.: Towards automating the synthesis of chatbots for conversational model query. In: EMMSAD. LNCS (2020)
20. Pérez-Soler, S., González-Jiménez, M., Guerra, E., de Lara, J.: Towards conversational syntax for domain-specific languages using chatbots. Journal of Object Technology **18**(2) (2019)
21. Rasa: `https://rasa.com/`, last access in 2020
22. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006)
23. Shevat, A.: Designing bots: Creating conversational experiences. O'Reilly (2017)
24. SmartLoop: `https://smartloop.ai/`, last access in 2020
25. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework, 2nd edition. Pearson Education (2008)
26. Tegos, S., Demetriadis, S.N.: Conversational agents improve peer learning through building on prior knowledge. Educ. Technology & Society **20**(1), 99–111 (2017)
27. Väänänen, K., Hiltunen, A., Varsaluoma, J., Pietilä, I.: CivicBots - chatbots for supporting youth in societal participation. In: CONVERSATIONS. LNCS, vol. 11970, pp. 143–157. Springer (2019)
28. Watson: `https://www.ibm.com/cloud/watson-assistant/`, last access in 2020
29. Winkler, R., Hobert, S., Salovaara, A., Söllner, M., Leimeister, J.M.: Sara, the lecturer: Improving learning in online education with a scaffolding-based conversational agent. In: CHI. pp. 1–14. ACM (2020)
30. von Wolff, R.M., Nörtemann, J., Hobert, S., Schumann, M.: Chatbots for the information acquisition at universities - A student's view on the application area. In: CONVERSATIONS. LNCS, vol. 11970, pp. 231–244. Springer (2019)
31. Xenioo: `https://www.xenioo.com/en/`, last access in 2020

B.6 Creating and Migrating Chatbots with CONGA

Title **Creating and migrating chatbots with CONGA**

Publication IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)

Core 2021 Core A*

Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid

Doi 10.1109/ICSE-Companion52605.2021.00030

Date May 2021

Abstract Chatbots are agents that enable the interaction of users and software by means of written or spoken natural language conversation. Their use is growing, and many companies are starting to offer their services via chatbots, e.g., for booking, shopping or customer support. For this reason, many chatbot development tools have emerged, which makes choosing the most appropriate tool difficult. Moreover, there is hardly any support for migrating chatbots between tools.

To alleviate these issues, we propose a model-driven engineering solution that includes: (i) a domain-specific language to model chatbots independently of the development tool; (ii) a recommender that suggests the most suitable development tool for the given chatbot requirements and model; (iii) code generators that synthesize the chatbot code for the selected tool; and (iv) parsers to extract chatbot models out of existing chatbot implementations. Our solution is supported by a web IDE called CONGA that can be used for both chatbot creation and migration.

A demo video is available at https://youtu.be/3sw1FDdZ7XY.

Creating and Migrating Chatbots with CONGA

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

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

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

***Abstract*—Chatbots are agents that enable the interaction of users and software by means of written or spoken natural language conversation. Their use is growing, and many companies are starting to offer their services via chatbots, e.g., for booking, shopping or customer support. For this reason, many chatbot development tools have emerged, which makes choosing the most appropriate tool difficult. Moreover, there is hardly any support for migrating chatbots between tools.**

To alleviate these issues, we propose a model-driven engineering solution that includes: (i) a domain-specific language to model chatbots independently of the development tool; (ii) a recommender that suggests the most suitable development tool for the given chatbot requirements and model; (iii) code generators that synthesize the chatbot code for the selected tool; and (iv) parsers to extract chatbot models out of existing chatbot implementations. Our solution is supported by a web IDE called CONGA that can be used for both chatbot creation and migration. A demo video is available at https://youtu.be/3sw1FDdZ7XY.

***Index Terms*—Chatbots, Model-Driven Engineering, Domain-Specific Languages, Migration.**

I. INTRODUCTION

Chatbots are conversational agents that support interaction via natural language (NL) [1]. The improvements in NL processing have triggered their proliferation to access all kind of services, like flight booking, food delivery or customer support. By 2022, Gartner predicts that 70% of all customer interactions will involve machine learning, chatbots and mobile messaging[1]. Many companies are offering their services via chatbots to make them more accessible and user-friendlier, since chatbots are used via NL and can be deployed in social networks (called *channels*) like Telegram or Slack with no need to install dedicated apps [2].

Many chatbot development tools have emerged in recent years. Prominent software companies like Google, IBM, Microsoft or Amazon have launched products for chatbot development (Dialogflow[2], Watson[3], Microsoft Bot Framework[4], Amazon Lex[5]), but a plethora of other options exist, like Rasa[6], Xenioo[7] or Landbot.io[8], to name a few. This variety of tools poses several challenges to chatbot developers:

- *Challenge 1: How to identify the most appropriate development tool based on the chatbot requirements?* [3]. For example, only some tools offer off-the-shelf speech recognition, and tools wildly vary on the supported deployment channels. Choosing an inadequate tool may lead to increased effort [4], lower chatbot quality or project failure.
- *Challenge 2: How to design chatbots independently of the particular tool to enable early reasoning and analysis, prior to the implementation?* Chatbot development tools are very diverse, ranging from low-level programming frameworks (like Rasa) to lowcode development platforms based on forms (like Dialogflow). Grasping the design behind a chatbot implementation may be challenging due to accidental, technical details of the tools themselves.
- *Challenge 3: How to keep up with the rapidly evolving ecosystem of chatbot tools?* With a few exceptions [5], most chatbot tools are closed, proprietary software with no support for migration between tools, e.g., to benefit from the pricing plans of a competitor. This leads to vendor lock-in.

To address these challenges, we propose a web IDE called CONGA that offers a neutral domain-specific language (DSL) for chatbot modelling [6]. Chatbot models can be statically analysed to detect errors and quality issues, and be compiled into tools such as Rasa or Dialogflow. CONGA includes a recommender of suitable development tools for a given chatbot design. The recommender relies on the criteria identified in [3], and takes into account the chatbot model and the answers to a questionnaire of chatbot technical aspects (e.g., *is hosted deployment required?*) and managerial requirements (e.g., *pricing model*). Chatbot migration is facilitated by parsers from development tools into CONGA models, which in turn can be compiled into other platforms. The envisioned users of CONGA are developers and designers with conceptual knowledge on chatbots but not necessarily on their technologies.

This paper showcases the CONGA web IDE, which comprises a textual editor for chatbot modelling, graphical views of the designed conversation flow, a chatbot tool recommender, and generators/parsers to/from some prominent chatbot tools.

II. APPROACH

Next, we overview our approach (Section II-A) and describe its two main components: the DSL for chatbot modelling (Section II-B) and the recommender system (Section II-C).

A. *Overview of the usage methodology of* CONGA

We address the 3 challenges identified in the introduction by means of an automated process supporting both forward (i.e., creating new chatbots) and backward engineering (i.e., migrating existing chatbots). Fig. 1 depicts this process.

[1] https://www.gartner.com/smarterwithgartner/top-cx-trends-for-cios-to-watch/
[2] https://dialogflow.com/ [3] https://www.ibm.com/cloud/watson-assistant/
[4] https://dev.botframework.com/ [5] https://aws.amazon.com/en/lex/
[6] https://rasa.com/ [7] https://www.xenioo.com/en/ [8] https://landbot.io/

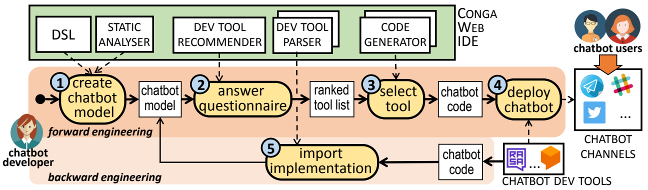


Fig. 1: Forward/backward chatbot engineering with CONGA.

Forward engineering. First, the developer describes the chatbot with a dedicated DSL (label 1 in the figure), explained in Section II-B. The result is a chatbot model that is independent of any development tool and can be statically analysed to detect flaws. Next, to get recommendations on suitable chatbot development tools, the developer answers a questionnaire on additional bot requirements beyond its functional behaviour (label 2). The recommender – detailed in Section II-C – analyses the developer's answers and the chatbot model to elaborate a ranked list of tools. This recommendation step is optional. Then, the developer selects a particular tool (label 3), and the system generates a fully functional chatbot implementation for it. Finally, the developer can deploy the chatbot on a channel (e.g., Telegram) using the selected tool (label 4).

Backward engineering. To support migration, the developer can import an existing chatbot implemented in a specific development tool, and CONGA parses the code to produce the corresponding chatbot model (label 5). The developer can then use this model for forward engineering.

B. The neutral DSL for chatbot modelling

CONGA provides a textual DSL for chatbot modelling, designed based on an analysis of 15 prominent chatbot development tools [6]. Listing 1 illustrates its usage to model a chatbot to help booking flights. First, line 1 declares the *languages* the chatbot should converse in, in this case just English (en), but multi-language chatbots are also possible.

Chatbots are designed around *intents*. These are the actions that users can perform with the bot, like booking or changing a flight. In CONGA, intents can be defined either by regular expressions, or by a set of training phrases showcasing typical ways in which users may express the intention (lines 5–10 in Listing 1). Training phrases may contain *parameters*, which are relevant data that the chatbot needs, such as the source, destination and date of a flight ("from", "to" and "when" in lines 6–7). Each parameter is formally declared by providing its type, whether it is optional or required, and in the latter case, a phrase that the chatbot should ask to the user to request a value for the parameter if it is missing (lines 11–14).

Parameters are typed by *entities* (lines 16–21), which can be pre-defined (like "date") or user-defined (like "City"). User-defined entities specify a set of entries and their synonyms.

Upon recognizing an intent, the chatbot can perform different *actions* such as replying to the user or accessing an external database. This is configured in an "actions" section (lines 23–35). The listing declares a text response (lines 24–27), an image response (lines 28–29) and a POST HTTP request (lines 30–35). The text and the HTTP request use parameters gathered in the intent (Book_flight.from and Book_flight.to).

```
 1 chatbot FlightBooking language: en
 2
 3 intents:
 4   Book_flight:
 5     inputs {
 6       "I need to fly from" ("Madrid")[from] "to" ("Paris")[to]
 7         "on" ("Monday at 9 AM")[when],
 8       "I want to book a flight",
 9       "I need a flight to" ("Rome")[to]
10     }
11     parameters:
12       from: entity City, required, prompts ["What's the flight origin?"];
13       to: entity City, required, prompts ["What is the destination?"];
14       when: entity date, required, prompts ["When do you want to fly?"];
15
16 entities:
17   Simple entity "City":
18     inputs in en {
19       Madrid synonyms MAD, madrid
20       Rome synonyms ROM
21   }
22
23 actions:
24   text response fly_response:
25     inputs in en {
26       "Your flight from" [Book_flight.from] "to" [Book_flight.to] "is booked"
27   }
28   image response send_image:
29     URL: "https://image.shutterstock.com/image−vector.jpg"
30   Request post airline_service:
31     URL: "myURL.com";
32     basicAuth: "user":"pass";
33     headers: "header1":"value1";
34     data: "from": [Book_flight.from], "to": [Book_flight.to];
35     dataType: JSON;
36
37 flows:
38   − user Book_flight => chatbot airline_service, fly_response;
```

Listing 1: A chatbot for booking flights with CONGA (excerpt).

A last "flows" section permits defining conversation *flows* (lines 37–38). These are sequences of user intents (Book_flight) followed by chatbot actions (airline_service and fly_response). Flows can have any length, and there may be several possible user continuations after a chatbot action.

C. The recommender of chatbot development tools

CONGA models are not executable, but they can be compiled into code for a particular development tool. In previous work [3], we identified technical and managerial requirements influencing the tool selection process. To help in selecting an appropriate tool, CONGA integrates a recommender.

The recommender infers some tool requirements from the chatbot model, like the need to support multiple languages, user-defined entities or phrase parameters, among others. Additionally, the developer is presented a questionnaire concerning other non-functional requirements that may influence the tool selection but cannot be derived from the chatbot model. Some examples include the channels where the chatbot is to be deployed, support for chatbot analytics, speech recognition, or being open-source. Overall, the questionnaire has 10 questions [6], each with a customizable relevance that reflects its importance on the recommendation. This way, developers can specify that the answer to a given question is *irrelevant* (disregarded in the recommendation but stored for documentation), *relevant*, *double relevant* or *critical* (tools that do not fulfil the requirement will not be recommended).

The recommender uses the chatbot model and the answers the questionnaire to assign a score to each development to where higher scores indicate wider requirements coverage.

III. Tool Support

This section describes the architecture (Section III-A) a front-end (Section III-B) of CONGA. The tool is open sourc and is available at https://saraperezsoler.github.io/CONGA/.

A. Architecture

CONGA is available to chatbot developers as a web app cation. Fig. 2 shows its architecture. The front-end includ user and project managers, a DSL editor, a graphical render of conversation flow models, importers/exporters for so chatbot tools, a questionnaire for the tool recommender, and a

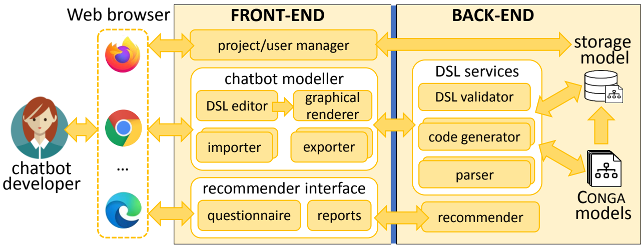


Fig. 2: CONGA's architecture.

The storage model of CONGA conforms to the meta-model of Fig. 3. It includes a *Recommender* class that defines the list of *Tool*s that may be recommended (e.g., Dialogflow, Lex), and the *Requirements* considered to calculate the recommendation. There are two types of requirements: *Question*, which corresponds to a query in the recommender questionnaire (e.g., the deployment channels), and *Analysis*, which refers to technical requirements extracted from the chatbot model (e.g., the bot spoken languages). Both requirement types have a *name*, a *text* question, a closed list of response *options*, and can optionally be *multi*-option. Each tool considered for recommendation must define which of the specified requirements options are *available*, *unavailable*, *unknown* or might be *possible* in the tool. Currently, our recommender considers 10 questions, 7 model analyses, and 14 up-to-date target implementation tools; however, our model-based design makes the recommender fully extensible with new questions, analyses and tools.

Chatbot definitions are stored in *Project*s. Each project stores the developer's *Answer*s to the *Question*s in the recommender *Questionnaire*. The answers comprise both the selected options and the relevance level assigned to the question.

B. Front-end

Fig. 4 shows the main interface of CONGA. The header (label 1) includes the logged user name, and a sign out button. The toolbar (label 2) contains buttons to save the file with the chatbot model, create a new project, format the displayed file, select a development tool to generate code for (currently Dialogflow or Rasa), fill in the recommender questionnaire, and display the recommendation results. New projects can be created empty, or be populated with a model parsed from an existing chatbot implementation (currently from Dialogflow).

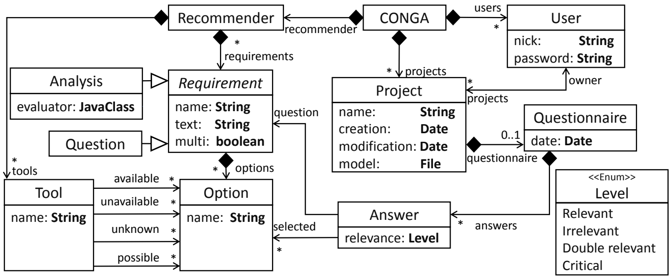
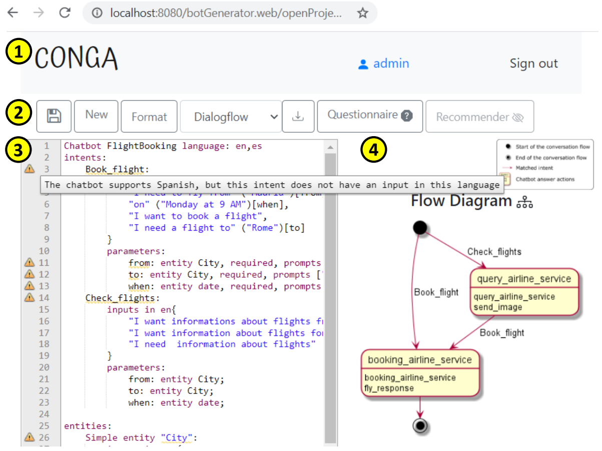


Fig. 3: CONGA storage meta-model.



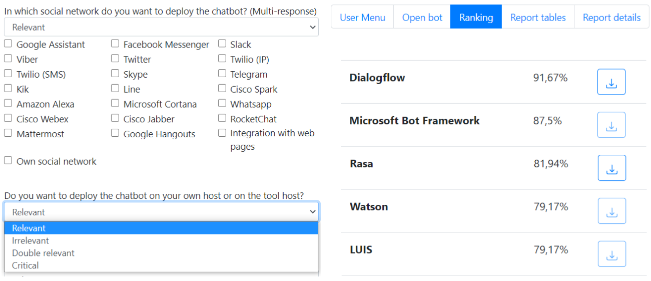
```
Chatbot FlightBooking language: en,es
intents:
    Book_flight:
The chatbot supports Spanish, but this intent does not have an input in this language
            "I need to fly from" ("Madrid")[from
            "on" ("Monday at 9 AM")[when],
            "I want to book a flight",
            "I need a flight to" ("Rome")[to]
        }
        parameters:
            from: entity City, required, prompts
            to: entity City, required, prompts ['
            when: entity date, required, prompts
    Check_flights:
        inputs in en{
            "I want informations about flights fr
            "I want information about flights fo
            "I need  information about flights"
        }
        parameters:
            from: entity City;
            to: entity City;
            when: entity date;

entities:
    Simple entity "City":
```

Fig. 4: CONGA's main interface.

The DSL editor (label 3) features syntax highlighting, content assistance and error reporting. In addition to syntax errors, a validator checks problems like intents with overlapping training phrases, similar conversation flows, or flows where an action uses parameters that no previous intent in the flow defines. In the figure, the editor reports some warnings; the first one warns that the chatbot is multi-language (English, Spanish) but the training phrases only consider one language (English). Technically, the editor is implemented in Xtext, using its web deployment options for the codemirror JavaScript library.

The flow diagram to the right (label 4) depicts graphically the conversation flow defined by the chatbot model. The diagram represents the user interactions as transitions, and the chatbot interactions as states with the actions that the chatbot performs inside. This view is built using PlantUML, and becomes updated whenever the chatbot file is saved.

Fig. 5a shows an excerpt of the questionnaire that developers can answer to obtain tool recommendations. The questionnaire is created on-the-fly according to the modelled requirements (cf. Fig. 3), which allows updating easily the requirements.


In which social network do you want to deploy the chatbot? (Multi-response)

Relevant

□ Google Assistant □ Facebook Messenger □ Slack
□ Viber □ Twitter □ Twilio (IP)
□ Twilio (SMS) □ Skype □ Telegram
□ Kik □ Line □ Cisco Spark
□ Amazon Alexa □ Microsoft Cortana □ Whatsapp
□ Cisco Webex □ Cisco Jabber □ RocketChat
□ Mattermost □ Google Hangouts □ Integration with web pages

□ Own social network

Do you want to deploy the chatbot on your own host or on the tool host?

Relevant

Relevant
Irrelevant
Double relevant
Critical

(a) Requirements questionnaire.

Dialogflow	91,67%
Microsoft Bot Framework	87,5%
Rasa	81,94%
Watson	79,17%
LUIS	79,17%

(b) Resulting tool ranking.

Fig. 5: Conga recommender support.

TABLE I: Assessment metrics.

	Dialogflow		Conga	Rasa		
	Back-end	#Files	LOC	Python LOC	Markd. LOC	YAML LOC
Bike Shop[a]	yes	13	80	185	61	187
Mystery Animal[b]	yes	199	7042	9494	13722	879
Smalltalk[c]	no	58	1515	284	1421	281
IoT: Turn lights[d]	yes	6	53	125	23	168

[a] https://bit.ly/38THi8h [b] https://bit.ly/2IQZ8yf [c] https://bit.ly/36KMKrq [d] https://bit.ly/3lEchc2

Each question has a list of options and a selector of relevance.

Fig. 5b displays the ranking of tools ordered by decreasing score. By clicking on the button to the right of a tool, the corresponding code generator is invoked and the developer can download the resulting artefacts.

IV. Evaluation

We have evaluated the migration capabilities of Conga by importing four third-party, non-trivial Dialogflow agents from GitHub into Conga, and then generating corresponding chatbot implementations for the Rasa development framework. This evaluation extends the one presented in [6] by considering more challenging bots with back-ends or complex logic, leading to models with thousands LOC in Conga.

Table I shows size metrics of the chatbots in Dialogflow, Conga and Rasa. Bike Shop schedules appointments for a shop; Mystery Animal is a guessing game via Q&A; Smalltalk is a chitchatting agent; and IoT turns the lights on/off via NL.

Conga was able to automatically migrate all chatbot logic from Dialogflow to Rasa, obtaining functional bots. The largest bot parsed into >7000 Conga LOC, and produced a Rasa implementation with >9000 Python LOC and >14000 LOC in configuration files. This proves the usefulness of our tool.

However, two aspects required manual intervention. First, Smalltalk uses emojis, currently not supported by Conga. Second, three Dialogflow agents had back-ends developed using Google libraries tightly integrated with Dialogflow. Those cases required configuring the Google services manually and, in one case, implementing a middleware. Generally, the chatbot/back-end connection cannot be migrated fully automatically since it may rely on native technologies of the chatbot platform (e.g., Google's cloud, AWS services).

V. Related Work

The raising popularity of chatbots has led to new tools for their construction (see [3] for a survey). Most are frameworks or platforms, and only a few provide DSLs. The closest work to ours is the model-based solution Xatkit [5]. This provides a textual DSL for chatbot development, but contrary to Conga, the defined chatbots are executable by providing an execution engine. Moreover, even though Xatkit can help addressing challenge 2 in the introduction (chatbot design), it neither provides a neutral language nor supports tool recommendation or migration (challenges 1 and 3).

Baudart et al. [7] propose an embedded DSL to define Watson chatbots based on an OCaml library, and orchestrate the dialog using ReactiveML. However, an embedded DSL makes the chatbot design less explicit, and while the approach is generative, it is limited to Watson and does not support migration. Protochat [8] provides a graphical DSL for conversation design, and supports a crowd-testing approach whereby crowd workers can provide feedback on the conversation. Finally, some approaches automate chatbot construction from existing artefacts, such as web sites [9].

Overall, there are previous proposals of DSLs for chatbot design, but Conga is unique for being designed from an analysis of 15 chatbot development tools, and because it addresses tool migration and recommendation.

VI. Conclusions and Future Work

This paper has presented Conga, a model-driven solution for forward and backward chatbot engineering, featuring a recommender system that assists in selecting the most suitable chatbot development tools. Our approach is extensible by implementing interfaces to create new code generators and parsers, but we are currently working in extension points to facilitate this extensibility. Finally, we plan to conduct a user study to assess the usability of Conga.

Acknowledgment

Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

References

[1] A. Shevat, *Designing bots: Creating conversational experiences*. O'Reilly, 2017.

[2] P. B. Brandtzæg and A. Følstad, "Why people use chatbots," in *INSCI*, ser. LNCS, vol. 10673. Springer, 2017, pp. 377–392.

[3] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, and J. de Lara, "Choosing a chatbot development tool," *IEEE Software*, vol. in press, 2020.

[4] A. Abdellatif, D. Costa, K. Badran, R. Abdalkareem, and E. Shihab, "Challenges in chatbot development: A study of stack overflow posts," in *MSR*. ACM, 2020, pp. 174–185.

[5] G. Daniel, J. Cabot, L. Deruelle, and M. Derras, "Xatkit: A multimodal low-code chatbot development framework," *IEEE Access*, vol. 8, pp. 15 332–15 346, 2020.

[6] S. Pérez-Soler, E. Guerra, and J. de Lara, "Model-driven chatbot development," in *ER*, ser. LNCS, vol. 12400. Springer, 2020, pp. 207–222.

[7] G. Baudart, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon, "Reactive chatbot programming," in *REBLS@SPLASH*. ACM, 2018, pp. 21–30.

[8] Y. Choi, T. K. Monserrat, J. Park, H. Shin, N. Lee, and J. Kim, "Protochat: Supporting the conversation design process with crowd feedback," in *CSCW*. ACM, 2020, pp. 19–23.

[9] P. Chittò, M. Báez, F. Daniel, and B. Benatallah, "Automatic generation of chatbots for conversational web browsing," in *ER*, ser. LNCS, vol. 12400. Springer, 2020, pp. 239–249.

B.7 Automating the Synthesis of Recommender Systems for Modelling Languages

Title	Automating the Synthesis of Recommender Systems for Modelling Languages	
Publication	ACM SIGPLAN International Conference on Software Language Engineering (SLE)	
Core	2021 Core B	
Authors	Lissette Almonte	Universidad Autónoma de Madrid
	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Iván Cantador	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1145/3486608.3486905	
Date	October 2021	
Abstract	We are witnessing an increasing interest in building recommender systems (RSs) for all sorts of Software Engineering activities. Modelling is no exception to this trend, as modelling environments are being enriched with RSs that help building models by providing recommendations based on previous solutions to similar problems in the same domain. However, building a RS from scratch requires considerable effort and specialized knowledge. To alleviate this problem, we propose an automated approach to the generation of RSs for modelling languages. Our approach is model-based, and so we provide a domain-specific language called Droid to configure every aspect of the RS (like the type and features of the recommended items, the recommendation method, and the evaluation metrics). The RS so configured can be deployed as a service, and we provide an out-of-the-box integration of this service with the EMF tree editor. To assess the usefulness of our proposal, we present a case study on the integration of a generated RS with an existing modelling chatbot, and report an offline experiment that yields promising results.	

Automating the Synthesis of Recommender Systems for Modelling Languages

Lissette Almonte
Universidad Autónoma de Madrid
Madrid, Spain

Sara Pérez-Soler
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain

Iván Cantador
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain

Abstract

We are witnessing an increasing interest in building recommender systems (RSs) for all sorts of Software Engineering activities. Modelling is no exception to this trend, as modelling environments are being enriched with RSs that help building models by providing recommendations based on previous solutions to similar problems in the same domain. However, building a RS from scratch requires considerable effort and specialized knowledge.

To alleviate this problem, we propose an automated approach to the generation of RSs for modelling languages. Our approach is model-based, and we provide a domain-specific language called Droid to configure every aspect of the RS (like the type and features of the recommended items, the recommendation method, and the evaluation metrics). The RS so configured can be deployed as a service, and we offer out-of-the-box integration of this service with the EMF tree editor. To assess the usefulness of our proposal, we present a case study on the integration of a generated RS with a modelling chatbot, and report on an offline experiment measuring the precision and completeness of the recommendations.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; **Designing software**; • **Information systems** → **Information retrieval**.

Keywords: Modelling Languages, Model-Driven Engineering, Domain-Specific Languages, Recommender Systems

ACM Reference Format:
Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Automating the Synthesis of Recommender Systems for Modelling Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486608.3486905

1 Introduction

Modelling plays a fundamental role in Software Engineering, especially in model-driven engineering (MDE) [9]. In this paradigm, models are actively used in the different development phases to specify, analyse, design, simulate and test the system to be built, among other activities.

Modelling is performed using modelling languages which can be general-purpose ones, like the Unified Modelling Language (UML) [56], or domain-specific languages (DSLs) tailored to a target domain [27, 58]. Today, sophisticated development environments and powerful language workbenches are the norm. However, modelling remains mostly a manual activity, which often does not profit from knowledge found in existing models, or the experience of engineers working on similar domains.

Recommender systems (RSs) [1] are information filtering systems that help users in choosing among a potentially large set of items (e.g., movies, songs or books). They aim at predicting the preferences of users to offer a prioritised list of potentially interesting items. They are widely used in all sorts of commercial and leisure applications, and their use in software engineering activities is increasing as well [46]. This way, we can find RSs that help in choosing appropriate third-party programming libraries [38, 55], recommend API method invocations [37], suggest code refactorings [13, 22], and assist on the evaluation of change impact analysis [8], to name a few. Recently, we are also witnessing an incipient interest to apply RSs to modelling (see, e.g., [2, 5, 10, 12, 15, 20, 23, 31, 34, 35]); however, their use in MDE is not the norm yet. One possible reason is that building RSs requires deep expertise in recommendation techniques, and involves an important development effort [4, 36].

With the aim to facilitate the adoption of RSs in MDE, we propose a model-driven solution to automate the synthesis of RSs for modelling languages. Based on the vision put forward in [3], our solution consists of a DSL called Droid supporting

SLE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486905

the configuration of the kind of model elements to be recommended, and an engine that automates the evaluation of different recommendation methods against configurable metrics. The selected recommendation method is deployed as a service, which heterogeneous modelling clients can integrate. Currently, we provide an automated, out-of-the-box integration with the tree editor of the Eclipse Modeling Framework (EMF) [53], but additionally, the generated recommenders can be integrated with other modelling technologies. To assess this fact, we describe a case study of the integration of a recommender with a third-party modelling chatbot called Socio [42]. Finally, to assess the usefulness of our proposal, we report on an offline evaluation of a RS created with our approach over UML models. The experiment results are in line with RSs specifically created for class diagrams [10], but our approach does not require any programming.

Paper organization. Section 2 provides background on RSs. Section 3 overviews our approach, and Section 4 presents the Droid DSL. Section 5 details the technical architecture and tool support. Section 6 presents a case study incorporating a RS to a modelling chatbot, and an offline evaluation. Section 7 compares with related research, and Section 8 ends with the conclusions and future work.

2 Recommender Systems

Recommender systems have become a key component of a large and varied number of software applications. Nowadays, everyone is exposed to recommendation services on music (e.g., Spotify, Pandora) and video (e.g., Netflix, YouTube) streaming platforms, e-commerce sites (e.g., Amazon, eBay), and social networks (e.g., Facebook, Twitter), among others.

Common to all these applications, RSs analyse the activity of a typically very large group of users to provide them with personalised suggestions of options (items), based on the evidence observed about their interests and preferences. In this context, they provide advantages both for the users – whose experience is improved by receiving ideas about content to consume or products to buy – and the service providers – by promoting increased sales and customer loyalty, as customers are able to discover content or products that they would not have known otherwise.

In Software Engineering in general [21], and MDE in particular [4], RSs have also found a wide array of applications. Integrated in software design and development tools, recommendation services can assist on the creation [43], completion [2, 20], repair [6, 41, 50], search [35], and reuse [54] of artefacts, e.g., models, meta-models and transformations.

In these cases, the target *user* for whom recommendations are generated and her *preferences* (or *profile*) may have special meanings. For instance, it may refer to a *class* in a UML diagram for a recommender that is devoted to suggesting potential attributes and methods of interest for incomplete classes; the recommended *items* or artefacts are thus class attributes and methods, and the preferences and features that describe users and items can be the names and types of class attributes, methods and method arguments.

In general, the recommendations are generated based on content-based similarities between users and items [33], user-item preference (rating) patterns identified in the user community via collaborative filtering techniques [39], or both sources of information via hybridisation methods [11]. *Content-based* (CB) systems suggest items "similar" to those the target user preferred in the past, whereas *collaborative filtering* (CF) systems suggest items preferred by like-minded people. Moreover, CF approaches can consider either the similarities of the k most similar users – known as nearest neighbours – to the target user (UBCF), or the similarities of the k most similar items (IBCF) to the target user's items. Finally, typical hybrid approaches follow user- or item-based CF strategies exploiting content-based similarities (CBUB or CBIB) instead of rating-based similarities as CF does.

The quality of the recommendations can also be evaluated through different approaches [24]. User studies allow evaluating a recommender online, e.g., via A/B tests that capture the impact that recommendations have in real time. Offline experiments, by contrast, are conducted on datasets made of past user-item interactions, and are split into training and test data to build and evaluate a recommender, respectively. For both types of evaluations – online and offline – several metrics can be computed [7]. Typical metrics that measure the ranking quality of the recommendation lists are *precision*, i.e., the probability that a selected item is relevant; *recall*, i.e., the percentage of relevant items in the recommendation lists; *F1*, i.e., the harmonic mean of precision and recall; *MAP* (Mean Average Precision), i.e., the mean average precision over all the users; and *nDCG* (Normalized Discounted Cumulative Gain), which considers if the most useful items appear in the top positions of the recommendation lists. Other complementary metrics are *USC* (User Space Coverage), which measures the percentage of users that the RS can recommend, and *ISC* (Item Space Coverage), which measures the diversity of the recommendations.

As we will present in the subsequent sections, our model-based approach to automatically generate RSs allows configuring all the above-mentioned aspects: recommendation methods, target user and item profiles, and offline evaluation methodologies and metrics (cf. Listing 1).

3 Overview of the Approach

To facilitate the construction of RSs for arbitrary modelling languages, we propose a model-based solution whose scheme is depicted in Fig. 1. Our solution permits customising a RS for a particular modelling language, assists in deciding which recommendation method works better for the recommendation task at hand, and generates a recommendation service that can be integrated with external modelling tools.

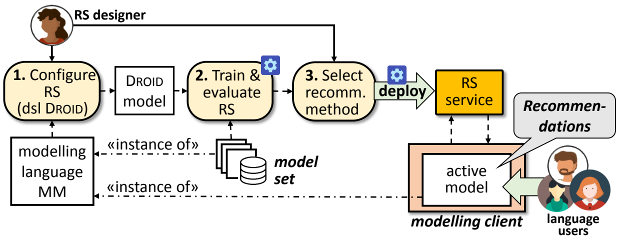


Figure 1. Overview of the approach.

Our approach makes available a DSL called Droid to configure the RS for the targeted modelling language. The approach assumes that the modelling language is defined by a meta-model. This way, in the first step, the designer of the RS uses Droid to select from the modelling language meta-model the elements that are to be recommended (e.g., attributes for a class, tasks for a process model). The DSL also permits specifying the candidate recommendation methods, the dataset used to train the recommenders, and the evaluation metrics used to rank the built recommenders. Section 4 will present Droid in detail.

Next, in the second step, our system automatically evaluates each selected recommendation method against the indicated metrics, using the provided dataset. The result is an interactive report with the value of each metric for the recommendation methods.

In the last step, the RS designer chooses a recommendation method, and the system automatically synthesizes a RS service that can be integrated within different modelling tools. Currently, our system provides full automatic support for deploying the RS within the Eclipse modelling tree editor [53], but other modelling clients are possible as well. Sections 5.4 and 6.2 will provide details on this out-of-the-box client integration, and others.

4 The Droid DSL

Droid is a textual DSL for the configuration of RSs for particular modelling languages. Fig. 2 shows its meta-model, which permits detailing the following aspects of a RS:

(i) The URI of the meta-model of the modelling language, and the repository containing the instance models to be used for training the RS. In Fig. 2, this is captured by the class RecommenderConfiguration and its attributes.

(ii) The class subject to recommendations, called *target* (class DomainClass and reference RecommenderConfiguration.target). The items to be recommended (class DomainProperty, its subclasses and reference DomainClass.items).

(iii) The way the objects of the target class and the items are identified in the models (references DomainClass.pk and DomainClass.features).

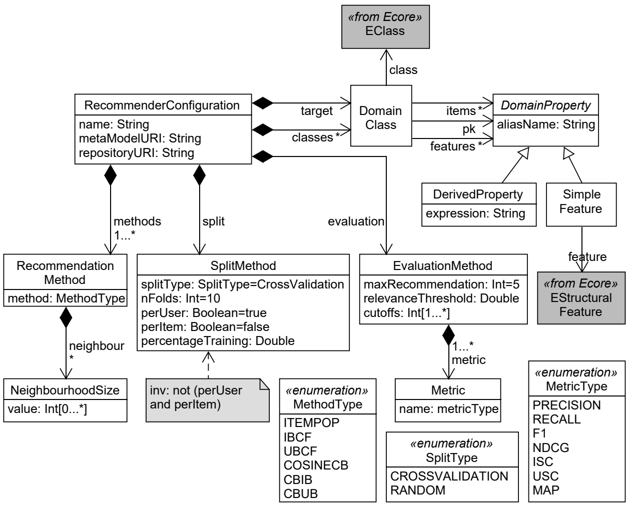


Figure 2. Meta-model of Droid.

(iv) The information about the candidate recommendation methods for the RS, and the metrics used to compare them. This is specified through the classes RecommendationMethod, SplitMethod and EvaluationMethod.

To illustrate Droid, we will use an example consisting of the configuration of a RS for UML class modelling. The RS will recommend attributes, operations and superclasses for a given class. Listing 1 shows its configuration using Droid.

In Listing 1, line 1 specifies the *name* of the recommender; line 2 specifies the *meta-model* that the RS will use (in this case, the meta-model of UML 2.0 class diagrams); and line 3 specifies the *repository* of models to be used to train and evaluate the selected candidate recommendation methods. The latter models need to conform to the specified meta-model. Since in our example, the training models are UML class models in the domain of libraries, the recommender is named "Literature Recommender."

Fig. 3 shows a small, simplified excerpt of the UML meta-model needed for our example: Classes, which contain both attributes (class Property) and Operations, and relate to their ancestors using the superClass derived reference.

The section "Target" in Listing 1 (lines 5–11) declares the *target* class of the recommendation and its relevant *items*. This way, the synthesised RS will provide recommendations for each declared item when invoked on objects of the target class. Line 6 specifies that the class Class is the recommendation target, and lines 7–9 specify three types of items to recommend: attributes, methods and superclasses. Each item has a *name* (displayed to the user when the recommendation is performed), and the attributes or references leading from the target class to the items (in the example, ownedAttribute, ownedOperation and superClass, cf. Fig. 3). The meta-model of Droid also permits specifying derived properties via OCL

```
 1 Recommender: "LiteratureRecommender"
 2 Metamodel: "http://www.eclipse.org/uml2"
 3 Repository: "/LiteratureRecommender/instances"
 4
 5 Target {
 6   class Class {
 7     item "attributes" : ownedAttribute;
 8     item "methods" : ownedOperation;
 9     item "super classes" : superClass;
10   }
11 }
12
13 Identifiers {
14   class Class {
15     pk feature name;
16   }
17   class Property {
18     pk feature name;
19     pk feature type;
20   }
21   class Operation {
22     pk feature name;
23     pk feature type;
24   }
25   class Type {
26     pk feature name;
27   }
28 }
29
30 Recommendations {
31   Methods {
32     collaborativeFiltering: ItemPop, IBCF(10,15,20,25,50,100),
33                                        UBCF(10,15,20,25,50,100);
34     contentBased: CosineCB;
35     hybrid: CBIB(10,15,20,25,50,100), CBUB(10,15,20,25,50,100);
36   }
37   Split {
38     splitType: CrossValidation;
39     nFolds: 10;
40     perUser: true;
41   }
42   Evaluation {
43     metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
44     cutoffs: 1,5,10,15,20;
45     maxRecommendations: 50;
46     relevanceThreshold: 0.5;
47   }
48 }
```

Listing 1. Defining a RS for UML class diagrams with Droid.

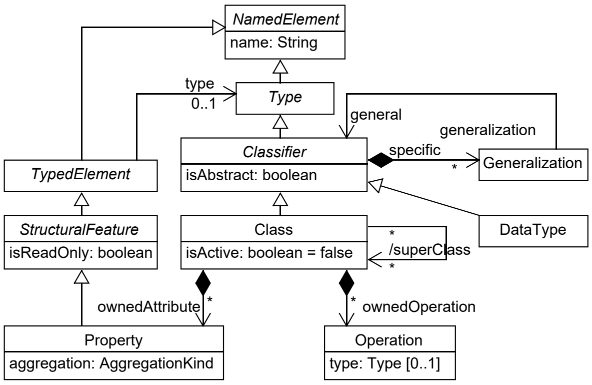


Figure 3. Simplified excerpt of the UML meta-model.

expressions [40] (class DerivedProperty in Fig. 2), though our current implementation does not support it yet.

The section "Identifiers" in lines 13–28 declares the identifiers and features of the involved classes. In our example, Classes are described by their name (line 15), which is their primary key (prefix pk), while Properties and Operations are identified by their name and type. The type of the latter is Type (cf. Fig. 3), and so, lines 25–27 declare its identifier.

The last section is "Recommendations", in lines 30–48. This enables the configuration of recommendation methods for the RS, and the way to evaluate them. This information is optional, since Droid provides default values in case the RS designer does not have the required expertise or is not interested in a fine-grained configuration of the evaluation process (see, e.g., the default values of class SplitMethod in Fig. 2). The section has three subsections: "Methods", "Split" and "Evaluation" (classes RecommendationMethod, SplitMethod and EvaluationMethod in Fig. 2).

First, the subsection "Methods" (lines 31–36) specifies the recommendation methods that the designer wants to experiment with to determine the best one for the case at hand. Lines 32–33 select the collaborative filtering methods *ItemPOP* (item popularity), *IBCF* (item-based collaborative filtering), and *UBCF* (user-based collaborative filtering). The latter two are configured with neighbourhood sizes of 10, 15, 20, 25, 50 and 100. Next, line 34 specifies the content-based method *CosineCB* (pure content-based method), and line 35 selects the hybrid methods *CBIB* (content-based item-based) and *CBUB* (content-based user-based) with neighbourhood sizes of 10, 15, 20, 25, 50 and 100. Overall, these are the six recommendation methods currently supported by Droid (cf. enumeration MethodType in Fig. 2).

The subsection "Split" (lines 37–41) specifies how to divide the dataset for the evaluation of the recommendation methods. The listing defines a 10-fold *cross-validation* split type, following a *perUser* technique. The split type refers to the approach to divide the data into training and test sets. *Cross-validation* divides the data into k subsets, one used for test and the rest for training, and repeats the process assigning in each iteration the role of test set to each one of the k subsets. Droid also supports *random* split, in which case, the percentage of data used for training/test must be given, and the sampling for the test/training sets is done randomly with a uniform distribution [45]. Splits can be built using either a *perUser* or a *perItem* technique. In the former case, the subsets are built per available user, while with *perItem*, they are built by available item. For example, a *perUser random* split type with 80% training percentage implies that 80% of the preferences of each user (i.e., 80% of the attributes, methods and superclasses of each class) will be used as the training set, and the remainder 20% as the test set.

The last part of the listing (subsection "Evaluation" in lines 42–47) describes the evaluation protocol. This includes the desired metrics for the evaluation (Precision, Recall, F1, nDCG, ISC, USC and MAP, cf. Section 2); the number of items in the top of the ranking that will be used to calculate the metrics (cutoffs, line 44); the maximum number of items that the RS will recommend (maxRecommendations, line 45); and a threshold value for the rating of items used to determine whether an item is relevant and a good recommendation, or not (relevanceThreshold, line 46). This threshold defines a binary classification for the probability of a prediction to be true. In the listing, the relevance threshold of 0.5 implies that the rating values below 0.5 are considered false (irrelevant), and those equal to or above 0.5 are true (relevant).

5 Architecture and Tool Support

In this section, we introduce the architecture of Droid (Section 5.1), the tool support for RS design (Section 5.2), the generated recommendation service (Section 5.3), and the automatic integration with the EMF tree editors (Section 5.4).

5.1 Architecture

Fig. 4 shows the architecture of the Droid ecosystem, which comprises three parts. The first one is the *Droid Configurator*, which permits the configuration, evaluation and synthesis of RSs. The configurator provides an Eclipse textual editor for the DSL presented in Section 4, where the RS designer can configure the RS for a particular modelling language (label 1). The specified configuration is the input to the *RS Evaluator* (label 2), which relies on the external libraries *RankSys* [57] and *RiVal* [49] to evaluate the recommendation methods selected by the RS designer using Droid. RankSys is a frame-

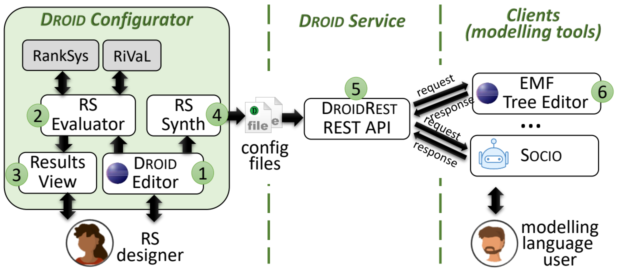


Figure 4. Architecture of Droid.

Based on the obtained results, the RS designer can select the preferred recommendation method, and a *RS Synthesizer* generates a set of configuration files out of the selection and the RS configuration (label 4). The configuration files are used by the second part of our ecosystem, which is the *Droid Service* (label 5). This is a generic recommendation REST API that can be customised for particular modelling languages using the configuration files generated by the *RS Synthesizer*. The service enables clients to request recommendations using a JSON-based model representation. The service processes such requests and sends the recommendations as a response. Section 5.3 will elaborate on this service.

Finally, in the *Client* part, any modelling tool can use the *Droid Service* to obtain recommendations and make them available to the modelling language users (label 6). Currently, our tooling supports the automatic integration of the resulting RSs within the default tree editor that EMF provides for Ecore-based languages. In Section 6.2, we will show another integration within a modelling chatbot.

5.2 Tool support: The Droid configurator

The *Droid Configurator* (https://droid-dsl.github.io/) is an Eclipse plug-in that helps the RS designer in configuring and evaluating RSs for a modelling language.

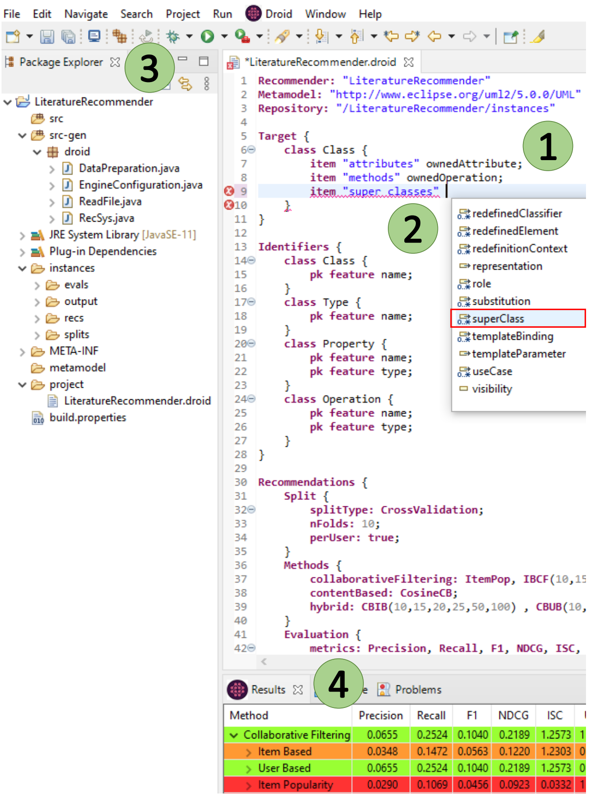
It provides a wizard where the RS designer can create *droid projects* by specifying a name for the project, the meta-model of the language for which the RS is being developed, a folder containing the models to be used for training and evaluating the RS, and the format of these models (XMI, Ecore, or UML). To simplify the RS configuration, the wizard gives the option to automatically generate a default one (i.e., default values for the "Recommendations" section in lines 30–48 of Listing 1), which the designer can modify later if so desired.

Fig. 5 shows the *Droid Configurator* environment. The Droid editor (label 1) permits the configuration of the RS via the DSL introduced in Section 4. The editor has been built using Xtext [59], and features syntax highlighting, autocompletion, and markers for errors and warnings. With label 2, the figure shows an auto-completion pop-up window to choose an existing attribute of the class Class from the UML meta-model, to serve as an item of the target class.

The environment includes a code generator that synthesizes Java code from the Droid specification. This code is in charge of evaluating the RSs. The package explorer in the figure (label 3) shows the generated Java classes in the *src-gen* folder. The RS designer does not need to look into this code, since the *RS Evaluator* component (cf. Fig. 4) automatically generates the code and displays the results in a dedicated Eclipse view (label 4).

The *Results View* (label 4) summarises in a drill-down table the evaluation results for each recommendation method and metric. The table uses different colours to facilitate the comparison of the metric values (specifically, of the values of the *F1* metric). The recommendation methods whose *F1* value is in the top 20% are shown in green; the methods whose *F1* value is under the median are shown in red; and the rest of the methods are shown in orange.

Fig. 6 shows the *Results View* in more detail. The view groups the evaluated methods by category: *Collaborative Filtering*, *Content-Based* and *Hybrid*. Within a group, each method contains a subsection per neighbourhood size, if applicable. The rows corresponding to a group show the results of the method with the best *F1* value within the group.



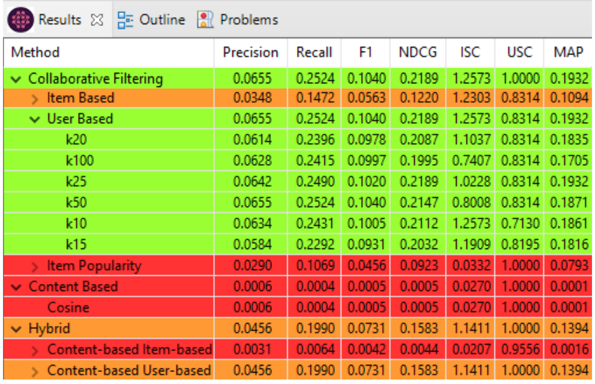
```
LiteratureRecommender
    src
    src-gen
        droid
            DataPreparation.java
            EngineConfiguration.java
            ReadFile.java
            RecSys.java
    JRE System Library [JavaSE-11]
    Plug-in Dependencies
    instances
        evals
        output
        recs
        splits
    META-INF
    metamodel
    project
        LiteratureRecommender.droid
    build.properties
```

```
Recommender: "LiteratureRecommender"
Metamodel: "http://www.eclipse.org/uml2/5.0.0/UML"
Repository: "/LiteratureRecommender/instances"

Target {
    class Class {
        item "attributes" ownedAttribute;
        item "methods" ownedOperation;
        item "super classes"
    }
}

Identifiers {
    class Class {
        pk feature name;
    }
    class Type {
        pk feature name;
    }
    class Property {
        pk feature name;
        pk feature type;
    }
    class Operation {
        pk feature name;
        pk feature type;
    }
}

Recommendations {
    Split {
        splitType: CrossValidation;
        nFolds: 10;
        perUser: true;
    }
    Methods {
        collaborativeFiltering: ItemPop, IBCF(10,15
        contentBased: CosineCB;
        hybrid: CBIB(10,15,20,25,50,100) , CBUB(10,
    }
    Evaluation {
        metrics: Precision, Recall, F1, NDCG, ISC,
```

Method	Precision	Recall	F1	NDCG	ISC	
Collaborative Filtering	0.0655	0.2524	0.1040	0.2189	1.2573	1
Item Based	0.0348	0.1472	0.0563	0.1220	1.2303	0
User Based	0.0655	0.2524	0.1040	0.2189	1.2573	0
Item Popularity	0.0290	0.1069	0.0456	0.0923	0.0332	1

Figure 5. Screenshot of the *Droid Configurator.*

For example, row "Collaborative Filtering" shows the metrics of the collaborative filtering method with best *F1* value.



Method	Precision	Recall	F1	NDCG	ISC	USC	MAP
Collaborative Filtering	0.0655	0.2524	0.1040	0.2189	1.2573	1.0000	0.1932
Item Based	0.0348	0.1472	0.0563	0.1220	1.2303	0.8314	0.1094
User Based	0.0655	0.2524	0.1040	0.2189	1.2573	0.8314	0.1932
k20	0.0614	0.2396	0.0978	0.2087	1.1037	0.8314	0.1835
k100	0.0628	0.2415	0.0997	0.1995	0.7407	0.8314	0.1705
k25	0.0642	0.2490	0.1020	0.2189	1.0228	0.8314	0.1932
k50	0.0655	0.2524	0.1040	0.2147	0.8008	0.8314	0.1871
k10	0.0634	0.2431	0.1005	0.2112	1.2573	0.7130	0.1861
k15	0.0584	0.2292	0.0931	0.2032	1.1909	0.8195	0.1816
Item Popularity	0.0290	0.1069	0.0456	0.0923	0.0332	1.0000	0.0793
Content Based	0.0006	0.0004	0.0005	0.0005	0.0270	1.0000	0.0001
Cosine	0.0006	0.0004	0.0005	0.0005	0.0270	1.0000	0.0001
Hybrid	0.0456	0.1990	0.0731	0.1583	1.1411	1.0000	0.1394
Content-based Item-based	0.0031	0.0064	0.0042	0.0044	0.0207	0.9556	0.0016
Content-based User-based	0.0456	0.1990	0.0731	0.1583	1.1411	1.0000	0.1394

Figure 6. *Results View* of the *Droid Configurator.*

5.3 Tool support: The Droid service

We have built a generic recommender called DroidREST. It is a REST service implemented in Java using Jersey[1] and Tomcat[2]. The service computes the recommendations based on the configuration files generated by the *RS Synthesizer* (cf. Fig. 4). These configuration files store the trained recommender that knows which items to suggest based on the context information. Hence, there is no need to deploy a different service for each RS defined with Droid.

Clients can make POST requests to the service, which receives a recommender name together with a JSON file containing the target object of the recommendation and its context (i.e., the items that the target contains). The response to the request is a list of recommended items for the given target, using the recommendation method selected by the designer. In addition, clients can pass optional parameters for specific settings, like the maximum number of recommended items to retrieve (*newMaxRec*), the threshold for the ranking value (*threshold*), and the type of item (*itemType*). The response time of the service to calculate the recommendations is less than a second.

The REST service implementation comprises three main classes: *Recommender*, which handles the requests from clients; *ContextItem*, which parses the received JSON files to extract the recommendation target and its items from the modelling context; and *RecommenderGenerator*, which generates the recommendations for the given target taking its context and the provided query parameters into account.

5.4 Tool support: Integration with EMF tree editor

EMF automates the synthesis of a default modelling editor starting from the Ecore meta-model of a modelling language. This editor permits creating instances of the meta-model using a tree view. Given the widespread use of these editors, our implementation generates out-of-the-box an integration of the Droid recommendation service into the default EMF tree editor of a modelling language. Next we explain the technical details of this client integration, and show an example.

In EMF, the generation of the default tree editors is automated by means of a model-to-text template language called Java Emitter Template (JET)[3]. JET supports the definition and execution of code generation templates from EMF models. This way, EMF provides a set of predefined JET templates that generate the Java code implementing the editor for a given Ecore meta-model. We have overwritten those templates to extend the generated tree editor with a "Recommender" pop-up menu on the objects that may be target of recommendations. This menu shows, for a selected object, the kinds of items that can be recommended. This information (i.e., the kinds of recommendation targets and items, see lines 5–11 in Listing 1) is not hard-coded in Java, but stored in a configuration file called *recommender.properties*. This permits building the "Recommender" menu dynamically upon clicking on an object, and facilitates the external evolution of the menu. Upon selecting a recommendation item kind for an object, a request is sent to the Droid service, passing

[1] https://eclipse-ee4j.github.io/jersey/
[2] http://tomcat.apache.org/
[3] https://projects.eclipse.org/projects/modeling.m2t.jet

the object, its context and the item type as parameters. The response is a list of recommendations, which are displayed in a table ordered by their relevance. The users can then select recommendations and apply them to the current model.

As an example, next, we show the integration of a RS specified with Droid, within the default tree editor generated for a simple modelling language for object-oriented design. The RS recommends attributes, methods and superclasses for classes. We do not use the running example to illustrate our client integration, since our approach requires starting from an Ecore meta-model and generates the whole editor from scratch, while the UML modelling editor has not been created using the JET templates predefined in EMF. The main concepts used in both examples are similar though.

Fig. 7 shows the use of the RS within the generated tree editor. The package explorer (label 1) contains a project with a model and the *recommender.properties* configuration file. The model is being edited in the window to the right (label 2). Right-clicking on any object of type Klass (label 3) shows the "Recommender" pop-up menu (label 4). This menu contains a submenu for each available kind of recommendation (in

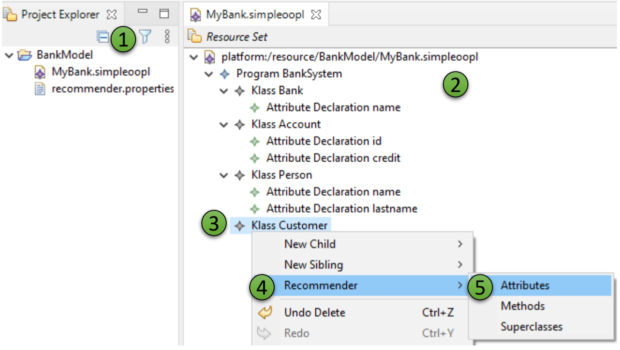
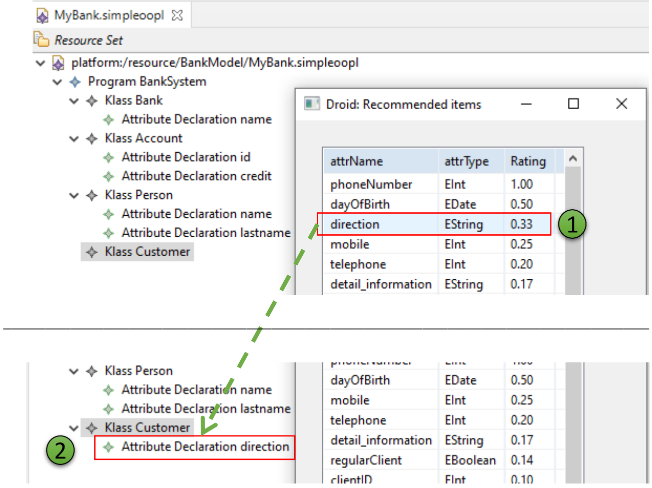


Figure 7. Selecting the recommendation item kind

The upper part of Fig. 8 shows the result of selecting the submenu "Attributes" on a Klass named Customer. A list of recommended attributes is presented to the user, including their name, type and rating (i.e., trust on the recommendation). When the user selects an attribute ("direction" in the figure, label 1), this is automatically added to the Klass Customer (label 2) and removed from the list.

6 Evaluation

With the aim to check the usefulness of the recommendations provided by Droid RSs, Section 6.1 reports on an offline evaluation with UML class models. To assess the feasibility of using Droid RSs outside Eclipse, Section 6.2 presents a case study that integrates a Droid RS with a modelling chatbot [42]. Finally, Section 6.3 discusses threats to validity.


MyBank.simpleoopl
Resource Set
platform:/resource/BankModel/MyBank.simpleoopl
Program BankSystem
Klass Bank
Attribute Declaration name
Klass Account
Attribute Declaration id
Attribute Declaration credit
Klass Person
Attribute Declaration name
Attribute Declaration lastname
Klass Customer
Droid: Recommended items
1
Klass Person
Attribute Declaration name
Attribute Declaration lastname
Klass Customer
2
Attribute Declaration direction

attrName	attrType	Rating
phoneNumber	EInt	1.00
dayOfBirth	EDate	0.50
direction	EString	0.33
mobile	EInt	0.25
telephone	EInt	0.20
detail_information	EString	0.17

dayOfBirth	EDate	0.50
mobile	EInt	0.25
telephone	EInt	0.20
detail_information	EString	0.17
regularClient	EBoolean	0.14
clientID	EInt	0.10

Figure 8. Selecting and applying a recommendation

6.1 Usefulness of recommendations

The goal of this first experiment is to answer the research question (RQ) RQ1: "*How precise and complete are the recommendations provided by Droid recommenders?*". To this aim, we performed the offline experiment that is reported next.

6.1.1 Experiment setup. We ran an offline experiment on two datasets from two different domains. The purpose was to analyse the performance of the RSs generated with Droid on distinct domains.

The used datasets contain models extracted from MAR [25]. This is a structure-based search engine for models and meta-models, which can be queried via input keywords. In particular, we retrieved UML models, since they are the most numerous in MAR. As domains for our experiment, we chose *Literature* and *Education*. The keywords used to retrieve the models for the *Literature* domain were *bibliography*, *book*, *author*, *journal* and *magazine*. The keywords used for the *Education* domain were *professor*, *teacher*, *student* and *alumn* (as stem of other words like *alumnus* or *alumni*). The resulting datasets are available at https://github.com/Droid-dsl/DroidConfigurator.

Table 1 shows, per each domain, the number of models, users (i.e., classes), items (i.e., attributes, methods and superclasses) and features (i.e., attributes describing users and items) in the datasets. The *Literature* and *Education* datasets have 1,447 and 1,051 UML models, respectively, conformant to the UML 2.0 class diagrams meta-model (cf. Fig. 3). The table does not consider duplicate elements. Hence, if two models contain classes with the same name, they are considered to be the same class. This is more evident in the *Education* domain, which has more models than users.

6.1.2 Experiment. We used Droid to configure a RS for each domain, selecting all available recommendation methods with different parameters. Specifically, we used the

Table 1. Description of the datasets.

	Literature	Education
Num. models	1,447	1,051
Num. users	1,740	905
Num. items	6,731	3,317
Num. features	6,497	3,231

Droid specification shown in Listing 1, and so, we trained multiple RSs through a variety of collaborative, content-based and hybrid recommendation methods: item popularity (*ItemPop*), item-based collaborative filtering (*IBCF*), user-based collaborative filtering (*UBCF*), content-based with cosine similarity (*CosineCB*), content-based item-based (*CBIB*) and content-based user-based (*CBUB*). We parameterised the methods *IBCF*, *UBCF*, *CBIB* and *CBUB* with neighbourhood sizes *k* 10, 15, 20, 25, 50 and 100. In the following, we refer to the methods that use neighbourhoods by concatenating the method name and the neighbourhood size *k*. For instance, IBCF50 refers to the IBCF *k*-NN method with 50 neighbours.

In all cases, we used *10-fold cross-validation* and a *per-user* technique to split the datasets (cf. Section 4). We analysed the performance of the RSs by means of the ranking quality metrics *precision* (p), *recall* (r), *F1*, *MAP* (Mean Average Precision) and *nDCG* (Normalized Discounted Cumulative Gain); and the coverage and diversity metrics *USC* (User Space Coverage) and *ISC* (Item Space Coverage). Additionally, in the experiment, we used a relevance threshold of 0.5, and cut-offs 5, 10, 15 and 20.

6.1.3 Experiment results. Table 2 shows the results of the experiment for each domain/dataset (*Literature* and *Education*). The rows show the selected recommendation methods, and the columns correspond to the metric values. For space constraints, the table omits the results of the recommendation methods *IBCF* and *CBIB*, as their performance is worse than that of *UBCF* and *CBUB*.

We can observe that the order of magnitude of the metric values is the same in both domains. As studied in the RS field [7], this magnitude depends on many factors, such as the dataset characteristics (e.g., the average number of preferences per user, or the rating sparsity, which is the proportion of existing ratings from the whole set of potential user-item preference relations), and the evaluation methodology (e.g., the method to split training and test data, or the test ratings for which the metrics are computed). In our experiment, we followed the TestItems methodology [7] which, for a target user, evaluates recommendation lists that may contain test items from all users. This explains why the precision values are close to 0. For this reason, in general, the important aspect to consider is the relative difference of the metric values achieved by the different recommendation methods.

Analysing Table 2, a first conclusion is the fact that the content-based method *CosineCB* was the worst performing, being outperformed even by the *ItemPop* baseline. This is not surprising in our experiment. *CosineCB* estimates the preference of a user (class) for an item (attribute, method, or superclass) by means of the cosine of the angle between the user and item feature vectors. These feature vectors correspond to the names of the classes, attributes and methods in the models of the datasets. Since we do not perform any text pre-processing on those names (e.g., to unify lowercase and uppercase, singular and plural, morphological deviations, misspellings, synonyms, ambiguities), there are different names that could have been considered the same, facilitating the cosine similarity. Moreover, we may have used finer-grained user and item profiles which capture the occurrence frequency of features.

By contrast, *UBCF* and *CBUB* were the best performing recommendation methods. The results of their item-based counterparts were worse, and are not reported in the table. *UBCF* with neighbourhoods of sizes 10 and 15 achieved the best *F1* values in both domains. In terms of *MAP* and *nDCG*, which focus on the precision of the top positions in the recommendation lists, the best results were obtained with neighbourhoods of sizes 20 and 25 in the *Education* domain, and sizes 50 and 100 in the *Literature* domain. If we consider *F1*, *MAP* and *nDCG* all together, *UBCF* with neighbourhood size 15 seems the best choice for the available data and targeted task.

As expected, since *CosineCB* and *ItemPop* do not depend on user-item rating patterns, they have an *USC* of 1, which means that they are able to make recommendations for 100% of the users. In terms of *ISC* diversity, there is no significant difference between methods and domains, which reflects that both popular and unpopular items are recommended.

Table 3 shows the precision and recall of the recommendation methods on both domains per cut-off values, p@k and r@k, focusing on the first k = 5, 10, 15 and 20 recommendations. We observe that the higher the value k, the lower the precision and the higher the recall. Again, *CosineCB* was outperformed by *ItemPop*. As we explained above, the poor performance of *CosineCB* can be improved by performing some text pre-processing, which we plan to address in future work. However, even with raw data, these results are in-line with the precision reported by other RSs for class diagrams [10] (around 0.04). Although not shown in the table, *UBCF* outperformed *IBCF*. The hybrid use of content-based and collaborative filtering techniques did not improve the recommenders based on a single technique. When considering both p@k and r@k, *UBCF* with neighbourhood size 50 was the best performing method.

Answering RQ1, our evaluation shows that standard recommendation methods are able to provide sensible recommendations for every class, starting from relatively small datasets that have not been pre-processed. These results are in-line with RSs specifically created for class diagrams [10]. Still, we have identified some aspects that would allow

Table 2. Results of the experiment. The best values are shown in bold.

Method	Literature							Education						
	p	**r**	**F1**	**MAP**	**nDCG**	**USC**	**ISC**	**p**	**r**	**F1**	**MAP**	**nDCG**	**USC**	**ISC**
ItemPop	0.006	0.180	0.012	0.055	0.086	**1.000**	0.012	0.007	0.224	0.013	0.082	0.117	**1.000**	0.017
CosineCB	0.001	0.032	0.002	0.017	0.020	**1.000**	0.004	0.002	0.076	0.004	0.003	0.017	**1.000**	0.007
UBCF10	**0.033**	0.290	**0.060**	0.157	0.195	0.824	0.059	**0.035**	0.337	**0.064**	0.184	0.227	0.830	0.056
UBCF15	**0.026**	0.304	**0.048**	0.160	0.201	0.860	**0.060**	**0.026**	0.346	**0.048**	**0.184**	0.228	0.863	**0.057**
UBCF20	0.022	0.319	0.041	**0.161**	0.205	0.863	**0.060**	0.022	0.360	0.042	0.183	**0.232**	0.868	**0.057**
UBCF25	0.020	0.327	0.038	**0.163**	0.208	0.865	**0.060**	0.021	0.369	0.039	**0.184**	**0.235**	0.868	**0.057**
UBCF50	0.019	**0.348**	0.037	0.159	**0.211**	0.865	0.058	0.018	**0.383**	0.035	0.176	0.231	0.868	**0.057**
UBCF100	0.020	**0.360**	0.037	0.155	**0.210**	0.865	0.056	0.019	**0.387**	0.035	0.166	0.224	0.868	0.055
CBUB10	0.015	0.202	0.028	0.108	0.132	0.929	0.055	0.023	0.258	0.042	0.137	0.168	0.926	0.053
CBUB15	0.011	0.210	0.022	0.105	0.130	0.962	0.056	0.015	0.260	0.029	0.135	0.165	0.984	0.054
CBUB20	0.009	0.213	0.017	0.102	0.129	0.963	0.056	0.012	0.265	0.022	0.133	0.165	**1.000**	0.055
CBUB25	0.008	0.212	0.015	0.098	0.125	0.987	0.056	0.010	0.271	0.019	0.133	0.166	**1.000**	0.055
CBUB50	0.006	0.212	0.011	0.088	0.117	**1.000**	0.055	0.008	0.304	0.016	0.133	0.176	**1.000**	0.055
CBUB100	0.007	0.242	0.014	0.097	0.133	**1.000**	0.051	0.008	0.302	0.016	0.124	0.169	**1.000**	0.052

Table 3. Results of the experiment per cut-offs. The best values are shown in bold.

Method	Literature								Education							
	p@5	**p@10**	**p@15**	**p@20**	**r@5**	**r@10**	**r@15**	**r@20**	**p@5**	**p@10**	**p@15**	**p@20**	**r@5**	**r@10**	**r@15**	**r@20**
ItemPop	0.022	0.014	0.012	0.010	0.072	0.090	0.110	0.123	0.027	0.018	0.015	0.012	0.099	0.129	0.154	0.168
CosineCB	0.003	0.002	0.002	0.001	0.016	0.018	0.020	0.021	0.001	0.001	0.001	0.001	0.001	0.004	0.006	0.007
UBCF10	0.053	0.033	0.025	0.020	0.194	0.231	0.253	0.265	0.061	0.037	0.027	0.022	0.228	0.270	0.295	0.308
UBCF15	0.055	0.034	0.026	0.021	0.200	0.237	0.259	0.272	0.060	0.038	0.028	0.022	0.227	0.272	0.299	0.313
UBCF20	0.057	0.036	0.027	0.022	0.205	0.245	0.267	0.282	**0.062**	0.038	0.029	0.023	**0.232**	0.276	0.305	0.320
UBCF25	0.058	0.037	0.028	0.022	0.207	0.250	0.274	0.289	**0.063**	**0.039**	**0.029**	0.023	**0.235**	**0.281**	**0.310**	**0.329**
UBCF50	**0.060**	**0.038**	**0.029**	**0.023**	**0.212**	**0.261**	**0.286**	**0.302**	0.062	**0.039**	**0.030**	**0.024**	0.229	**0.279**	**0.313**	**0.335**
UBCF100	**0.060**	**0.039**	**0.029**	**0.024**	**0.210**	**0.260**	**0.287**	**0.306**	0.059	0.038	0.029	**0.024**	0.217	0.270	0.302	0.327
CBUB10	0.031	0.018	0.013	0.011	0.139	0.163	0.176	0.184	0.040	0.024	0.018	0.014	0.178	0.209	0.226	0.237
CBUB15	0.030	0.019	0.014	0.011	0.135	0.165	0.180	0.188	0.039	0.024	0.018	0.014	0.174	0.206	0.226	0.236
CBUB20	0.029	0.019	0.014	0.011	0.130	0.168	0.184	0.193	0.038	0.024	0.018	0.014	0.170	0.205	0.225	0.236
CBUB25	0.028	0.018	0.014	0.011	0.124	0.162	0.180	0.190	0.038	0.024	0.018	0.014	0.169	0.206	0.227	0.239
CBUB50	0.025	0.016	0.013	0.011	0.114	0.142	0.167	0.183	0.044	0.027	0.021	0.017	0.173	0.211	0.243	0.261
CBUB100	0.034	0.022	0.017	0.014	0.129	0.163	0.182	0.197	0.043	0.026	0.021	0.017	0.166	0.202	0.230	0.252

improving the generated recommendations, such as using larger datasets, pre-processing the text features that the content-based methods exploit, or even incorporating more specific, task-oriented recommendation methods.

6.2 Case study on RS integration

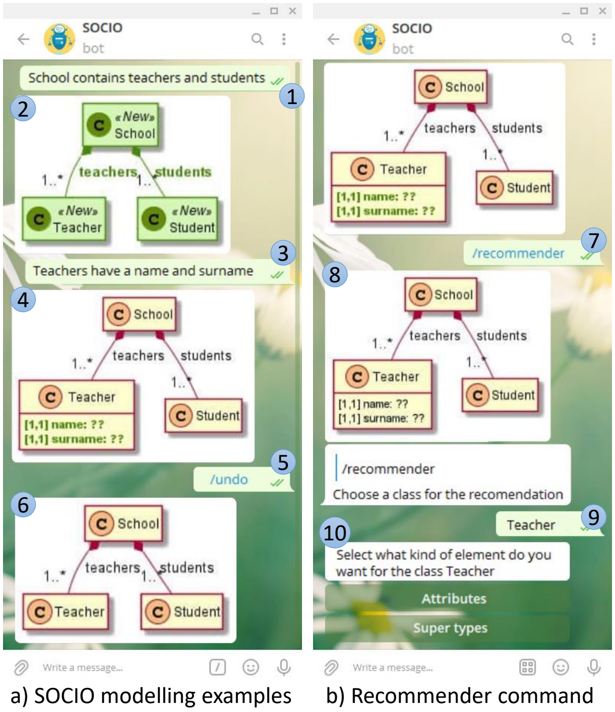
This section shows a case study on the integration of a RS specified with Droid into a modelling chatbot called Socio. With this study, we aim to answer the following RQ (RQ2): *How difficult is it to integrate a Droid-based RS with a non-Eclipse-based modelling client?*

Socio [42] is a chatbot or conversational agent that enables heterogeneous groups of domain and modelling experts to collaborate on modelling tasks. It works in social networks, like Telegram or Twitter, and facilitates the active participation of domain experts with no technical background in building models (class diagrams) by using natural language (NL) as the modelling interface.

Fig. 9(a) shows a user interaction with Socio in Telegram. The user can send messages expressing domain requirements in NL to the chatbot (labels 1 and 3). Socio interprets the messages and the current status of the model, infers the necessary modelling actions, updates the model, and sends back an image of the model with the modified elements in green (labels 2 and 4). For example, given the message "School contains teachers and students" (label 1), Socio infers that there must be three classes named *School*, *Teacher* and *Student*. Then, because of the *contains* verb, it infers that *School* should have two containment references with cardinality one to many (as teachers and students are plural), one called *teachers* and going to *Teacher*, and the other called *students* and going to *Student*. Since the model is empty at this moment, Socio creates all these elements (label 2).

Users normally do not provide all requirements in a single message, and so, Socio permits a model to be incomplete or incorrect. The interaction with label 3 illustrates this. When the user says "Teachers have a name and surname", Socio interprets that there must be a class named *Teacher* with two features, *name* and *surname*. Since the class already exists, it only adds the two features, but since there is no information about their types, their definition is incomplete (label 4).

Besides model creation via NL processing, the chatbot has commands to manage, validate, download the model, or undo and redo the modelling actions. In Telegram, these commands start by a backslash followed by a keyword. Labels 5 and 6 in Fig. 9(a) show an example of the undo command.



School contains teachers and students

Teachers have a name and surname

/undo

/recommender

/recommender

Choose a class for the recomendation

Teacher

Select what kind of element do you want for the class Teacher

Attributes

Super types

a) SOCIO modelling examples b) Recommender command

Figure 9. Example of Socio interaction in Telegram.

For this case study, we extended Socio with a RS specified with Droid and hence available as a service a scheme of the integration of the RS withir the new components are highlighted in gree front-end provided by Telegram and a back-er the main component of the architecture sinc the functionality of Socio: information and

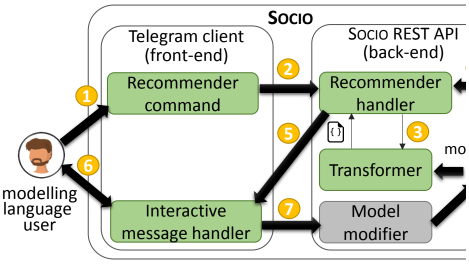
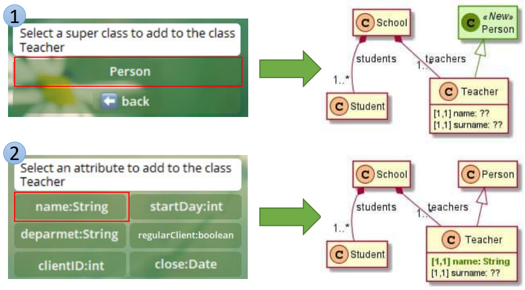


Figure 10. Architecture of Droid integrati

For the integration, we created a Recommender command on the client side (label 1 in Fig. 10). When the user types this command to obtain recommendations (label 1), the Telegram client sends a request to the back-end, which is handled by the *Recommender handler* (label 2). Since the model is internally represented with EMF, a *Transformer* converts the context element of the recommendation into the JSON format required by Droid (label 3). Then, the *Recommender handler* requests recommendations to the Droid service (label 4), and sends the returned recommendations back to the client (label 5). In the client, an *Interactive message handler* transforms the recommendations into an interactive message containing one button per recommended item (label 6). When the user selects one of these buttons, the handler sends a request to the back-end to add the selected item to the model (label 7). Then, the selected button is deleted, while the other buttons remain available to permit applying further recommendations.

Fig. 9(b) shows the usage of the /recommender command in Telegram. When a user types the command (label 7), Socio displays the current model and prompts the user to select a class (label 8). Once the user selects a class (label 9), Socio asks the kind of items to be recommended (label 10). Since Socio models do no support methods, the user can choose the recommendation of attributes and supertypes.

Fig. 11 illustrates the recommendations provided by Droid. It shows the recommended supertypes (label 1) and attributes (label 2) for the class *Teacher*. When the user presses the button with the recommendation *Person*, Socio creates a new class because it does not exist, and adds it as a supertype of *Teacher*. When the user presses the button with the recommendation *name*, Socio detects that *Teacher* already defines this attribute and only updates its type. This way, recom-



Select a super class to add to the class Teacher

Person

back

Select an attribute to add to the class Teacher

name:String startDay:int

deparmet:String regularClient:boolean

clientID:int close:Date

Figure 11. Droid recommendations in Socio.

Table 4 shows the LOC and number of Java classes developed to achieve the RS integration. The *Interactive message handler* is the largest component, which is normal as it handles several user interactions. We can observe that the integration did not require many changes in the Socio architecture, and the new components are not large.

Answering RQ2, this case study proves that Droid-based RSs can be easily integrated with tools outside Eclipse. While the integration with Socio has not many LOC, we added code on both its front-end and its back-end. Moreover, more than 50% of the code was dedicated to the user interaction.

Table 4. Metrics for integrating Droid with Socio.

		LOC	Num. Classes
Back	Recommender handler	160	2
	Transformer	44	1
Front	Recommender command	128	2
	Interac. message handler	400	1
	Total	732	6

These two circumstances can make a big difference in the effort required to integrate the RS with other modelling tools.

6.3 Threats to validity

Next, we discuss threats to validity in our evaluations.

With respect to the offline experiment of Section 6.1, we tried to minimize the threats to its external validity by making use of two independent and large datasets from two different domains. However, the domain selection and the keywords chosen for the query may affect the generality of the results. To tackle this issue, we plan to conduct experiments with other domains and datasets in the future. Another threat is the specific recommendation task accomplished in the experiment, namely the completion of class diagrams with new attributes, methods and superclasses. Hence, further experiments are needed to draw conclusions on the use of our approach for other recommendation tasks. Concerning internal validity, we tried to avoid any bias on the results by the use of third-party datasets.

Regarding the case study reported in Section 6.2, the results are specific to Socio and cannot be generalized, which remains the main threat to the external validity. However, the integration of Socio was specially challenging due to its distributed architecture and its independence of Eclipse; hence, we expect that the effort to integrate a Droid-based RS in other clients will not be higher than for Socio.

7 Related Work

In this section, we review the two main areas of related works: recommenders for modelling languages, and automated approaches for the synthesis of RSs.

7.1 Recommenders for modelling languages

According to [4], the most common usage purposes for recommenders in MDE are completion, finding, repair, reuse, and to a lesser extent, creation of modelling artefacts. The recommendations typically apply to models and meta-models, while recommenders for model transformations and code generators are scarce. Droid can be applied to any kind of artefact, provided that it is defined by a meta-model.

Most recommenders for modelling languages target UML, especially class diagrams. IPSE [20] has a knowledge-based RS that guides students on creating class diagrams, and the recommendations build on Prolog constraints defined by the teacher. RapMOD [31] recommends relevant autocompletion actions for graphical UML class diagrams. REBUILDER [23] relies on case-based reasoning, Bayesian networks and WordNet to recommend class diagrams similar to a given one. Elkamel et al. [16] use similarity metrics to recommend similar classes to the ones in the current class diagram. Other researchers propose RSs for other UML diagrams: Cerqueira et al. [12] propose a CB approach for recommending behavioural features for UML sequence diagrams, and Aquino et al. [5] present a recommender of actors and use cases for use case diagrams. While these works tackle useful modelling tasks, they serve a specific modelling language and the recommendation method is fixed. Instead, Droid is not UML-specific but it permits customizing the target modelling language, the kind of items to be recommended, and the recommendation algorithm.

Some approaches aim to provide semantically related terms and context-sensitive information for a modelling task. Burgueño et al. [10] propose a domain concept recommender based on the analysis of the textual information available on the domain model being constructed, as well as on general knowledge about the business domain. The domain modelling tool DoMoRe [2] exploits a knowledge base of domain-specific terms and their relationships to provide context-sensitive recommendations. Other tools, like Extremo [35] or the assistant envisioned by Savary-Leblanc [51], employ semantic similarity based on lexical databases like WordNet to recommend semantically related terms. While these tools target a specific modelling task, our framework is generic and configurable for arbitrary modelling languages.

Recommenders have also been applied to business process modelling. For example, to recommend complete process models based on the user profile [28], as well as finer-grained recommendations that pursue completing a process model with new fragments [30], activity nodes [14, 32], tasks [44] or actor roles [44]. Again, these works are specific to a modelling language, and the recommendation method is fixed.

In contrast to the previous language-specific approaches, others are language-independent. These are typically applicable to arbitrary modelling languages defined in a given meta-modelling framework, such as EMF. For example, PARMOREL [6, 26] uses reinforcement learning to repair malformed EMF models based on the user preferences and the experience gained from previous repairs. ReVision [41] suggests consistency-preserving model editing rules for model repair. SimVMA [54] uses clone detection to help modellers find models or operations relevant to them. Finally, Kögel [29] proposes to analyse the history of past model changes to suggest recommendations, and foresees the use of machine learning, heuristic search algorithms, association rules and decision trees. Altogether, even though these works plan on frameworks for different languages, the recommendation method is fixed, and the recommendations cannot be customised, as we can do using Droid.

7.2 Recommender system generation

While we can find many RSs for modelling languages, most were developed by hand from scratch, which requires a high effort [36]. Hence, recent studies [4] have identified the need of methods and tools automating the construction of recommenders for modelling languages. This work aims to fill this gap. Next, we compare with other related approaches.

Fellmann et al. [19] define a reference model with the data perspective requirements of RSs for process modelling. The model can be instantiated as a guide for developing new process modelling recommenders, or to assess existing ones. While useful, the approach is specific to process modelling, and does not provide automation or code synthesis.

Rojas et al. [48] present an MDE framework to create mobile RSs of geographic points of interest. The framework helps defining the structural, behavioural and navigational aspects of the RS, and customising the user preferences, similarity metrics and similarity formula. In [47], a similar solution is used to recommend trips and tours. However, in both works, the target domain of the recommendation is fixed.

We also find MDE proposals to support non-expert users on applying data mining. For example, Espinosa at al. [17, 18] reuse the past experiences of data mining experts to compute the accuracy for a given new dataset and recommend the one with the best performance. The framework permits customising the data mining task to perform, the evaluation method and metrics, and the mining algorithm. Even though this solution offers the flexibility and benefits of MDE, the generated recommenders are data mining applications.

In a more general setting, Hermes [15] is a generic framework to build recommenders for modelling environments. Its extensible architecture permits defining new recommendation strategies, new widgets to trigger and display the recommendations, and new contexts to adapt the recommendations to the modelling environment. These elements are coded as extensions of base classes, or registered in the case of resources like icons and labels. Hermes provides a dashboard to define the class extensions, and supports the manual testing of the recommender. In contrast, our DSL Droid does not require coding, but it provides a simple syntax to configure the kinds of recommended items, the recommendation method, and its evaluation based on standard metrics. Moreover, it automatically generates a tailored RS as a web service to make it available from arbitrary environments.

More similar to our proposal, the vision paper [52] foresees a lowcode development environment where end users can define RSs by using graphical interfaces, drag-and-drop utilities and forms. The authors aim to support the construction of arbitrary RSs, not specific for modelling languages. The lowcode environment will build on a generic meta-model to provide components implementing recurring functionalities for RSs, such as data pre-processing, capturing context, and producing and presenting recommendations. The authors foresee having several DSLs to configure each aspect of the recommender. Our philosophy is similar, but we focus on RSs for modelling. This way, our DSL allows the fine-grained specification of the recommendation target and items, and our tooling generates a RS available as a REST API that can be integrated in other tools.

8 Conclusions and Future Work

RSs are increasingly being used in Software Engineering, and MDE is no exception to this trend. Since building RSs for DSLs is time expensive, we have developed a model-based approach to automate their construction. The approach provides a DSL to configure the target of the recommendation and the type of the recommended items, and supports the evaluation of the RSs to identify the best one for the problem at hand. Our solution relies on a generic recommendation service that can be integrated out-of-the-box with the EMF tree editor for models. We have demonstrated the feasibility of its integration with non-Eclipse tools, and have evaluated the precision and completeness of the recommendations.

In practice, the creation of RSs for modelling requires having big sets of models for training. These exist for popular modelling languages (e.g., UML, BPMN, Simulink), but not for other DSLs. We trust that the emergence of dedicated model search engines [25] and repositories will facilitate this task. Moreover, there are other options. First, RSs can be trained with the available models, and retrained as more models become available. Second, one may apply "transfer learning" for some DSLs, i.e., training the RS with models of another similar DSL. For example, one may build a RS for UML class diagrams, and apply it to Ecore meta-models.

In the future, we plan to work on pre-processing techniques for the model sets. For instance, for the UML class recommender, it could be useful to pre-process names (e.g., deleting blank spaces) and cluster semantically similar names. We would also like to enrich the recommendation context, e.g., including classes related to the recommendation target. We have focussed on classical recommendation methods, but we aim to make Droid a DSL front-end to configure arbitrary recommendation methods, including those specific for modelling tasks. For this purpose, we are making our architecture extensible via extension points, to be implemented for specific methods. Another line to explore is to gather recommendation feedback from the users, and adjust future recommendations based on it. Finally, we plan to make a user study to identify strengths and weaknesses of our proposal.

Acknowledgments

This project has received funding from the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884, the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

References

[1] Gediminas Adomavicius and Alexander Tuzhilin. 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering* 17, 6 (2005), 734–749.

[2] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A recommender system for domain modeling. In *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 71–82.

[3] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2020. Towards automating the construction of recommender systems for low-code development platforms. In *Proc MODELS Companion Proceedings*. ACM, 66:1–66:10.

[4] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Recommender systems in model-driven engineering: A systematic mapping review. *Software and System Modeling* in press (2021).

[5] Erika Rizzo Aquino, Pierre de Saqui-Sannes, and Rob A. Vingerhoeds. 2020. A methodological assistant for use case diagrams. In *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 227–236.

[6] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Improving model repair through experience sharing. *Journal of Object Technology* 19, 2 (2020), 13:1–21.

[7] Alejandro Bellogín, Iván Cantador, and Pablo Castells. 2013. A comparative study of heterogeneous item recommendations in social systems. *Information Sciences* 221 (2013), 142–169.

[8] Markus Borg, Krzysztof Wnuk, Björn Regnell, and Per Runeson. 2017. Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context. *IEEE Transactions on Software Engineering* 43, 7 (2017), 675–700.

[9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers, San Rafael, California (USA).

[10] Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, and Jordi Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In *CAiSE (LNCS)*, Vol. 12751'. Springer International Publishing, 91–106.

[11] Robin Burke. 2002. Hybrid recommender systems: Survey and experiments. *User Modeling and User-adapted Interaction* 12, 4 (2002), 331–370.

[12] Thaciana Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A content-based approach for recommending UML sequence diagrams. In *28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 644–649.

[13] Marcos César de Oliveira, Davi Freitas, Rodrigo Bonifácio, Gustavo Pinto, and David Lo. 2019. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software* 158 (2019).

[14] ShuiGuang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2017. A recommendation system to facilitate business process modeling. *IEEE Transactions on Cybernetics* 47, 6 (2017), 1380–1394.

[15] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 282–290.

[16] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. 2016. An UML class recommender system for software design. In *13th IEEE/ACS International Conference of Computer Systems and Applications (AICCSA)*. IEEE Computer Society, 1–8.

[17] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2013. Development of a knowledge base for enabling non-expert users to apply data mining algorithms, In SIMPDA. *CEUR Workshop Proceedings* 1027, 46–61.

[18] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2019. S3Mining: A model-driven engineering approach for supporting novice data miners in selecting suitable classifiers. *Computer Standards and Interfaces* 65 (2019), 143–158.

[19] Michael Fellmann, Dirk Metzger, Sven Jannaber, Novica Zarvic, and Oliver Thomas. 2018. Process modeling recommender systems - A generic data model and its application to a smart glasses-based modeling environment. *Bus. Inf. Syst. Eng.* 60, 1 (2018), 21–38.

[20] H. Garbe. 2012. Intelligent assistance in a problem solving environment for UML class diagrams by combining a generative system with constraints. In *eLearning*. IADIS, 412–416.

[21] Marko Gasparic and Andrea Janes. 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software* 113 (2016), 101–113. https://doi.org/10.1016/j.jss.2015.11.036

[22] Github. 2021. Copilot. https://copilot.github.com/.

[23] Paulo Gomes. 2004. Software design retrieval using Bayesian networks and WordNet. In *7th European Conf. on Advances in Case-Based Reasoning (ECCBR) (Lecture Notes in Computer Science)*, Vol. 3155. Springer, 184–197.

[24] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In *Recommender Systems Handbook*. Springer, 265–308.

[25] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2020. MAR: a structure-based search engine for models. In *MoDELS '20*. ACM, 57–67.

[26] Ludovico Iovino, Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Model repair with quality-based reinforcement learning. *Journal of Object Technology* 19, 2 (2020), 17:1–21.

[27] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.

[28] Hadjer Khider, Slimane Hammoudi, and Abdelkrim Meziane. 2020. Business process model recommendation as a transformation process in MDE: Conceptualization and first experiments. In *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 65–75.

[29] Stefan Kögel. 2017. Recommender system for model driven software development. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 1026–1029.

[30] Agnes Koschmider, Thomas Hornung, and Andreas Oberweis. 2011. Recommendation-based editor for business process modeling. *Data & Knowledge Engineering* 70, 6 (2011), 483–503.

[31] Tobias Kuschke and Patrick Mäder. 2017. RapMOD - in situ autocompletion for graphical models: poster. In *39th International Conference on Software Engineering (ICSE), Companion Volume*. IEEE Computer Society, 303–304.

[32] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, ShuiGuang Deng, Yuyu Yin, and Zhaohui Wu. 2014. An efficient recommendation method for improving business process modeling. *IEEE Transactions on Industrial Informatics* 10, 1 (2014), 502–513.

[33] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. 2011. Content-based recommender systems: State of the art and trends. In *Recommender Systems Handbook*. Springer, 73–105.

[34] Pyry Matikainen, P. Michael Furlong, Rahul Sukthankar, and Martial Hebert. 2013. Multi-armed recommendation bandits for selecting state machine policies for robotic systems. In *2013 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 4545–4551.

[35] Ángel Mora Segura, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. 2018. Automated modelling assistance by integrating heterogeneous information sources. *Computer Languages, Systems and Structures* 53 (2018), 90–120.

[36] Gunter Mussbacher, Benoît Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien

Mosser, Houari A. Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. 2020. Opportunities in intelligent modeling assistance. *Softw. Syst. Model.* 19, 5 (2020), 1045–1053.

[37] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. FOCUS: a recommender system for mining API function calls and usage patterns. In *41st International Conference on Software Engineering (ICSE).* IEEE / ACM, 1050–1060.

[38] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020).

[39] Xia Ning, Christian Desrosiers, and George Karypis. 2015. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender Systems Handbook*. Springer, 37–76.

[40] OCL. 2014. http://www.omg.org/spec/OCL/.

[41] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: a tool for history-based model repair recommendations. In *40th International Conference on Software Engineering (ICSE), Companion Proceeedings*. ACM, 105–108.

[42] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2018. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Softw.* 35, 6 (2018), 48–54.

[43] Ana Pescador and Juan de Lara. 2016. DSL-maps: from requirements to design of domain-specific languages. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, 438–443. https://doi.org/10.1145/2970276.2970328

[44] Mohammad Ehson Rangiha, Marco Comuzzi, and Bill Karakostas. 2015. Role and task recommendation and social tagging to enable social business process management. In *BPMDS/EMMSAD@CAiSE (Lecture Notes in Business Information Processing)*, Vol. 214. Springer, 68–82.

[45] Z. Reitermanová. 2010. Data splitting. In *WDS*. Matfyzpress, 31–36.

[46] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. 2010. Recommendation systems for Software Engineering. *IEEE Software* 27, 4 (2010), 80–86.

[47] Gonzalo Rojas, Francisco Dominguez, and Stefano Salvatori. 2009. Recommender systems on the Web: A model-driven approach. In *E-Commerce and Web Technologies*, Tommaso Di Noia and Francesco Buccafurri (Eds.). Springer Berlin Heidelberg, 252–263.

[48] Gonzalo Rojas and Claudio Uribe. 2013. A conceptual framework to develop mobile recommender systems of points of interest. In *SCCC*. IEEE Computer Society, 16–20.

[49] Alan Said and Alejandro Bellogín. 2014. Rival: a toolkit to foster reproducibility in recommender system evaluation. In *Eighth ACM Conference on Recommender Systems, RecSys '14*. ACM, 371–372. See also https://github.com/recommenders/rival.

[50] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. Quick fixing ATL transformations with speculative analysis. *Software and Systems Modeling* 17, 3 (2018), 779–813.

[51] Maxime Savary-Leblanc. 2019. Improving MBSE tools UX with AI-empowered software assistants. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Volume*. IEEE, 648–652.

[52] Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2020. Democratizing the development of recommender systems by means of low-code platforms. In *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, Esther Guerra and Ludovico Iovino (Eds.). ACM, 68:1–68:9.

[53] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, Upper Saddle River, NJ.

[54] Matthew Stephan. 2019. Towards a cognizant virtual software modeling assistant using model clones. In *41st International Conference on Software Engineering: New Ideas and Emerging Results (NIER@ICSE).* IEEE / ACM, 21–24.

[55] Masateru Tsunoda, Takeshi Kakimoto, Naoki Ohsugi, Akito Monden, and Ken-ichi Matsumoto. 2005. Javawock: A Java class recommender system based on collaborative filtering. In *17th International Conference on Software Engineering and Knowledge Engineering (SEKE).* 491–497.

[56] UML. 2017. UML 2.5.1 OMG specification. http://www.omg.org/spec/UML/2.5.1/.

[57] Saúl Vargas and Pablo Castells. 2011. Rank and relevance in novelty and diversity metrics for recommender systems. In *Fifth ACM Conference on Recommender Systems, RecSys '11*. ACM, New York, NY, USA, 109–116. See also http://ranksys.github.io/.

[58] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org. http://www.dslbook.org

[59] Xtext. 2021. http://www.eclipse.org/Xtext/. (last accessed in July 2021).

B.8 Automating the Measurement of Heterogeneous Chatbot Designs

Title	**Automating the measurement of heterogeneous chatbot designs**	
Publication	37th ACM Symposium on Applied Computing (SAC 2022)	
Core	2021 Core B	
Authors	Pablo C. Cañizares	Universidad Autónoma de Madrid
	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1145/3477314.3507255	
Date	April 2021	

Abstract Chatbots are being increasingly used to provide a natural language interface to all kinds of software services. However, while there are many platforms and tools for chatbot development, they typically lack support to statically measure properties of the designed chatbots, as indicators of their size, complexity, quality or usability, and facilitating comparison.

To attack this problem, in this paper we propose a suite of 20 metrics for chatbot designs. The metrics are defined on a neutral chatbot design language, becoming independent of the implementation platform. We have developed a tool, called Asymob, which supports the translation of chatbots defined in several platforms into this neutral format to perform the measurements. As a proof-of-concept, we evaluate the metrics over a collection of Dialogflow and Rasa chatbots from several sources and open-source repositories. Our metrics helped detecting quality issues statically and served as a basis for comparing chatbots from different origins and built using different technologies.

Automating the Measurement of Heterogeneous Chatbot Designs

Pablo C. Cañizares
Universidad Autónoma de Madrid
Madrid, Spain
pablo.cerro@uam.es

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

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

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

ABSTRACT

Chatbots are being increasingly used to provide a natural language interface to all kinds of software services. However, while there are many platforms and tools for chatbot development, they typically lack support to statically measure properties of the designed chatbots, as indicators of their size, complexity, quality or usability, and facilitating comparison.

To attack this problem, in this paper we propose a suite of 20 metrics for chatbot designs. The metrics are defined on a neutral chatbot design language, becoming independent of the implementation platform. We have developed a tool, called Asymob, which supports the translation of chatbots defined in several platforms into this neutral format to perform the measurements. As a proof-of-concept, we evaluate the metrics over a collection of Dialogflow and Rasa chatbots from several sources and open-source repositories. Our metrics helped detecting quality issues statically, and served as a basis for comparing chatbots from different origins and built using different technologies.

CCS CONCEPTS

• **Human-centered computing → Natural language interfaces**; • **General and reference → Metrics**; • **Social and professional topics → Quality assurance**;

KEYWORDS

Chatbot design, metrics, quality assurance

ACM Reference Format:
Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2022. Automating the Measurement of Heterogeneous Chatbot Designs. In *Proceedings of ACM SAC Conference (SAC'22).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Chatbots are becoming popular to access all sorts of services (e.g., banking, shopping, tourism, health) using conversation in natural language [36], and they are being increasingly used to assist in software engineering activities [17]. For this reason, many platforms are available for their construction [29], like Google's Dialogflow [12], Amazon Lex [18], IBM's Watson [38], the Microsoft bot framework [21], and many others by smaller companies and communities like Rasa [32], Pandorabots [26], or FlowXO [15].

While there are many chatbot implementation platforms, their support for chatbot quality assurance is limited [29]. Since chatbots are a kind of software, their construction should follow sound engineering principles. Some recent approaches [4, 5, 13] and tools [3] propose methods for testing chatbots. Dynamic testing is essential to ensure the quality of the resulting chatbot, but it requires having a functional, deployed chatbot; it demands high effort for creating testing phrases and oracles; and it is time-consuming.

In this paper, we propose the use of metrics as a tool to guide and control the quality of the chatbot throughout its development, becoming a complement to dynamic testing. Metrics are an accepted mechanism for assessing and controlling properties of software products and processes [14]. They are complementary to dynamic testing as they can be used at design time, even when the chatbot is not yet functional. They can discover issues (e.g., regarding the design complexity and size) which are not the target of dynamic testing, and can be used to trigger recommendations for the improvement of the chatbot design. However, to our knowledge, there is hardly any proposal for the practical application of metrics to chatbot designs.

We argue that static metrics for chatbots can be useful to detect potential problems related to user experience (e.g., complex conversation flows, hard-to-read chatbot answers); as indicators of chatbot complexity; to compare properties of heterogeneous chatbots; to discover chatbot commonalities and cluster similar chatbots; and to understand how different implementation platforms can impact on the chatbot design. Ultimately, the availability of metrics may have a notable impact on current bot development practices and tools, helping to increase the quality of chatbots.

To pursue this goal, we propose a suite of 20 static metrics for chatbot designs, and an accompanying tool called Asymob that supports their evaluation over heterogeneous chatbot implementations.

SAC'22, April 25 - April 29, 2022, Brno, Czech Republic

ACM ISBN 978-1-4503-8713-2/22/04…$15.00
https://doi.org/xx.xxx/xxx_x

To avoid reimplementing the metrics for every chatbot implementation platform, Asymob defines the metrics over a neutral design notation called Conga [28], and provides importers from several platforms into Conga. We report on an evaluation applying the metrics over Dialogflow and Rasa chatbots from public repositories, and over predefined chatbots provided by the implementation platforms. Our experiment reveals quality issues in some chatbots, and shows that the metrics can serve as a basis for comparing chatbots from different sources and built using different technologies.

The rest of the paper is organized as follows. Section 2 explains the basics about chatbots. Section 3 revises related work. Section 4 proposes a suite of chatbot metrics over a neutral chatbot design notation. Section 5 describes tool support, and Section 6 evaluates the metrics over chatbots from several sources (predefined, open-source repositories) and technologies (Rasa, Dialogflow). Finally, Section 7 concludes and outlines lines for future work.

2 AN OVERVIEW OF CHATBOTS

Chatbots are conversational software systems with a natural language interface to existing services, like those in banking or shopping. Figure 1 shows a diagram with their typical working scheme. Normally, the user starts the interaction by providing an *utterance* –

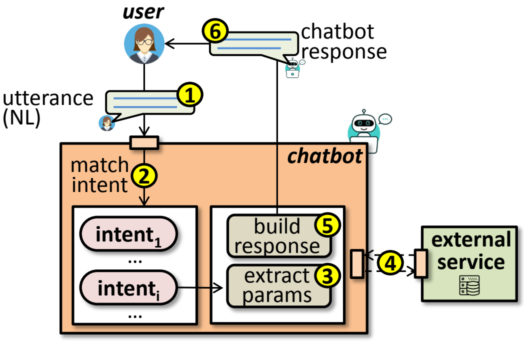


Figure 1: Chatbot working scheme.

Most chatbots are designed around a set of *intents*. These are conversation topics the chatbot aims at recognizing (step 2 in the figure), related to the offered functionality. Depending on the implementation platform, intents are defined either using *regular expressions* (e.g., as in Pandorabots [26]) or with *training phrases* that become interpreted using natural language processing (NLP). Additionally, intents can include *parameters* identifying relevant information pieces to be extracted from user utterances (step 3).

As an example, a chatbot for a cafeteria would define an intent to recognize the users' orders. This intent would be matched by phrases like *"I'd like a medium cappuccino"*, from which the chatbot would extract two parameters: the type of drink (*cappuccino*) and its size (*medium*). Parameters are typed by *entities*, which can be pre-existing in the platform (e.g., dates or numbers) or user-defined for a specific domain (e.g., the type and size of drinks). User-defined entities declare a list of literals (e.g., *medium* and *large* for the size of drinks) with their synonyms.

Upon receiving a user utterance, the chatbot matches the more likely intent, performs some predefined actions to handle the intent, such as accessing an external service (step 4), and composes a text/voice response (step 5) which may incorporate media elements (e.g., images, links) or widgets supported by the social network (e.g., buttons in Telegram). For example, the cafeteria chatbot may need to access an information system to check the availability of discounts and annotate the order, and the response may report the final price. If the chatbot lacks a matching intent for a user utterance, it can trigger a *fallback* intent to ask for clarification.

Typically, conversations are structured into *flows* that intertwine user utterances and chatbot responses. As an example, upon the user utterance *"I'd like a medium cappuccino"*, the chatbot may answer *"Would you like something to eat?"*, leading to a new user interaction (e.g., *"No thanks"*), and so on, according to the defined conversation flow within the chatbot design.

Moore and Arar [23] propose a classification of chatbots depending on their conversation style. *System-centric* chatbots answer user queries or interpret commands by means of 2-turn conversations (i.e., each user turn starts a new conversation and the chatbot lacks state). *Content-centric* chatbots act as an interface for FAQs, typically providing long document-like responses that may not be appropriate for voice-based interfaces or mobile devices with small screens. *Visual-centric* chatbots present buttons and other widgets to facilitate user interaction, in a style borrowed from mobile phones. Finally, *conversation-centric* chatbots mimic human conversations, offering conversation management utterances (e.g., "*What do you mean?*") and short responses. This latter type of chatbots are normally preferred because their conversation style suits a wider variety of devices and engages better in natural conversations.

3 RELATED WORKS ON CHATBOT QUALITY ASSESSMENT

Since the early days of conversational systems [39], researchers have proposed ways for evaluating their quality. For example, PARADISE [37] is an early framework based on the correlation of performance and user satisfaction.

Recently, the popularity of chatbots has raised concerns on proper conversational design. For example, IBM's Natural Conversation Framework [24] proposes conversation patterns [25] and design principles [23, 34]. The latter include guidelines like *recipient design* (i.e., allow multiple conversation paths for different user types), *minimization* (i.e., use concise chatbot answers), and *repair* (i.e., provide support for clarifications). In this line, *Chatbottest* [9] defines guidelines for chatbot design issues in categories like answering, error management, intelligence, navigation, personality and understanding. However, the burden is on the developer to manually test whether the chatbot fulfils the guidelines.

Literature reviews [27, 31] have also identified chatbot quality properties and ways to assess them. Radziwill and Benton align

[1] https://telegram.org/
[2] https://en.wikipedia.org/wiki/Amazon_Echo

bot quality attributes with the ISO-9241 notion of usability [1] (efficiency, effectiveness and satisfaction), while Peras [27] adds further categories (e.g., information retrieval, affect). Generally, the assessment of these quality properties relies on the dynamic execution of the chatbot, on collecting statistical data, or on subjective evaluations [10, 16, 22, 33]. Instead, our goal is to provide metrics that can be calculated automatically and statically on chatbot designs.

Other approaches assess quality via testing [7]. For instance, tools like Botium [3] or OggyBug [13] support test automation. Still, the developer has to provide a set of user utterances and expected chatbot answers within the envisioned conversation flows. To alleviate this burden, some works focus on the generation of challenging test user utterances [4, 5], e.g., via mutation of the training phrases defined for the intents. ChatEval [35] targets testing readability, which can be done statically by applying metrics (e.g., BLEU2 and average cosine similarity [19]) to the chatbot responses, and interactively by requiring the user to complete evaluation tasks. Instead, our goal is to provide complementary assessment mechanisms to testing in the form of metrics, which can be applied prior to deploying the chatbot and can reveal defects in the chatbot design.

As we will see in Section 4.2, some of our metrics profit from the work of the NLP community, which has developed useful readability metrics [19, 30]. For example, Pitler and Nenkova [30] combine lexical, syntactic and discourse features in a highly predictive model of human judgements of text readability. In this model, several linguistic features correlate best with readability judgments. In particular, the average number of verb phrases per sentence, the number of words in the text, and the vocabulary, among others, are associated with human assessments of how well a text is written. More specific to chatbots, Liu et al. [19] identify some weaknesses of metrics for chatbot responses, and provide recommendations for future chatbot evaluation systems.

Overall, we observe a lack of metrics to evaluate statically and automatically quality aspects of chatbot designs, independently of the chatbot implementation platform. Our goal is to fill this gap.

4 CHATBOT DESIGN METRICS

In order to provide a suite of metrics independent from the chatbot implementation technology, we propose using a neutral design notation to represent chatbots, over which the metrics can be computed. In this section, firstly, Section 4.1 introduces the chatbot design notation, and then, Section 4.2 details our proposed metrics.

4.1 A neutral notation for chatbot designs

Since our aim is to develop metrics for chatbot designs, we need a concrete notation over which to define the metrics. For this purpose, we rely on the chatbot neutral notation we proposed in [28], called Conga. We opt for this neutral notation because, as reported in [28], its definition is based on a thorough revision of 15 widely used chatbot development platforms. This means that the design concepts in Conga can be mapped from and to all these platforms. Hence, by defining the metrics over Conga, they become platform-agnostic as well as significant for many chatbot development platforms. As we will show in Section 5, another practical implication is that one can build importers from different platforms into Conga to perform the measurement of existing chatbots.

Figure 2 depicts the meta-model of Conga. It permits representing a chatbot by a Chatbot object, which contains a set of Intents, Entities, Actions performed by the bot, and conversation Flows. The notation supports multi-language chatbots, and so, each intent can declare a number of TrainingPhrases per definition language. The phrases may refer to Parameters, which are defined at the level of the intent. Parameters are typed either by predefined entities (enumeration PredefinedEntity) or user-defined Entity objects. Entities can be Simple, Regex (regular expressions, a change in this new version of the meta-model) or Composite, and for each language (EntityLanguage), they declare the set of literals and synonyms making up the entity. For example, a chatbot can declare a simple entity for drink sizes with literals small, medium and large in English, and additionally define synonyms regular for medium and big for large.

A chatbot can define one or more Actions of type Text, Image, HttpRequest, HttpResponse and Empty. The two first types are used to compose responses combining text and images. HttpRequest and HttpResponse allow configuring the communication of the chatbot with external services in the backend. The last action type Empty is a wildcard for other platform-specific actions, added in this new version of the meta-model to facilitate transformation between platform-specific definition into Conga (explained in Section 5.2).

Finally, the conversation flow between the chatbot and the users is modelled by Flow objects consisting of user and bot turns (classes BotInteraction and UserInteraction). The user turn refers to the intent to be recognized in the interaction (reference UserInteraction.intent). The bot turn specifies the actions that the bot has to perform (reference BotInteraction.actions).

4.2 A metrics suite for chatbot designs

We propose the suite of metrics for chatbot designs that Table 1 shows. All metrics measure internal attributes of chatbots. We considered three sources when designing the metrics:

- Some of them, like INT (the number of intents) or ENT (the number of user-defined entities), are calculated by taking statistics of concepts from the meta-model in Figure 2. According to [28], these concepts are common in chatbot development frameworks.
- Some other metrics have been adapted from the NLP literature [19, 30] to assess the readability of the chatbot responses or the complexity of the expected user utterances.
- Finally, we use the conversation design principles proposed in [23, 34], and Moore and Arar's classification of chatbots [23], to interpret the value of some metrics such as PATH (the number of conversation paths), FLOW (the number of conversation entry points) and WPOP (the number of words per bot output phrase).

The fourth column of Table 1 classifies the potential impact of the metrics on usability (as defined in the ISO 9241-11) [1] in terms of Effectiveness (i.e., accuracy and completeness with which users achieve their goals), efficiencY (i.e., time and resources that users expend to achieve their goals) and Satisfaction (i.e., comfort and acceptability of use). We also classify metrics based on their target: either global design properties, or specific aspects of intents, entities or conversation flows. Non-global metrics can be computed per element (intent, entity, flow) or averaged for all elements of a kind.

4.2.1 Global metrics. We start introducing *global metrics*. These measure the number of intents (INT), entities (ENT) and flows (FLOW,

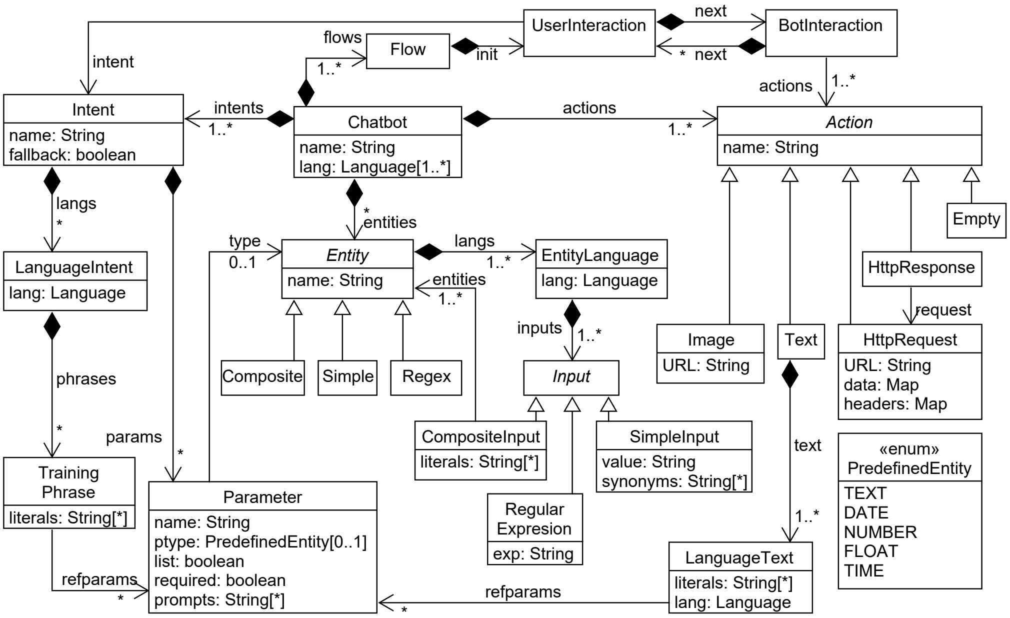


Figure 2: Meta-model for chatbot design (adapted from [28]).

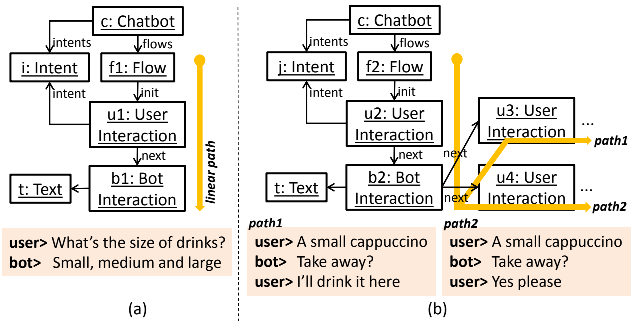
Table 1: Metrics for chatbot designs. Column dimension uses abbreviations for Effectiveness, efficiencY and Satisfaction.

Metric	Description	Type	Dim
Global metrics			
INT	# intents	design size	E
ENT	# user-defined entities	vocabulary size	S
FLOW	# conversation entry points	conversation diversity	E
PATH	# different conversation flow paths	conversation complexity	S,Y
CNF	# confusing phrases [8]	bot understanding	E,S
SNT	# positive, neutral, negative output phrases [33]	user experience	S
Intent metrics			
TPI	# training phrases per intent	topic complexity	E,S
WPTP	# words per training phrase	topic complexity	Y
VPTP	# verbs per training phrase	topic complexity	S,Y
PPTP	# parameters per training phrase	topic complexity	E
WPOP	# words per output phrase	readability	S,Y
VPOP	# verbs per output phrase	readability	S
CPOP	# characters per output phrase	readability	S,Y
READ	reading time of the output phrases [6]	readability	Y
Entity metrics			
LPE	# literals per entity	vocabulary complexity	S
SPL	# synonyms per literal	vocabulary complexity	S
WL	word length	readability	Y,S
Flow metrics			
FACT	# actions per flow	bot response complexity	E,S
FPATH	# conversation flow paths	conversation complexity	S,Y
CL	conversation length	conversation complexity	Y

PATH), and include understanding and user experience metrics (CNF, SNT).

INT is an indicator of design size and functionality, since each intent contributes to functionality offered to the user. The larger INT is, the more functionality the bot offers, potentially impacting effectiveness. ENT measures the size of the chatbot vocabulary and the conversation topic diversity, which may affect satisfaction.

FLOW counts the number of conversation entry points for users, being an indicator of conversation diversity. Since each entry point might correspond to a functionality, FLOW may impact effectiveness. PATH measures conversation complexity. If PATH=FLOW, all conversations are linear, while if PATH>FLOW, some conversation splits into several paths. As an example, Figure 3 shows two small excerpts of chatbot designs conformant to the CONGA meta-model. The chatbot design (a) depicts a linear flow (i.e., FLOW=PATH=1). Linear flows enable simple conversations, typically request/reply, which may indicate a system-centric chatbot [23]. The chatbot design (b) shows a conversation flow that splits after the bot interaction (FLOW=1, PATH=2). This kind of flows permits non-linear conversations with multiple turns and dialogue alternatives, typical



user> What's the size of drinks?
bot> Small, medium and large

user> A small cappuccino
bot> Take away?
user> I'll drink it here

user> A small cappuccino
bot> Take away?
user> Yes please

Figure 3: Chatbot design excerpts illustrating (a) a linear conversation flow, (b) a forked conversation flow.

The combined use of FLOW and PATH can help detecting deviations of some design principles. The *recipient* principle [34] advices to design for the target users, from experts (who may give all information at once) to novices (where the bot needs to prompt for more information). In turn, the *repair* principle [34] recommends supporting clarifications in the conversation, and multiple paths may be an indication of this. Moreover, having several paths per flow potentially results in more natural conversations (impacting satisfaction) but less predictable for the user (likely impacting the user effort or efficiency).

The CNF metric measures the semantic distance between the training phrases of different intents, identifying similar phrases that may confuse the bot to make it identify a wrong intent [8]. Since this may cause errors, the metric is related to effectiveness and satisfaction.

Finally, SNT measures the sentiment of the chatbot output phrases, classifying them into positive, negative and neutral. This is related to satisfaction, since a bot that outputs mostly negative phrases may cause a negative user experience [33].

4.2.2 Intent metrics. Intent metrics measure quality properties of each intent with respect to the expected user utterances and the bot output phrases.

Related to user utterances, TPI counts the number of training phrases in the intent definition. The larger TPI is, the more precise the intent recognition might be, but this also may indicate a complex intent. WPTP measures the length of the training phrases in words. Long phrases are not adequate or even possible in social networks (e.g., Twitter restricts message length), and so, large WPTP values might be problematic. VPTP measures the number of verbs per training phrase. This is an indication of interaction complexity, since composite phrases with several verbs can be more difficult to elaborate for the user [30]. PPTP measures the number of information items (i.e., parameters) the user needs to provide, and the larger PPTP is, the more complex is the intent domain concept.

Regarding chatbot outputs, WPOP measures the number of words per bot output phrase. According to the *minimization* principle [23, 34], the bot answers should be concise. Large phrases are more difficult to understand and can be problematic in social networks. The latter is more concretely targeted by CPOP, as high values may require scrolling (e.g., in mobile devices) and long reading times (with the risk that the user does not complete the reading [23]). Long outputs are especially problematic for voice-based chatbots, since speaking takes longer than reading [23]. Hence, large CPOP values may decrease user satisfaction and efficiency. Similarly, VPOP is another indicator of the complexity of the chatbot responses, given by the number of verbs per output phrase. Finally, READ measures the expected reading time of the bot output responses (a metric related to efficiency). This is calculated as the ratio between the number of words per output phrase, and the number of words that an average person can read per minute [6].

4.2.3 Entity metrics. Entity metrics target user-defined entities representing domain concepts. LPE and SPL are indicators of the complexity of the concepts managed by a chatbot, impacting satisfaction. High LPE and SPL values signal elaborate concepts, but since SPL counts synonyms, a large number may improve recognition in user utterances (better satisfaction). A narrow vocabulary (low SPL) may constrain the way users communicate with the chatbot, and may lead to frustration if the chatbot does not recognize important parameters within user utterances. WL measures the length of words, and similar to CPOP, it contributes to readability and may impact user satisfaction and efficiency.

4.2.4 Flow metrics. Flow metrics consider features of the conversation flows. FACT measures the bot actions (presenting images, text, calling backends) in each conversation flow. The more actions, the more sophisticated tasks can be achieved. Moreover, rich controls help to reduce the user cognitive load and speed up the completion of the intended task. Hence, FACT may impact effectiveness and satisfaction. FPATH measures the number of possible paths per conversation flow. High values signal complex conversations (i.e., more natural-sounding but less predictable). The PATH global metric is calculated by adding up FPATH for each flow. Finally, CL measures the length of each path within a flow, as the number of bot and user turns. This is an indicator of conversation complexity. Longer paths require more time to complete – which affects efficiency – and are typical of conversation-centric chatbots [23].

5 ARCHITECTURE AND TOOL SUPPORT

We have developed a tool called Asymob supporting the automatic measurement of chatbot designs specified with Conga. Next, Section 5.1 presents the architecture of Asymob, including its main features, underlying technologies, and steps required to compute the metrics. Then, Section 5.2 details the conversion of chatbots implemented in two mainstream platforms into Conga.

5.1 Overview of Asymob

We have built a Java framework called Asymob for measuring chatbot designs. The framework is available at https://github.com/ASYM0B/tool. Asymob has a modular architecture that facilitates adding new metrics in multiple programming languages, like Java, Python and Perl. To support chatbots from different platforms, it relies on the neutral chatbot design notation Conga, introduced in Section 4.1. Hence, Asymob computes the metrics on Conga models, independently of any chatbot implementation platform. To measure chatbots from a specific platform, an importer from the platform into Conga must be provided. Currently, Asymob has importers from Dialogflow and Rasa. Section 5.2 will provide more details about these two importers.

To simplify the implementation of new metrics, Asymob supports third-party technologies such as Stanford CoreNLP [20], TensorFlow [2] and Deep Java Learning [11]. Stanford CoreNLP is an NLP library that Asymob uses to perform sentiment and syntactic analysis of the chatbot training and output phrases. The implementation of metrics SNT and VPTP make use of this library. Asymob relies on Deep Java Learning and TensorFlow to detect confusing phrases between intents, using the cosine similarity algorithm. The CNF metric is based on this algorithm.

Figure 4 shows the architecture of Asymob, and the steps to measure a chatbot design. First, the user selects a set of metrics (label 1) and a chatbot (label 2). Then, the Asymob core configures the *metrics database* with the selected metrics, and converts the provided chatbot into a Conga model (label 3, see Section 5.2). Next, the *metric engine* applies the selected metrics to the Conga model, and stores the results in a meta-data file (label 4). On request (label 5), the user can obtain a report with the results in several formats like plain text, Excel and LaTeX (label 6).

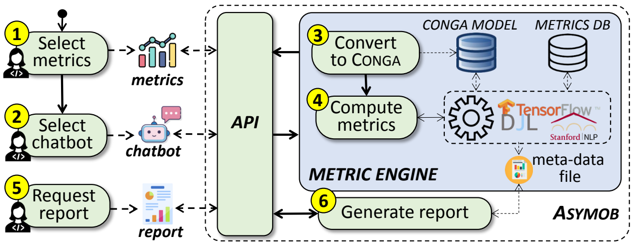


Figure 4: Architecture of ASYMOB.

5.2 Importing chatbots into CONGA

In the following, we provide details of the importers that we have built to convert chatbots from two representative and widely used chatbot platforms (Dialogflow and Rasa) into CONGA.

5.2.1 From Dialogflow to CONGA. Dialogflow is a low-code development platform to create chatbots using a graphical interface within the browser. Chatbots so defined can be exported as JSON files, which our importer is able to convert into CONGA models.

In the JSON-based representation of a Dialogflow chatbot, the file *Agent.json* describes global chatbot features, like its name, definition languages, or connection data to external services (the *webhook*). The latter include details such as the URL, headers, and authentication credentials. Our importer creates a CONGA Chatbot object using the agent name and languages, and an HttpRequest action with the webhook data.

Entities in Dialogflow can be predefined or user-defined. The latter are described either by a regular expression, a list of literals with synonyms, or a composite entity. Each user-defined entity becomes exported as a JSON file containing the entity name and configuration information (if it is a regular expression or a composite entity), and one file per definition language with the corresponding literals. Our importer converts these files into CONGA Entity objects.

Intents in Dialogflow have a name, training phrases, responses, parameters, and an indication of whether they are fallback or enable a webhook, among other features. Intents are exported into JSON files. For each intent definition file, our importer creates a CONGA Intent object with its Parameters and TrainingPhrases, as well as the necessary Actions to compose each response. We currently support text and image responses, and convert other custom responses into Empty actions. Anyhow, this does not affect the defined metrics.

Finally, Dialogflow controls the conversation flow via contexts. These can be input/output to intents, and can store relevant conversation state. Our importer uses the contexts and the responses of the related intents to generate CONGA Flow objects.

5.2.2 From Rasa to CONGA. Rasa is a framework to develop chatbots using Python, markdown and YAML. The definition of a Rasa chatbot comprises several files. The *config.yml* file defines configuration properties, like the chatbot language or the used NL prediction model. The *data/nlu.md* file contains training data to identify the intents correctly, with its entities and synonyms or regular expressions. As an example, the *data/nlu.md* file in Listing 1 defines an intent called order (lines 1–4). The parameters in the training phrases can be defined within brackets and followed by the entity name in parenthesis (e.g., [cappuccino](type)), or with curly brackets (e.g., [medium]{"entity": "size", "value": "medium"}).

```
1  ## intent:order
2  – I'd like a [medium]{"entity": "size", "value": "medium"} [cappuccino](type)
3  – I want a [small]{"entity": "size", "value": "small"} [latte](type)
4  – Can I order a [large]{"entity": "size", "value": "large"} [black](type) coffee?
5  ## synonym:small
6  – little
7  – short
8  ## synonym:medium
9  – regular
10 – median
11 ## synonym:large
12 – big
13 – extra
```

Listing 1: Example of data/nlu.md Rasa file

```
## story1
* order
  – utter_confirm_order
```

Listing 2: Example flow from data/stories.md Rasa file

The listing also declares synonyms for literals small (lines 5–7), medium (8–10) and large (11–13).

The file *domain.yml* defines the chatbot intents, entities and actions. Actions can be text, images, buttons, or custom actions defined in the Python file *actions.py*. Finally, the file *data/stories.md* specifies the conversation flows. Listing 2 shows a flow example, by which matching the intent order triggers the response utter_confirm_order.

We have built an importer that reads the chatbot language from the *config.yml* file and creates CONGA intents and entities from the *data/nlu.md* file, CONGA actions from the *domain.yml* file, and CONGA flows from the *data/stories.md* file. As in the case of Dialogflow, our importer from Rasa supports text and image responses, and converts Rasa custom actions into CONGA empty actions.

6 EVALUATION

We have used ASYMOB to perform an empirical study to assess the suitability of our metrics to detect quality issues and compare bots. We aim at answering the following research questions (RQs):

RQ1 Can the defined metrics detect quality issues in real chatbots?
RQ2 Can the defined metrics be used to compare heterogeneous chatbots?

Next, Section 6.1 describes the experiment setting, Sections 6.2 and 6.3 answer the RQs, and Section 6.4 discusses threats to validity.

6.1 Experiment setting

We have analysed 6 Dialogflow chatbots and 6 Rasa chatbots built by third parties, available at https://github.com/ASYM0B/evaluation. Table 2 shows the metric results, with some extreme values marked in bold. Chatbots are categorised depending on their implementation platform (**D**ialog**f**low or **R**a**s**a) and their source (**G**ithub or **P**redefined natively on the platform). We used ASYMOB to import the chatbots into the CONGA format (cf. Section 5.2) and obtain the metrics.

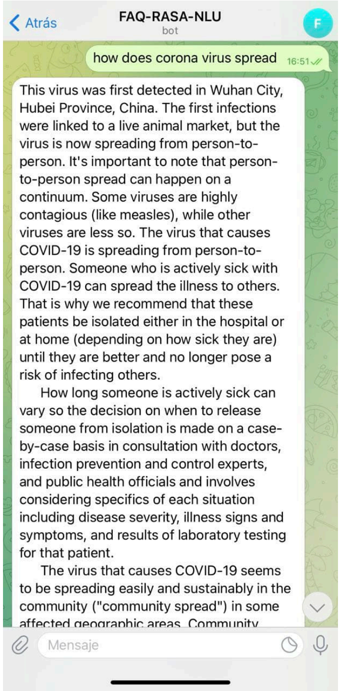
6.2 RQ1: Detection of quality issues

Some metric values reveal design issues. The CPOP of the FAQ-RASA-NLU chatbot is 285 characters. This indicates poor accessibility and

Table 2: Summary of the evaluation. Columns use abbreviations for Dialogflow (DF), Rasa (RS), Github (G) and Predefined (P).

Chatbot			Global metrics						Intent metrics								Entity metrics			Flow metrics		
Name	**Plat.**	**Src**	**INT**	**ENT**	**FLOW**	**PATH**	**CNF**	**SNT (%)**	**TPI**	**WPTP**	**VPTP**	**PPTP**	**WPOP**	**VPOP**	**CPOP**	**READ**	**LPE**	**SPL**	**WL**	**FACT**	**FPATH**	**CL**
bikeShop	DF	G	5	1	4	4	3	38 50 12	2.60	3.48	0.81	0.60	14.00	2.60	55.20	12.00	2.00	5.50	6.64	2.00	1.00	2
googleChallenge	DF	G	32	34	32	32	**1442**	19 59 22	6.94	9.62	1.76	0.94	19.81	3.49	105.16	16.00	3.15	4.54	11.37	1.00	1.00	1
mysteryAnimal	DF	G	62	37	62	62	770	0 0 0	6.52	4.34	1.30	2.27	0.00	0.00	0.00	0.00	163.84	3.89	9.21	3.00	1.00	1
Car	DF	P	77	14	61	117	**4188**	0 0 0	9.70	6.80	1.29	2.25	0.00	0.00	0.00	0.00	14.93	3.60	11.41	1.00	1.92	2
Dining-Out	DF	P	9	15	4	14	148	20 61 19	94.67	3.81	0.76	8.33	8.50	2.56	31.44	7.00	**1255.00**	2.39	11.36	1.25	3.50	3
Easter-Eggs	DF	P	6	0	6	6	0	10 51 **39**	7.17	6.23	1.31	0.00	7.46	1.63	37.54	6.00	0.00	0.00	0.00	1.00	1.00	1
05_event_bot	RS	G	17	0	1	20	167	40 60 0	3.82	2.05	0.12	0.00	15.00	2.75	87.00	12.00	0.00	0.00	0.00	1.02	**20.00**	6
FAQ-RASA-NLU	RS	G	8	0	7	7	4	15 34 **51**	3.38	4.67	1.00	0.00	54.56	3.56	**285.56**	46.00	0.00	0.00	0.00	1.00	1.00	1
small-talk-rasa-stack	RS	G	87	0	86	92	**8467**	24 62 14	22.51	3.82	1.13	0.00	6.66	1.81	28.48	5.00	0.00	0.00	0.00	1.00	1.07	**16**
concertbot	RS	P	6	2	1	1	0	25 75 0	0.00	0.00	0.00	0.00	2.25	0.25	12.25	1.00	0.00	0.00	0.00	3.50	1.00	2
formbot	RS	P	8	1	2	9	37	17 83 0	35.63	4.25	0.92	1.25	5.00	1.50	22.50	4.00	0.00	0.00	0.00	1.16	4.50	7
moodbot	RS	P	6	0	2	4	22	20 80 0	10.50	2.10	0.44	0.00	3.00	1.00	11.25	2.00	0.00	0.00	0.00	1.10	2.00	3

readability, as large answers require scrolling in mobile devices, long reading times (46 seconds for this bot), and cannot be fully displayed on social networks like Twitter due to their message length constraints. As an example, Fig. 5 shows a chatbot response deployed on Telegram using a mobile phone, which requires scrolling as the response has more than 30 lines. This is an example of a content-centric chatbot to access a FAQ. However, according to [23], conversation-centric chatbots with short answers and a natural conversation style are usable in more platforms. The googleChallenge chatbot has the same problem to a lesser extent (CPOP is 105, READ is 16).



how does corona virus spread

This virus was first detected in Wuhan City, Hubei Province, China. The first infections were linked to a live animal market, but the virus is now spreading from person-to-person. It's important to note that person-to-person spread can happen on a continuum. Some viruses are highly contagious (like measles), while other viruses are less so. The virus that causes COVID-19 is spreading from person-to-person. Someone who is actively sick with COVID-19 can spread the illness to others. That is why we recommend that these patients be isolated either in the hospital or at home (depending on how sick they are) until they are better and no longer pose a risk of infecting others.

How long someone is actively sick can vary so the decision on when to release someone from isolation is made on a case-by-case basis in consultation with doctors, infection prevention and control experts, and public health officials and involves considering specifics of each situation including disease severity, illness signs and symptoms, and results of laboratory testing for that patient.

The virus that causes COVID-19 seems to be spreading easily and sustainably in the community ("community spread") in some affected geographic areas. Community

Figure 5: Large response from FAQ-RASA-NLU in a mobile in Telegram.

The flow metrics reveal some complex conversations. The bot 05_event_bot has a single FLOW with 20 paths (PATH and FPATH are 20). Another indicator of conversation complexity is the conversation length CL. The chatbot with the highest CL value (16) is small-talk-rasa-stack.

The sentiment of the bot responses may affect the user experience. In this respect, FAQ-RASA-NLU and Easter-Eggs have 51% and 39% of negative responses (third value of column SNT). Also related to user experience, the high CNF values in bots small-talk-rasa-stack, Car and googleChallenge (8467, 4188 and 1442) may signal chatbot understanding problems due to the existence of similar training phrases in different intents, which may confuse the bots. For example, Car has similar training phrases in different intents, such as *"turn down the heater for each seat in the car"* and *"turn off the heating in my car"*. Other bots with confusing training phrases are small-talk-rasa-stack (*"I am very bored"* / *"I'm bored of you"*), googleChallenge (*"What is the time duration for completing Masters in Artificial Intelligence?"* / *"Completion period for masters in AI?"*), Dining-Out (*"now cafe"* / *"find cafe"*), and bikeShop (*"Can you fix my road bike?"* / *"Can you service my bike?"*). Hence, CNF provides useful information to detect intents that a chatbot may mismatch, without resorting to intensive dynamic testing.

Overall, we can answer RQ1 positively, since our metrics could detect issues regarding readability (CPOP), conversation complexity (FLOW, CL), user experience (SNT) and bot understanding (CNF).

6.3 RQ2: Comparing chatbots

Metrics also serve to compare or classify chatbots based on their design style [23]. For instance, some chatbots like Car, mysteryAnimal and small-talk-rasa-stack are very detailed and complex according to their number of intents (INT), flows (FLOW) and paths (PATH). Instead, others like bikeShop and moodbot are simpler.

Interestingly, two chatbots have no output phrases, one for being a predefined template bot that the developer needs to complete (Car), and the other because a backend API generates the output dynamically (mysteryAnimal). Likewise, chatbots Easter-Eggs, 05_event_bot, small-talk-rasa-stack and FAQ-RASA-NLU lack a domain-specific vocabulary, since ENT is 0. This might be explained as being general-purpose (e.g., for small talk) or simple bots (e.g., 05_event_bot).

Regarding conversations, some chatbots have linear conversations where FLOW=PATH (e.g., FAQ-RASA-NLU, concertbot), while others support complex conversations where FLOW<PATH (e.g., Car, Dining-Out, 05_event_bot). Additionally, the conversation length of some bots is limited to one user-bot interaction (CL=1), and hence, they can be classified as system-centric [23]. Within this set, bots providing long responses (like FAQ-RASA-NLU) are likely content-centric. Other bots allow longer, more elaborate conversations (CL>1). Bots with non-linear conversations (FLOW<PATH) and multiple turns (CL>1) can be classified as conversation-centric [23].

Metrics are also helpful to compare implementation platforms. First, all entity metrics of the analysed Rasa chatbots have value 0. This is so as entities in Rasa are not defined explicitly, but via a Python method that returns whether an entity accepts a given String. The concertbot bot has 0 training phrases because Rasa bots can be trained interactively. We observe that bots in Rasa define fewer entities (ENT) than in Dialogflow. In general, the analysed Dialogflow bots are more detailed in terms of functionality (INT), vocabulary (ENT) and intent recognition (TPI). Conversations in the Dialogflow bots tend to be linear (PATH=FLOW) while in Rasa they split in several paths (PATH>FLOW), denoting less predictability.

Finally, metrics can be used to compare open-source and predefined bots. In Rasa, the predefined bots are simpler than the Github ones, reflected on lower values of INT, FLOW and PATH. This does not happen with the Dialogflow bots.

Overall, we can answer RQ2 affirmatively. Our metrics permit comparing chatbot complexity and size regarding intents, flows and paths; enable classification of chatbots along Moore and Arar's taxonomy [23]; and – being defined over Conga – they can be applied to different chatbot technologies and chatbot sources.

6.4 Threats to validity

Given the limited size of the experiment, we cannot claim differences or similarities between implementation platforms or predefined/open-source bots, for which we would need a larger scale experiment. Instead, our goal was to hint at the usefulness of the defined static chatbot metrics.

Another limitation of our evaluation is that it relies on custom-made importers from existing platforms into Conga. Since Rasa is a framework, it permits programming some aspects of chatbots in different ways. For example, one may train the model on the fly instead of using training phrases, or even change the conversation flow using Python. All these variants may affect the metric values.

7 CONCLUSIONS AND FUTURE WORK

The increasing relevance of chatbots demands support for assessing their quality prior to testing. With this aim, we have proposed a suite of metrics that can be evaluated statically on chatbot designs, independently of their implementation platform. We have demonstrated the feasibility of our proposal by building the Asymob tool, which we have used to evaluate existing heterogeneous chatbots.

In the future, we plan to extend our evaluation to get a panorama of the features of open-source chatbots and derive metric thresholds. Our metrics could be correlated with development metrics like effort, and validated with usability metrics collected dynamically. We plan to extend our tool to cluster chatbots by similarity, and enable semantic clustering by representing chatbots using a bag-of-words model. The latter can be useful to provide a search facility over chatbot repositories. Technically, we aim at embedding Asymob as a web service to let the community profit from its metrics.

ACKNOWLEDGMENTS

Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

REFERENCES

[1] ISO 9241-11. 1998. Ergonomic requirements for office work with visual display terminals (VDTs). Part II guidance on usability. (1998).

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Gordon Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*. USENIX Association, 265–283.

[3] Botium. [n. d.]. https://www.botium.ai/. ([n. d.]). last access in 2021.

[4] J. Bozic and F. Wotawa. 2019. Testing chatbots using metamorphic relations. In *ICTSS (LNCS)*, Vol. 11812. Springer, 41–55.

[5] S. Bravo-Santos, E. Guerra, and J. de Lara. 2020. Testing chatbots with Charm. In *QUATIC (CCIS)*, Vol. 1266. Springer, 426–438.

[6] M. Brysbaert. 2019. How many words do we read per minute? A review and meta-analysis of reading rate. *J. of Memory and Language* 109 (2019), 104047.

[7] J. Cabot, L. Burgueño, R. Clarisó, G. Daniel, J. Perianez-Pascual, and R. Rodríguez-Echeverría. 2021. Testing challenges for NLP-intensive bots. In *BotSE*. IEEE, 31–34.

[8] D. Cer, Y. Yang, S.-Y. Kong, N. Hua, N. Limtiaco, R. St. John, N. Constant, M. Guajardo-Céspedes, S. Yuan, C. Tar, Y.-H. Sung, B. Strope, and R. Kurzweil. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175* (2018), 7.

[9] Chatbottest. [n. d.]. https://chatbottest.com/. ([n. d.]). last access in 2021.

[10] D. Coniam. 2014. The linguistic accuracy of chatbots: usability from an ESL perspective. *Text & Talk* 34, 5 (2014), 545–567.

[11] Deep Java Library. [n. d.]. https://djl.ai/. ([n. d.]). last access in 2021.

[12] Dialogflow. [n. d.]. https://dialogflow.com/. ([n. d.]). last access in 2021.

[13] M. B. dos Santos, A. P. C. C. Furtado, S. C. Nogueira, and D. D. Moreira. 2020. OggyBug: A test automation tool in chatbots. In *SAST*. ACM, 79–87.

[14] N. E. Fenton and S. Lawrence Pfleeger. 1996. *Software metrics - a practical and rigorous approach (2. ed.)*. International Thomson.

[15] FlowXO. [n. d.]. https://flowxo.com/. ([n. d.]). last access in 2021.

[16] J. Jiang and N. Ahuja. 2020. Response quality in human-chatbot collaborative systems. In *SIGIR*. ACM, 1545–1548.

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

[18] Lex. [n. d.]. https://aws.amazon.com/en/lex/. ([n. d.]). last access in 2021.

[19] C.-W. Liu, R. Lowe, I. Serban, M. Noseworthy, L. Charlin, and J. Pineau. 2016. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *EMNLP*. ACL, 2122–2132.

[20] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *ACL: System Demonstrations*. 55–60.

[21] Microsoft Bot Framework. [n. d.]. https://dev.botframework.com/. ([n. d.]). last access in 2021.

[22] S. Möller, R. Englert, K.-P. Engelbrecht, V. V. Hafner, A. Jameson, A. Oulasvirta, A. Raake, and N. Reithinger. 2006. Memo: Towards automatic usability evaluation of spoken dialogue services by user error simulations. In *ICSLP*. ISCA.

[23] R. J. Moore and R. Arar. 2018. Conversational UX Design: An Introduction. In *Studies in Conversational UX Design*. Springer, 1–16.

[24] R. J. Moore and R. Arar. 2019. *Conversational UX Design: A Practitioner's Guide to the Natural Conversation Framework*. ACM, New York, NY, USA.

[25] R. J. Moore, E. Young Liu, S. Mishra, and G.-J. Ren. 2020. Design systems for conversational UX. In *CUI*. ACM, 45:1–45:4.

[26] Pandorabots. [n. d.]. https://home.pandorabots.com/. ([n. d.]). last access in 2021.

[27] D. Peras. 2018. Chatbot evaluation metrics: Review paper. In *ESD*. Varazdin Development and Entrepreneurship Agency, 89–97.

[28] S. Pérez-Soler, E. Guerra, and J. de Lara. 2020. Model-driven chatbot development. In *ER (LNCS)*, Vol. 12400. Springer, 207–222.

[29] S. Pérez-Soler, S. Juárez-Puerta, E. Guerra, and J. de Lara. 2021. Choosing a chatbot development tool. *IEEE Software* 38, 4 (2021), 94–103.

[30] E. Pitler and A. Nenkova. 2008. Revisiting readability: A unified framework for predicting text quality. In *EMNLP*. ACL, 186–195.

[31] N. M. Radziwill and M. C. Benton. 2017. Evaluating quality of chatbots and intelligent conversational agents. (2017). http://arxiv.org/abs/1704.04579

[32] Rasa. [n. d.]. https://rasa.com/. ([n. d.]). last access in 2021.

[33] R. Ren, J. W. Castro, S. T. Acuña, and J. de Lara. 2019. Evaluation techniques for chatbot usability: A systematic mapping study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.

[34] E. A. Schegloff. 2007. *Sequence Organization in Interaction*. Cambridge University Press.

[35] J. Sedoc, D. Ippolito, A. Kirubarajan, J. Thirani, L. Ungar, and C. Callison-Burch. 2019. Chateval: A tool for chatbot evaluation. In *NAACL-HLT (Demonstrations)*. ACL, 60–65.

[36] A. Shevat. 2017. *Designing bots: Creating conversational experiences*. O'Reilly.

[37] M. A. Walker, D. J. Litman, C. A. Kamm, and A. Abella. 1997. PARADISE: A framework for evaluating spoken dialogue agents. In *ACL/EACL*. Morgan Kaufmann Publishers / ACL, 271–280.

[38] Watson. [n. d.]. https://www.ibm.com/cloud/watson-assistant/. ([n. d.]). last access in 2021.

[39] J. Weizenbaum. 1966. ELIZA - A computer program for the study of natural language communication between man and machine. *Commun. ACM* 9, 1 (1966), 36–45.

B.9 Asymob: a platform for measuring and clustering chatbots

Title	**Asymob: a platform for measuring and clustering chatbots**	
Publication	IEEE/ACM 43rd International Conference on Software Engineering: Tool demos (ICSE-DEMOS)	
Core	2021 Core A*	
Authors	José María López-Morales	Universidad Autónoma de Madrid
	Pablo C. Cañizares	Universidad Autónoma de Madrid
	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid
Doi	10.1109/ICSE-Companion55297.2022.9793784	
Date	May 2022	

Abstract

Chatbots have become a popular way to access all sorts of services via natural language. Many platforms and tools have been proposed for their construction, like Google's Dialogflow, Amazon's Lex or Rasa. However, most of them still miss integrated quality assurance methods like metrics. Moreover, there is currently a lack of mechanisms to compare and classify chatbots possibly developed with heterogeneous technologies.

To tackle these issues, we present Asymob, a web platform that enables the measurement of chatbots using a suite of 20 metrics. The tool features a repository supporting chatbots built with different technologies, like Dialogflow and Rasa. Asymob's metrics help in detecting quality issues and serve to compare chatbots across and within technologies. The tool also helps in classifying chatbots along conversation topics or design features by means of two clustering methods: based on the chatbot metrics or on the phrases expected and produced by the chatbot. A video showcasing the tool is available at https://www.youtube.com/watch?v=8lpETkILpv8.

Asymob: a platform for measuring and clustering chatbots

Jose María López-Morales
Universidad Autónoma de Madrid
Madrid, Spain
JoseMaria.LopezM@uam.es

Pablo C. Cañizares
Universidad Autónoma de Madrid
Madrid, Spain
Pablo.Cerro@uam.es

Sara Pérez-Soler
Universidad Autónoma de Madrid
Madrid, Spain
Sara.PerezS@uam.es

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain
Esther.Guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain
Juan.deLara@uam.es

ABSTRACT

Chatbots have become a popular way to access all sorts of services via natural language. Many platforms and tools have been proposed for their construction, like Google's Dialogflow, Amazon's Lex or Rasa. However, most of them still miss integrated quality assurance methods like metrics. Moreover, there is currently a lack of mechanisms to compare and classify chatbots possibly developed with heterogeneous technologies.

To tackle these issues, we present Asymob, a web platform that enables the measurement of chatbots using a suite of 20 metrics. The tool features a repository supporting chatbots built with different technologies, like Dialogflow and Rasa. Asymob's metrics help in detecting quality issues and serve to compare chatbots across and within technologies. The tool also helps in classifying chatbots along conversation topics or design features by means of two clustering methods: based on the chatbot metrics or on the phrases expected and produced by the chatbot. A video showcasing the tool is available at https://www.youtube.com/watch?v=8lpETkILpv8.

CCS CONCEPTS

• **Human-centered computing → Natural language interfaces**; • **General and reference → Metrics**; • **Social and professional topics → Quality assurance**;

KEYWORDS

Chatbot design, metrics, quality assurance

ACM Reference Format:
Jose María López-Morales, Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2022. Asymob: a platform for measuring and clustering chatbots. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516843

1 INTRODUCTION

Chatbots are increasingly used to access all sorts of services, including leisure (e.g., shopping, booking flights or hotels), customer services, professional support (e.g., banking) and information services (e.g., weather) [19]. Their success is due to their natural language conversational interface, which can be used through many channels such as social networks, web apps, or intelligent speakers.

The popularity of chatbots has triggered the emergence of a plethora of platforms, libraries and tools for their construction [14]. Some prominent examples are Google's Dialogflow[1], Amazon's Lex[2], IBM's Watson[3], Rasa[4] or Pandorabots[5], to name a few.

The quality of chatbots is critical for their success. In this respect, some researchers have proposed techniques for testing chatbots [3, 5] and guidelines for their design [11]. However, most tools lack static quality assurance mechanisms that can be used at design time to assess desired chatbot properties. Likewise, there is a lack of tools to compare, cluster and classify chatbots along design features or conversation topics. Such tools would enable a better understanding of the current chatbot landscape, the comparison of chatbots across implementation technologies (e.g., Dialogflow, Rasa) and provenance (e.g., open source repositories, proprietary platforms), and the extraction of valuable data for chatbot analysis.

In order to address these challenges, we present the web platform Asymob for chatbot measurement and clustering. The tool features a repository where chatbots developed using different technologies (currently Dialogflow and Rasa) can be uploaded. It offers a suite of 20 metrics that measure aspects of design size, complexity, and user experience. It also enables the clustering and comparison of chatbots based on these metrics; as well as on conversation topics extracted from the bot expected and issued phrases. The envisioned users of our tool are chatbot designers and developers.

In the rest of the paper, Section 2 introduces the basic notions of chatbots, Section 3 presents the Asymob platform and reports on a preliminary evaluation, Section 4 compares with related work, and Section 5 concludes with a summary and open research lines.

2 AN OVERVIEW OF CHATBOTS

Chatbots offer a conversational interface via natural language to software services. They are typically powered by natural language

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516843

[1] https://dialogflow.com/
[2] https://aws.amazon.com/en/lex/
[3] https://www.ibm.com/cloud/watson-assistant/
[4] https://rasa.com/
[5] https://home.pandorabots.com/

processing (NLP) technologies that provide good under capabilities on sets of predefined topics, called *intents*. In expected conversation topics, which reflect the functional chatbot. Frequently, intents are defined via training phr illustrate the different ways a user may approach the cha example, a chatbot for a pizzeria may have two main int for ordering (expecting phrases like *"A small margherita* and another for obtaining information about the availal types (expecting utterances like *"What pizzas are availab*

Intents may define *parameters*, whose value is extracted user utterances. For example, when ordering a pizza, the us specify the type of pizza (e.g., hawaiian) and the size (e.g., via phrases like *"I'd like a medium hawaiian pizza"*. Parame be tagged as mandatory, in which case, the chatbot wil their value if absent from the user phrase. Parameters are *entities*, which can be either user-defined (e.g., for pizza pre-defined (e.g., for numbers or dates).

Conversations are defined by means of *flows* of expecte and resulting bot *actions*. The latter normally involve a phrase (which may also include parameters), but may als other elements like images or widgets specific to the dep channel (e.g., buttons in the Telegram social network). In the chatbot may need to access an external service to ma intent. For example, in the pizzeria, the chatbot needs to access an information system to store the order.

3 THE ASYMOB PLATFORM

Asymob is a web platform providing static chatbot quality assurance. Next, Section 3.1 describes its architecture, Sections 3.2–3.4 detail its functionality, and Section 3.5 reports on a preliminary evaluation.

3.1 Overview and architecture

Asymob[6] permits *uploading* chatbots of heterogeneous technologies, which then are *measured* using a suite of 20 metrics. Asymob provides *statistics* of the metrics across all chatbots in the repository. In addition, users can *query* the repository to search for chatbots within certain metric bounds and *compare* them against each other according to their metric values. The platform also allows *clustering* chatbots by metric values, or by the conversation topics as given by the words used in training phrases, bot responses and entities.

Fig. 1 shows the architecture of Asymob. Its functionality is offered via a web interface, which interacts with a service layer via a REST API. The *presentation layer* is implemented in HTML and JavaScript, and supports the interactive presentation of metrics and clusters using the libraries Plotly[7] and Cytoscape[8].

The *service layer* (the Asymob core) implements the functionality related to measuring and clustering chatbots. This core has an extensible design, which makes it easy to add new types of metrics, clustering criteria and chatbot technologies. To support the uniform handling of chatbots from heterogeneous technologies, the core relies on a neutral chatbot design notation called Conga [12]. This way, our platform enables the contribution of importers from specific chatbot implementation platforms into

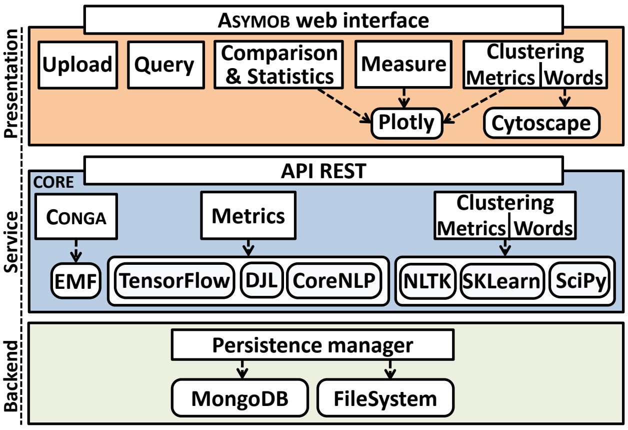


Figure 1: Architecture of Asymob.

Conga, and the measurement and clustering are applied to Conga models. Section 3.2 will provide more details about Conga and the available importers. Then, Section 3.3 will present the current list of supported metrics, built upon established technologies such as TensorFlow[9], CoreNLP[10] and the Deep Java Library (DJL)[11]. Next, Section 3.4 will focus on the chatbot clustering functionality, developed using Python libraries like NLTK[12], SKLearn[13] and SciPy[14].

An additional *backend layer* provides persistence. This stores the uploaded chatbots in the filesystem of the machine where the Asymob core is deployed, and uses mongoDB[15] for storing the data produced in the service layer (i.e., the metric values and the information required for conducting clustering).

3.2 Handling heterogeneous chatbots

Asymob provides static mechanisms to assess chatbot quality. Since there are many chatbot development tools, Asymob implements those mechanisms over a neutral, technology-agnostic chatbot design notation called Conga, and provides importers from different technologies into Conga. This permits reusing the functionality of Asymob with chatbots of heterogeneous technologies.

Conga [12] is designed based on an analysis of 15 popular chatbot tools, so its primitives can be mapped from/to all of them. Conga supports the concepts explained in Section 2 (intents, parameters, entities, conversation flows, bot actions). Intents can be defined via training phrases. User-defined entities can be described as a list of words with synonyms, via a regular expression, or providing a set of strings and other entities. Possible chatbot actions include sending text, images, HTTP requests to external services, or presenting widgets like buttons.

Currently, there are importers from Rasa and Dialogflow chatbots into Conga. Rasa is a framework to develop chatbots using

[6] http://miso.ii.uam.es/asymobService
[7] https://plotly.com/
[8] https://cytoscape.org/
[9] https://www.tensorflow.org/
[10] https://stanfordnlp.github.io/CoreNLP/
[11] https://djl.ai/
[12] https://www.nltk.org/
[13] https://scikit-learn.org/
[14] https://scipy.org/
[15] https://www.mongodb.com/

Table 1: Metrics for chatbot designs.

Metric	Description	Type
Global metrics		
INT	# intents	design size
ENT	# user-defined entities	vocabulary size
FLOW	# conversation entry points	conversation diversity
PATH	# different conversation flow paths	conversation complexity
CNF	# confusing phrases	bot understanding
SNT	# positive, neutral, negative output phrases	user experience
Intent metrics		
TPI	# training phrases per intent	topic complexity
WPTP	# words per training phrase	topic complexity
VPTP	# verbs per training phrase	topic complexity
PPTP	# parameters per training phrase	topic complexity
WPOP	# words per output phrase	readability
VPOP	# verbs per output phrase	readability
CPOP	# characters per output phrase	readability
READ	reading time of the output phrases	readability
Entity metrics		
LPE	# literals per entity	vocabulary complexity
SPL	# synonyms per literal	vocabulary complexity
WL	word length	readability
Flow metrics		
FACT	# actions per flow	bot response complexity
FPATH	# conversation flow paths	conversation complexity
CL	conversation length	conversation complexity

Python, markdown and YAML. Dialogflow is a lowcode development platform to create chatbots using a graphical web interface, and the chatbots can be exported as JSON files.

Overall, users of Asymob can register on the platform or use a generic user. When uploading a chatbot to the platform, the user must specify the chatbot implementation technology (Dialogflow, Rasa or Conga), its visibility (private, so that only the owner can see the chatbot, or public, to allow other users see it), and its version (to enable version control for the chatbot). Then, the proper importer is automatically applied to the chatbot, and both the original chatbot and the resulting Conga model are stored.

3.3 Measuring chatbots

Asymob includes a metrics engine to analyse static characteristics related to the correct design of chatbots. This provides a suite of 20 metrics, which we proposed in [6], covering both *global* design aspects and specific features concerning the design of *intents*, *entities* and conversation *flows*. The metrics are summarized in Table 1 and we explain them next.

- **Global metrics** capture global properties of the chatbot. Specifically, Asymob measures the number of intents (INT), user-defined entities (ENT), conversation entry points (FLOW), conversation flow paths (PATH), confusing phrases (CNF), and output phrases with positive, neutral or negative sentiment (SNT). Confusing phrases refer to similar training phrases (i.e., with small semantic distance) defined by different intents. They are problematic, since a chatbot may confuse them and end up identifying a wrong intent. Additionally, a chatbot that mostly outputs phrases with negative sentiment may impact negatively the user experience. Overall, global metrics are useful to assess the chatbot design size (INT), the chatbot vocabulary size (ENT), the conversation diversity and complexity (FLOW and PATH), and to report potential problems in bot understanding (CNF) and user experience (SNT).
- **Intent metrics** measure quality and complexity aspects of intents, namely, the number of training phrases per intent (TPI), words/verbs/parameters per training phrase (WPTP/VPTP/PPTP), words/verbs/characters per output phrase (WPOP/VPOP/CPOP) and average reading time of the output phrases (READ). Overall, these metrics quantify the complexity and readability of phrases. Large phrases are difficult to understand, are problematic in social networks with constrained message length (like Twitter), and may require scrolling in mobile devices with small screens. For example, high CPOP and READ values entail long reading times, which may make users not to read fully the bot answers. This is even more problematic for voice-based chatbots, since speaking takes longer than reading [11].
- **Entity metrics** analyse the user-defined entities, which represent domain concepts. They are useful to obtain a measurement of their complexity and readability. Entity metrics include the number of literals per entity (LPE), the synonyms per literal (SPL) and the length of words (WL). These are indicators of the complexity of the concepts and the width of the vocabulary of the chatbot.
- **Flow metrics** are concerned with the complexity of the conversation flows and the sophistication of the bot responses. They comprise the number of actions per flow (FACT), the number of conversation paths (FPATH) and the conversation length (CL).

When a chatbot is uploaded, Asymob computes its metrics and displays their value in a table and also in interactive graphs that compare these values with statistics of the chatbots in the repository. Fig. 2 shows the graph for metric INT. The left bar displays statistics of the chatbot repository, and the bar to the right displays the metric value for the uploaded chatbot. We observe that the new chatbot can be considered large, since it has 25 intents, while the average number of intents of the chatbots is around 10 (with a median of 6). The computed metrics are persisted to speed up the generation of statistics when new chatbots are uploaded, and to facilitate the functionalities we explain next.

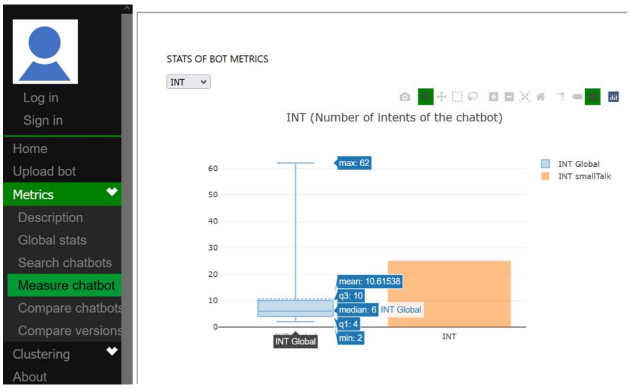


Figure 2: Displaying the value of metric INT.

First, Asymob offers statistics of the metrics of all chatbots in the repository (average, minimum, maximum, median and 1st and 3rd quartiles). They are displayed as a table, as a graph, and side-by-side with the metric values of a specific chatbot, as Fig. 2 shows.

Additionally, Asymob permits comparing a collection of chatbots based on a set of metrics selected by the user. Fig. 3 illustrates this functionality. The x-axis displays the selected metrics (ENT, INT and FLOW in the figure), and the y-axis shows their value for the selected chatbots from the repository (4 bots in this case). We can see that mysteryAnimal stands out in the three metrics, meaning that it has more vocabulary (entities), conversation alternatives (intents) and conversation flows. This comparison can also be performed for several versions of the same chatbot (if different versions were uploaded into the repository) to reason about the evolution between chatbot versions in terms of metrics.

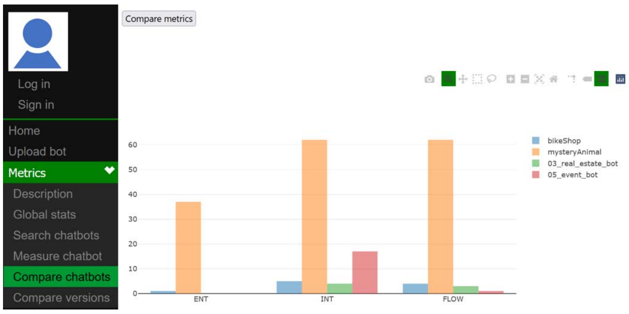


Figure 3: Metric-based comparison of chatbots.

The platform also includes a metric-based chatbot search facility. This permits users to specify the lower and upper limits for the value of some metrics of interest, and Asymob displays the chatbots in the repository with metric values within these boundaries. This is useful to obtain sets of chatbots with certain characteristics. For example, we might be interested in simple chatbots with few intents and no defined entities, or complex chatbots with many intents and complex conversation flows.

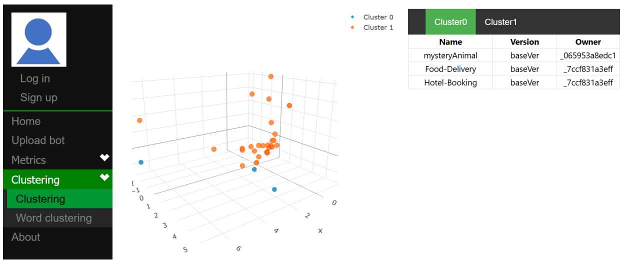
3.4 Clustering chatbots

Asymob supports the automated classification of chatbots based on two disjoint criteria: metric values, or the chatbot vocabulary.

Metrics-based clustering is useful to identify groups of chatbots with (dis)similar design features. For this purpose, the user can select one or more metrics, and the chatbots become classified based on the metric(s) values. For example, clustering by metric INT (i.e., number of intents) would create groups of chatbots with similar size complexity, whereas if the user performs the clustering using metrics FLOW and PATH, then the chatbots would be grouped according to the complexity of their conversations.

Technically, the platform implements the K-means algorithm for clustering the chatbots based on the value of the selected metrics. The user can also select the number of clusters to create (i.e., the k-value), or otherwise, it is automatically computed using the silhouette coefficient [16], as supported by SKLearn.

Asymob visualizes the resulting clusters in a table and graphically, as Fig. 4 shows. The graph can display two or three dimensions, so if the user selects more than three metrics, then the platform reduces the number of dimensions using the principal component analysis (PCA). The graphic represents each chatbot as a dot, and uses a different colour for each cluster of chatbots. In Fig. 4, there are two clusters of 3 and 26 chatbots.



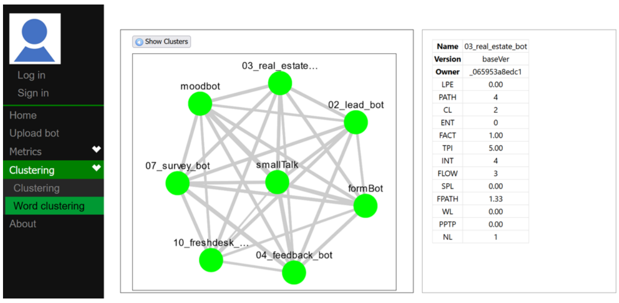
Name	Version	Owner
mysteryAnimal	baseVer	_065953a8edc1
Food-Delivery	baseVer	_7ccf831a3eff
Hotel-Booking	baseVer	_7ccf831a3eff

Figure 4: Metrics-based clustering.

Vocabulary-based clustering classifies chatbots by their vocabulary, which is useful to identify chatbots targeting analogous topics. For example, chatbots for booking flights are likely to be in the same cluster, since their vocabulary tends to be similar. We foresee this clustering to be useful as a way to search for chatbots by similarity to a given one, or by existing topics (represented by clusters). We also envision using this clustering method as a way to present and organize a large set of chatbots within a repository.

For this kind of clustering, Asymob stores all the relevant words that appear in the training phrases, chatbot responses and user-defined entities of each chatbot, along with their frequency of occurrence. Stop words such as prepositions, articles and conjunctions are discarded. Then, the similarity of two bots is given by the cosine-similarity of their *bag-of-words* vectors [10]. Note that each chatbot has to be compared with all the other ones, which becomes time-expensive as the repository grows. To reduce this time, Asymob calculates this similarity as a backend process when a chatbot is uploaded, and caches the result in a database.

In the front-end, users can select a set of chatbots and a similarity threshold for the agglomerative clustering algorithm. The results are shown in a table and an interactive hierarchical graph. The first graph layer has a node per cluster, and clicking on a node shows the chatbots it contains. Fig. 5 shows the chatbots within a cluster. The width of the edges conveys the similarity of two chatbots. Clicking on a chatbot displays its metrics on the right.



Name	03_real_estate_bot
Version	baseVer
Owner	_065953a8edc1
LPE	0.00
PATH	4
CL	2
ENT	0
FACT	1.00
TPI	5.00
INT	4
FLOW	3
SPL	0.00
FPATH	1.33
WL	0.00
PPTP	0.00
NL	1

Figure 5: Vocabulary-based clustering.

3.5 Preliminary evaluation

We have evaluated the cost of uploading a chatbot, which implies its measurement and extracting its bag-of-words for clustering. The latter requires updating a global vocabulary index when the chatbot introduces new words. We found this to require constant time, in the order of 100ms. At this point, a backend process compares and caches the cosine similarity between the bag-of-words vector of the uploaded chatbot and all the rest, which we found to grow linear with the number of chatbots (around 99ms per bot). Being a backend process, it does not affect responsiveness, but we are currently working on its parallelization. In the future, we plan to perform more detailed scalability experiments to detect possible bottlenecks in our architecture and optimize where needed.

4 RELATED WORK

Most approaches to assess chatbot quality rely on testing. Some development platforms (e.g., Dialogflow, Lex, Watson) integrate a web chat console to test chatbots manually. There are also dedicated testing tools like Botium [3] and OggyBug [7] which automate the testing of chatbots built with different technologies. Still, developers need to define concrete test conversation cases. To alleviate this burden, Bottester [20] simulates the user interactions, and other works generate challenging test user utterances automatically [4, 5, 17]. Compared to these works, Asymob provides complementary assessment mechanisms to testing in the form of metrics that are collected statically (i.e., without deploying the chatbot), with reduced effort compared to testing, and which can reveal defects on several quality aspects of the chatbot design.

Additionally, some development platforms (e.g., Dialogflow, Watson, Bot Framework) provide chatbot analytics. This information is collected dynamically when the chatbot is in production, while Asymob targets the design time.

Another popular way to evaluate chatbots is by means of user studies [9, 18]. These typically evaluate user satisfaction and chatbot performance, and require the recruitment and participation of users [15]. Asymob complements these studies with chatbot information that can be gathered automatically and inexpensively.

The support of static means for quality assessment – like those in Asymob – is less frequent. Next, we discuss some exceptions. Dialogflow performs some chatbot validations (e.g., detecting intents with similar training phrases) categorized by severity. Almansor and Hussain [1] use fuzzy logic to detect inappropriate responses based on the sentiment and length of utterances, and Gao et al. [8] use machine learning to predict the popularity of chatbots based on static metrics (e.g., number of intents, conversation flow length). These two works use metrics supported by Asymob, showing that our tool could enable the use of artificial intelligence for prediction.

Finally, our tool takes inspiration from services available in repositories of other artefacts, like meta-models (e.g., MDEForge [2]). However, to the best of our knowledge, Asymob is the first proposal of a chatbot repository featuring metrics and clustering.

5 CONCLUSIONS AND FUTURE WORK

This paper has presented Asymob, the first platform enabling measuring and clustering chatbots. The tool fills a gap on current practice, which is providing automatic means for assessing the quality of chatbots prior to their deployment and dynamic testing. The tool comprises a repository of chatbots, static metrics that can be homogeneously evaluated on heterogeneous technologies, and chatbot clustering facilities based on chatbot metrics and vocabulary.

We are currently building converters from other technologies (e.g., Pandorabots, Lex) into Conga. We are also improving the tool with visualization mechanisms able to capture a large amount of informations, e.g., heatmaps and dendograms for clusters. In the future, we plan to use Asymob to evaluate open source chatbots to get a panorama of their features and derive metric thresholds. We also plan to exploit our clustering techniques to provide search facilities over chatbot repositories. Finally, we would also like to integrate Asymob's services within chatbot development tools like the Conga web IDE [13].

ACKNOWLEDGMENT

This work has been funded by the Spanish Ministry of Science (RTI2018-095255-B-I00, project "MASSIVE") and the R&D programme of Madrid (P2018/TCS-4314, project "FORTE").

REFERENCES

[1] Ebtesam Hussain Almansor and Farookh Khadeer Hussain. 2021. Fuzzy prediction model to measure chatbot quality of service. In *FUZZ-IEEE*. IEEE, 1–4.

[2] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2016. Automated clustering of metamodel repositories. In *CAiSE (LNCS)*, Vol. 9694. 342–358.

[3] Botium. [n. d.]. `https://www.botium.ai/`. last access in 2021.

[4] Josip Bozic and Franz Wotawa. 2019. Testing chatbots using metamorphic relations. In *ICTSS (LNCS)*, Vol. 11812. Springer, 41–55.

[5] Sergio Bravo-Santos, Esther Guerra, and Juan de Lara. 2020. Testing chatbots with Charm. In *QUATIC (CCIS)*, Vol. 1266. Springer, 426–438.

[6] Pablo C. Cañizares, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2022. Automating the measurement of heterogeneous chatbot designs. In *SAC*. ACM, 1–8.

[7] Márcio Braga dos Santos, Ana Paula Carvalho Cavalcanti Furtado, Sidney C. Nogueira, and Diogo Dantas Moreira. 2020. OggyBug: A test automation tool in chatbots. In *SAST*. ACM, 79–87.

[8] Mingkun Gao, Xiaotong Liu, Anbang Xu, and Rama Akkiraju. 2021. Chatbot or Chat-Blocker: Predicting chatbot popularity before deployment. In *DIS*. ACM, 1458–1469.

[9] Jiepu Jiang and Naman Ahuja. 2020. Response quality in human-chatbot collaborative systems. In *SIGIR*. ACM, 1545–1548.

[10] Michael McTear, Zoraida Callejas, and David Griol. 2016. *The Conversational Interface. Talking to Smart Devices.* Springer.

[11] Robert J. Moore and Raphael Arar. 2019. *Conversational UX Design: A Practitioner's Guide to the Natural Conversation Framework.* ACM, New York, NY, USA.

[12] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2020. Model-driven chatbot development. In *ER (LNCS)*, Vol. 12400. Springer, 207–222.

[13] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2021. Creating and migrating chatbots with Conga. In *ICSE Companion*. IEEE, 37–40.

[14] Sara Pérez-Soler, Sandra Juarez-Puerta, Esther Guerra, and Juan de Lara. 2021. Choosing a chatbot development tool. *IEEE Softw.* 38, 4 (2021), 94–103.

[15] Ranci Ren, John W. Castro, Silvia Teresita Acuña, and Juan de Lara. 2019. Evaluation techniques for chatbot usability: A systematic mapping study. *Int. J. Softw. Eng. Knowl. Eng.* 29, 11&12 (2019), 1673–1702.

[16] Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.

[17] Elayne Ruane, Théo Faure, Ross Smith, Dan Bean, Julie Carson-Berndsen, and Anthony Ventresque. 2018. BoTest: A framework to test the quality of conversational agents using divergent input examples. In *IUI Companion*. ACM, 64:1–64:2.

[18] Bayan Abu Shawar and Eric Atwell. 2007. Different measurements metrics to evaluate a chatbot system. In *NAACL-HLT-Dialog*. ACM, 89–96.

[19] Amir Shevat. 2017. *Designing bots: Creating conversational experiences.* O'Reilly.

[20] Marisa Vasconcelos, Heloisa Candello, Claudio S. Pinhanez, and Thiago dos Santos. 2017. Bottester: Testing conversational systems with simulated users. In *IHC*. ACM, 73:1–73:4.

Appendix C

Full Text of Workshop Papers

C.1 Flexible modelling using conversational agents

Title **Flexible modelling using conversational agents**

Publication 5th Flexible MDE Workshop (satellite event of ACM/IEEE MODELS'19)

Authors	Sara Pérez-Soler	Universidad Autónoma de Madrid
	Esther Guerra	Universidad Autónoma de Madrid
	Juan de Lara	Universidad Autónoma de Madrid

Doi 10.1109/MODELS-C.2019.00076

Date September 2019

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 metamodel relaxation to make modelling more flexible, followed by correction steps to make the model conforming to its metamodel. The paper shows how this process is integrated within our conversational modelling framework, and illustrates the approach with an example.

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/

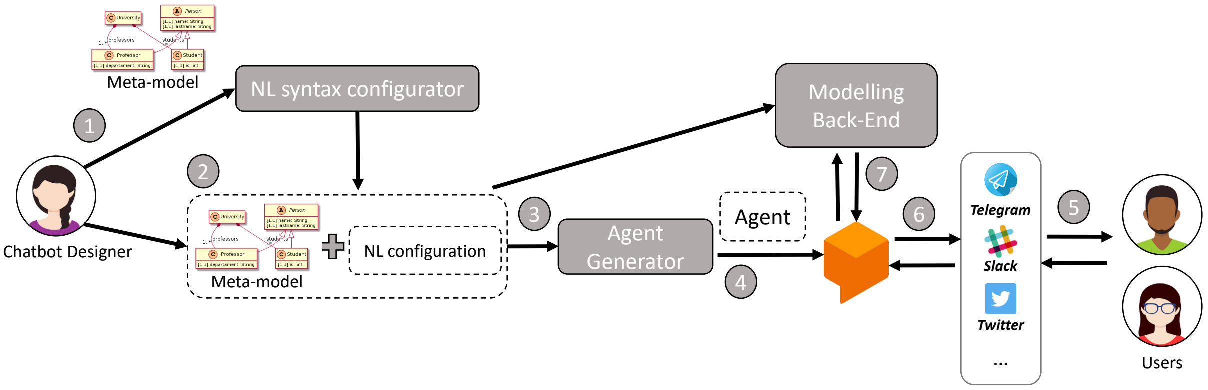
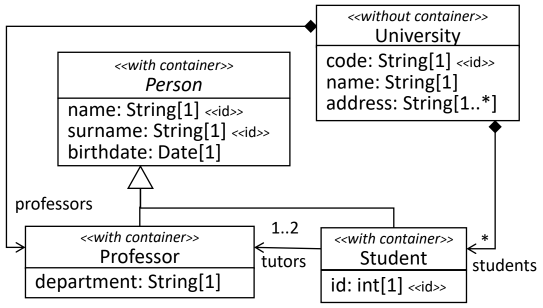
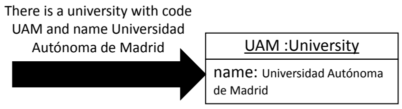


Fig. 1. Creation and use of the modelling agent



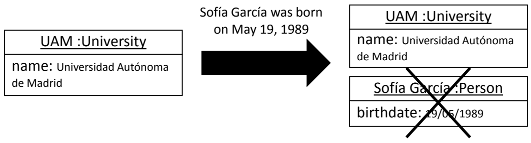


There is a university with code UAM and name Universidad Autónoma de Madrid
UAM :University
name: Universidad Autónoma de Madrid

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).



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. 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. *Studen*t 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

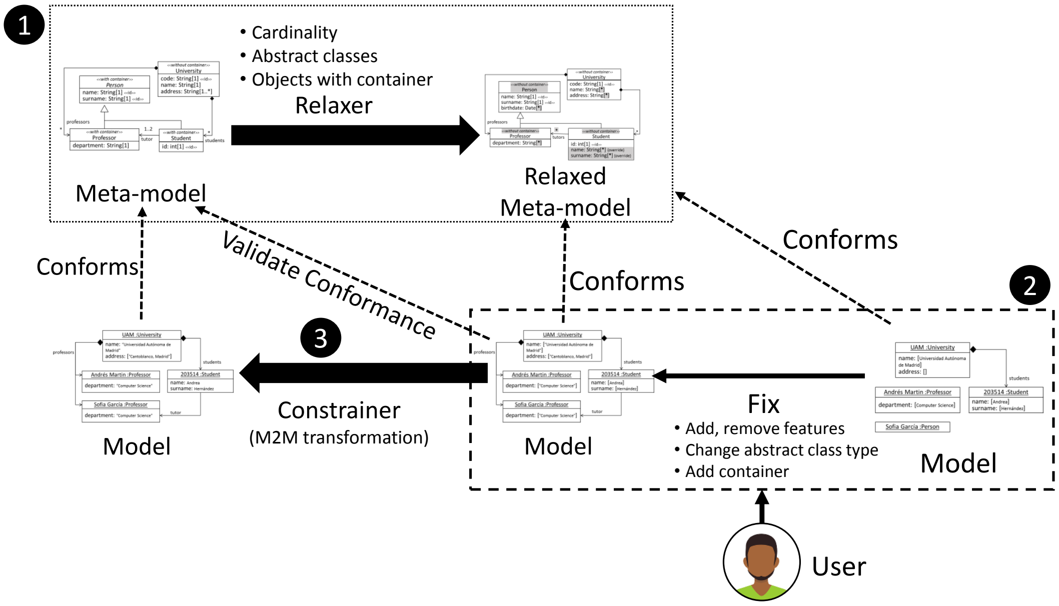


Fig. 5. Steps in flexible modelling. (1) Meta-model relaxation (2) User interaction to create a valid model. (3) Model constrainer

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:

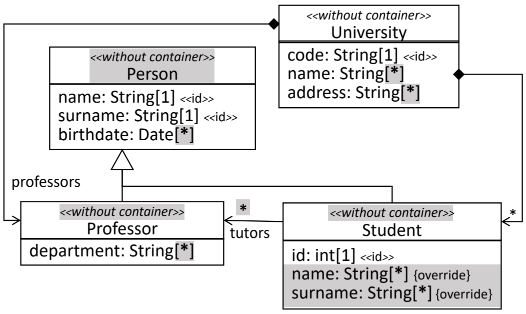
- **Cardinality:** It sets the cardinality of the features that are not identifiers to [0..*]. The identifiers can not change the cardinality because they can not be ambiguous, as users need them to refer to the objects.
- **Abstract classes:** The abstract classes become concrete.
- **With container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

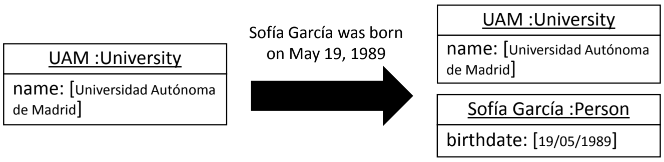
Then, users can build models according to the relaxed meta-model (step 2), so that they can instantiate abstract classes, or assign more values than permitted by the cardinality in a feature. At any moment, the user can validate the model to check its conformance to the original meta-model. The tool 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-





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]

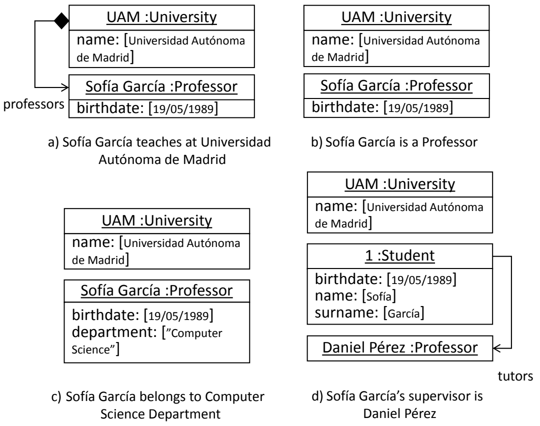
Fig. 7. Example of model creation in NL

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.



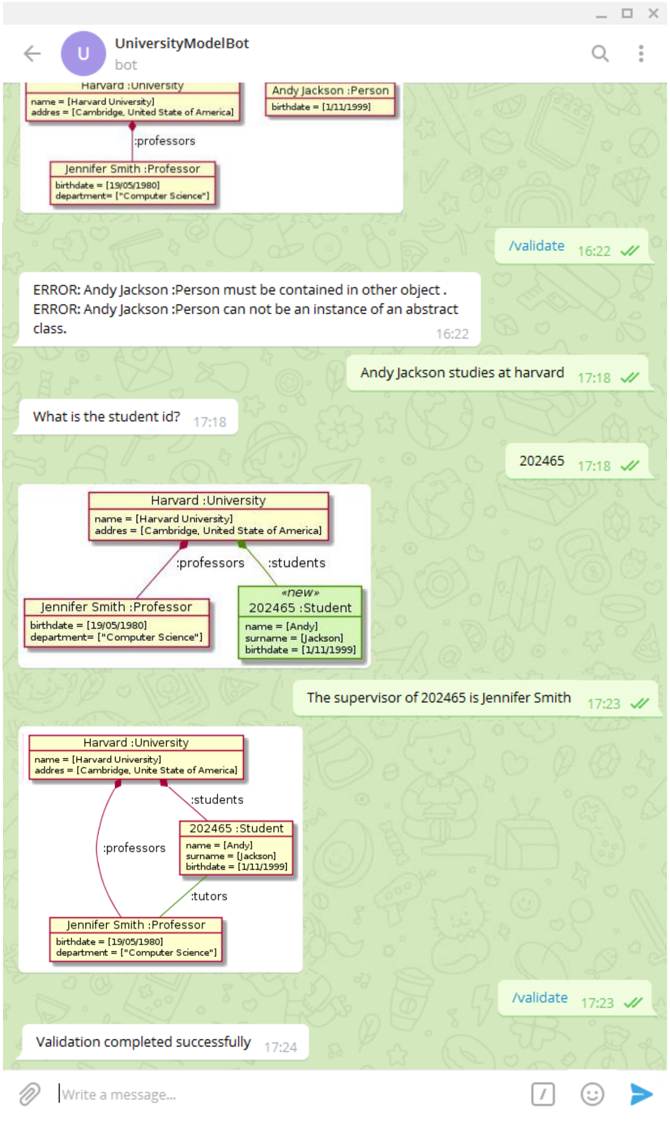
a) Sofía García teaches at Universidad Autónoma de Madrid

b) Sofía García is a Professor

c) Sofía García belongs to Computer Science Department

d) Sofía García's supervisor is Daniel Pérez

Fig. 8. Four examples to type a Person object



/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

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.