Chat2Code: A Chatbot for Model Specification and Code Generation, The Case of Smart Contracts

Ilham Qasse[1]*, Shailesh Mishra[2]*, Björn Þór Jónsson[1], Foutse Khomh[3], Mohammad Hamdaqa[1 3]
[1]Department of Computer Science, Reykjavik University, Reykjavik, Iceland
[2]School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland
[3]Department of Computer and Software Engineering, Polytechnique Montreal, Montreal, Canada
[1]{ilham20,bjorn,mhamdaqa}@ru.is, [2]shailesh.mishra@epfl.ch, [3]{foutse.khomh,mhamdaqa}@polymtl.ca

***Abstract*—The potential of automatic code generation through Model-Driven Engineering (MDE) frameworks has yet to be realized. Beyond their ability to help software professionals write more accurate, reusable code, MDE frameworks could make programming accessible for a new class of domain experts. However, domain experts have been slow to embrace these tools, as they still need to learn how to specify their applications' requirements using the concrete syntax (i.e., textual or graphical) of the *new* and *unified* domain-specific language. Conversational interfaces (chatbots) could smooth the learning process and offer a more interactive way for domain experts to specify their application requirements and generate the desired code. If integrated with MDE frameworks, chatbots may offer domain experts with richer domain vocabulary without sacrificing the power of agnosticism that unified modelling frameworks provide. In this paper, we discuss the challenges of integrating chatbots within MDE frameworks and then examine a specific application: the auto-generation of smart contract code based on conversational syntax. We demonstrate how this can be done and evaluate our approach by conducting a user experience survey to assess the usability and functionality of the chatbot framework. The paper concludes by drawing attention to the potential benefits of leveraging Language Models (LLMs) in this context.**

***Index Terms*—Model-driven Engineering, Automatic Code Generation, Chatbots, Smart Contracts, Blockchain, Natural Language Processing**

I. Introduction

Model-Driven Engineering (MDE) accelerates the development of complex software by raising its abstraction level [1]. One main application for MDE is automatic code generation from system designs [2]. This usage has drawn attention because it produces more accurate, reusable and less error-prone, code that is easier to maintain than manually written code [2]. Over the past two decades, many researchers [3, 4, 5, 6, 7, 8] have successfully used MDE to build frameworks for auto-generating code and applications in domain-specific fields like the Internet of Things, smart contracts, mobile applications, etc. Most MDE-based tools and development frameworks use textual or graphical interfaces as the concrete syntax for the developed languages. This makes it easy for domain experts to specify models, as it uses concepts they already know. However, this isn't as helpful for domain experts, for at least three reasons. First, they need to learn both the interfaces and the underlying syntax of the newly introduced languages [9, 10]. Second, these interfaces provide delayed feedback to the user, if any. This frustrates new non-technical users and makes them quickly abandon the frameworks [9, 10]. Finally, while many of the MDE development frameworks try to bridge the gap between the different software development stakeholders, interaction and collaboration between them are normally limited to providing them with different interfaces, not introducing mechanisms to facilitate active interaction and collaboration[9, 10]. In this paper, we demonstrate how these problems may be circumvented using intent-based chatbots (conversational interfaces) as a concrete syntax for domain-specific languages. This syntax can facilitate the specification of domain models, and auto-generate code from these models.

A chatbot is a software application that uses Natural Language (NL) conversation to conduct conversations (online chat) with users [11, 12]. Chatbots are common information-gathering interfaces. There are several advantages in using them as a concrete syntax or a layer on top of a domain-specific language: (i) the interactive NL interface eases the learning curve, particularly for users with limited knowledge of the exact notations and representations of the concrete syntax of domain-specific languages, compared to other modelling tools (e.g. deployed within eclipse) [9, 10], and (ii) they better suited for collaboration between different stakeholders, since they emulate natural interactions.

To show the feasibility of our approach we implement a chatbot framework, to design and develop smart contract codes from an existing smart contracts reference meta-model. To summarize the most salient contributions of our research, we:

1) investigate the usage of chatbots as a concrete syntax layer for modelling frameworks.
2) propose an approach to integrate chatbots in model-driven engineering frameworks to support generating code from chat conversations.
3) provide a running example from the domain of smart contracts, in which we extended, iContractML, a domain specific language (DSL) to support conversational model specification.
4) evaluate the chatbot based on real-world case studies.
5) conduct user experience study to evaluate the functionality and usability of the chatbot framework.

*These authors contributed equally to this work

6) discuss the impact of advances in LLMs on the future of integrating MDE and chatbots.

The paper is organized as follows: Section II presents a motivation and running example. Section III describes the research methods for Chat2Code. Section IV demonstrates the case study implementation. Section V presents an evaluation of the paper's contributions. Threats to validity are presented in Sections VI. Related work is covered in Section VII. Finally, future research directions and conclusion are presented in Sections VIII and IX respectively.

II. Motivation and Running Example

To highlight the challenges and benefits of utilizing chatbots with MDE to enable code generation from chat conversation, in this paper, we start from a domain-specific modelling language (DSML) and extend it with a conversation-based layer. As a running example, we use a DSML developed for smart contracts called iContractML [5].

iContractML is a unified modelling and development language for smart contracts (i.e., computer code deployed in blockchain to enforce agreements when conditions are met [13, 14, 15]). iContractML supports the "model once and deploy everywhere" approach that enables smart contract creators to deploy their contracts into multiple blockchain platforms. First, the contractual agreement needs to be understood by domain experts with limited coding knowledge, who prefer to use the terminologies they are familiar with. Second, the agreement is normally negotiated between the different contractual parties, and hence it requires an approach that supports capturing the collective intelligence [16].

Figure 1 shows a sample of a smart contract model built using iContractML. Using the iContractML code generator, this model can be transformed into code deployed into multiple blockchain platforms [5]. However, the graphical interface of iContractML is not interactive and provides latent feedback that might be difficult to understand for a new user of the framework. The iContractML users are still required to familiarize themselves with the language syntax and semantics. Furthermore, the framework users are limited to using the terminologies defined in the unified reference model of iContractML. For instance, iContractML defines the members of a smart contract as participants, the user may prefer to use role, struct, or party depending on the user's background. These obstacles might affect the usability of the framework. To enhance the usability of the framework, it is still necessary to make the interaction with it clear and understandable. It should also provide clarity about the working of smart contracts. Chatbots could make this goal attainable by allowing users to use less restrictive language with a wider range of vocabulary and by leveraging a guided interactive conversation with the user. Nevertheless, users may use incomplete or incorrect input in their conversations when using chatbots. Hence, the chatbot should be able to detect and correct the user input when possible, to generate correct model specifications.

Auto-generating code from a chat involves challenging steps, which include chatbot development, integration with the existing meta-model, and natural language processing. In this paper, we will study the feasibility of integrating chatbots in MDE development to generate smart contracts. We will focus on defining the required steps to achieve this goal in the form of a prototype wiring framework.

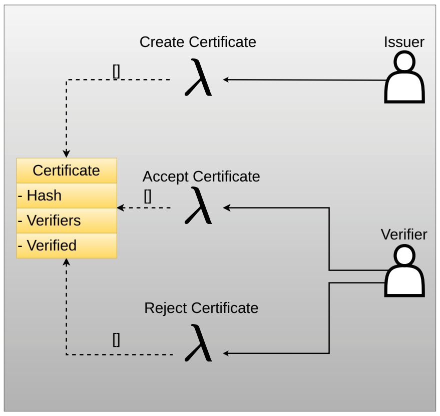

Fig. 1: Example use case modelled using iContractML.

III. Chat2Code Overview

The main goal of this paper is to demonstrate an approach for using chat conversations to generate code. To achieve this, we are following a model-based approach. Given a model-driven development framework that can be used to generate code from models, the goal is to extend it to support conversational model specification. This is a multi-part challenge. The information-gathering flow of the chatbot must conform to the meta-model/abstract syntax of the modelling framework, which is discussed in Section III-A. Then NL input must be transformed through that chatbot to an instance model specification that can be used to generate code as output. We approach the conformance problem by constructing an intermediate layer between the meta-model and chatbot enabling the two to be reciprocally mapped. We solve the NL translation to a model instance using a 5-step transformation process described in Section III-B.

A. *Defining Chatbot Specifications*

The approach for structuring Chat2Code consists of two main steps: constructing an intermediate layer that ensures meta-model conformity, and defining chatbot intent flow to gather instance model specifications. In general, these two steps are done by an expert (technical user) manually, to ensure that the created chatbot does not have errors.

1) Generating Intermediate Layer: We start by extracting the concepts of the language (vocabulary and taxonomy) which are captured by the meta-model/abstract syntax. Based on those concepts, we define a set of synonyms that describe natural language representations of concepts in an intermediate layer, between the abstract and concrete syntax. We also identify natural language synonyms in that layer for attributes from the meta-model and possible operations that can be applied to both concepts and attributes. Then we generate sets of training sentences for each concept and operation that can be used to capture user intent. This intermediate layer serves to link the abstract syntax with the conversational syntax, identifying how the elements are related, and mapping them to each other.

2) Chatbot Intent Flow: Since we are targeting domain experts, we need to direct them to articulate the concepts required to specify the models efficiently. For this reason, the chatbot conversation flow must conform tightly to meta-model concepts. Conversation flow must also capture potential actions commonly used to interact with the meta-model. We have used a statechart, as shown in Figure 2, to represent the intent flow underlying this process. Each concept is represented as a state, with subsequent states representing CRUD operations (create, read, update and delete) that might be required in the user-chatbot conversation. These operations describe user interface conventions that facilitate viewing, searching, and changing the model's information through chat commands. The create operation is to create a model instance from the meta-model, while the operations read, update and delete queries of an existing instance model. In the case of model instance query, as shown in Figure 2, the chatbot makes sure that the concept exists before reading, updating, or deleting a concept. If the concept doesn't exist the chatbot notifies the user. Rules can be enforced in the statechart to ensure that the flow is correct, and over several cycles, a complete model can be extracted from the user conversation. For instance, in creating a model instance the chatbot makes sure that the user has specified all the requirements for the model and notifies the user of the missing requirements. With the specification of this intent flow, and on the basis of entities and training sentences based on the intermediate layer, a chatbot can be implemented to gather information from the user that ultimately maps to the smart contract meta-model through the intermediate layer, allowing for auto-generating smart contract code.

B. Generating Model Specification

This section specifies the process for taking NL input from the user to generate instance model specifications and corresponding code. Figure 3 demonstrates our approach to generating codes from NL input. It involves the five steps described in the following.

1) Input Sanitization: Users frequently provide incomplete or incorrect input when using conversational agents or NL interfaces, especially domain experts. In this paper, we use a simple sanitization component that utilizes edit distance and the list of predefined concepts from the intermediate layer to support autocorrection of user input. Edit distance infers the number of changes that would need to be made to String A so that it becomes equal to String B. We use the Levenshtein Distance algorithm [17] to modify NL input based on similar existing data and confirm those modifications with the user. Handling this kind of deviation and correction in the early stages of our process provides support for users, guiding them to correctly specify the model.

2) NLP Component: The primary role of the NLP component is to detect and identify user intentions from their input. User input consists of noun phrases and verb phrases. The NLP component uses rule-based grammar matching and machine learning matching to identify user expressions from the input. This component includes text input acquisition, text understanding, and knowledge extraction using syntactic and semantic analysis. The syntactic analysis includes defining the structure of the user expression based on grammatical analysis and generating labeled text as nouns, verbs, and adjectives. The semantic analysis rids the labeled text from structural ambiguity, lexical ambiguity, or both. We can compare the resultant ambiguity-free labeled text to predefined training phrases for all intents to identify the required intent. Noun phrases can be divided into regular and proper nouns. Regular nouns can be matched to the predefined intent entities, while the proper nouns can be the names (identifiers) of these entities. Verb phrases define the context and the current state of the intent flow.

3) DSL Model Construction: This step uses the second layer discussed in Section III-A1 to map user input to the model concepts. The output is a DSL (instance model) representing user intention mapped to the meta-model concepts. The DSL follows a modular attribute–value pairs data structure, where it assigns the concepts of the meta-model to the user input for that concept. This enables switching between the defined concepts, and makes it possible to query, update or even delete the defined model more flexibly and accessibly. Moreover, the constructed DSL enables tracing the instance model or generated code to the use case defined in the user conversation.

4) Model Validation: The user intent captured by NLP is validated against a set of rules, which are based on the requirements of the abstract syntax of the meta-model. These form the criteria that user input must satisfy to complete the instance model specification with the minimum required information to run the model transformation and generate code.

5) Code Generation: The code generation component defines the set of rules to transform a DSL instance model into the targeted programming language that conforms to the correct syntax of the target platform. Transformation templates are used to specify the rules to automatically transform the constructed DSL instance model based on model-to-text transformation into code.

IV. Implementation

Having illustrated our general approach to integrating chatbots with MDE frameworks, we turn next to the demonstration of an implementation of this approach using smart contract development as a case study. All the source code for implementing the chatbot for smart contracts (including the transformation template, etc) are provided in the project repository [1].

[1]https://zenodo.org/record/7391855

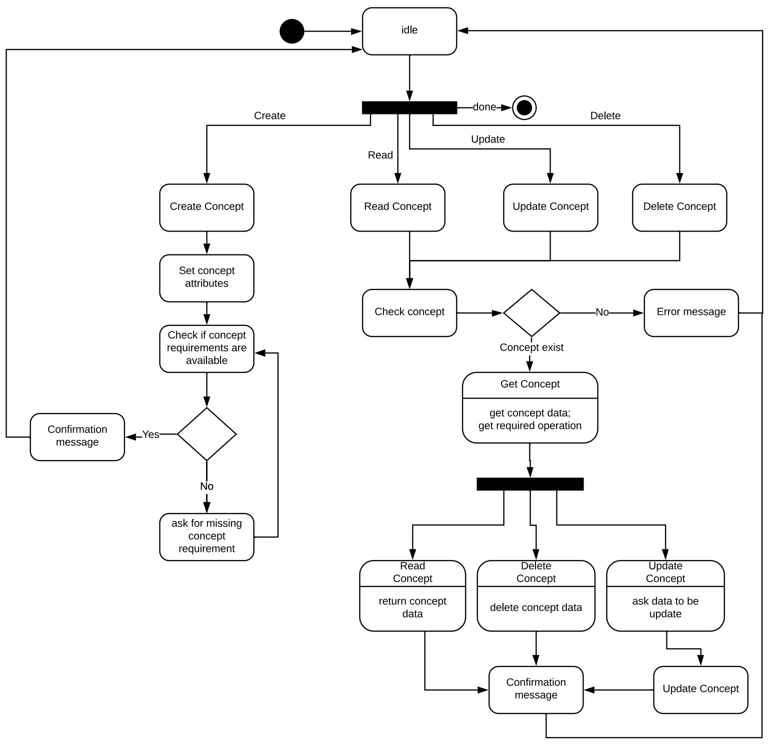

Fig. 2: Chatbot intent flow

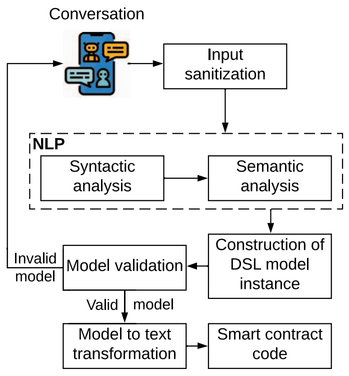

Fig. 3: Generation of model specifications and code from conversational input.

A. Smart Contracts Meta-model

In this paper, we have adopted the meta-model of iContractML [5]. iContractML is a platform-independent reference model for smart contract design and development. The proposed smart contract reference model includes (Contract) a business logic with a name and a target platform. The target platform implies the language in which the smart contract code will be generated. The framework presented here enables a user to generate code in three languages - Solidity, Hyperledger Composer and Microsoft Azure. The contract consists of three *elements*, which are asset, participant, and transaction. Assets are tangible or intangible goods (e.g, money, real estate, or vehicles) stored in the blockchain. Participants are contractual parties with certain access rules (conditions) to execute transactions (actions) to change the state of the assets. The elements in the smart contract are connected via *relationships*. Relationships connect a transaction to either a participant or an asset. Thus, the framework allows the creation of various relationships in transactions.

B. Chatbot Implementation

In this paper, we adopted Xatkit [18] as the bot framework to build the conversational bot and to detect the user input. Xatkit [18] is an open-source framework that helps capture user intent and understand advanced natural language. The framework empowers building platform-independent chatbots [18]. Xatkit is built on Java programming language, making it a suitable fit to integrate with modelling frameworks built on top of the Eclipse modelling framework. The Xatkit framework enables the deployment of the bot on multiple platforms, including web pages and social media platforms. In this paper, we have used the React platform to deploy our chatbot on a web page. On the chatbot web page, we introduce the main components of the smart contract with examples to help users interact with the chatbot. Moreover, within the chat, we explain these components briefly to the user. This is to assist people with limited knowledge of smart contract development to complete their tasks.

During the creation of the chatbot model, errors can emerge at three points: (i) while the chatbot switches between two elements; (ii) while the chatbot detects the actions and intents; and (iii) from the user's end while giving inputs. We address these failure points using several techniques, including input sanitization and validation.

C. *DialogFlow for Intent Detection and Entity Extraction*

In the case of intent detection and entity extraction, Xatkit performs well up to a certain extent, but to enhance the chatbot's performance, DialogFlow [19] has been integrated. By using DialogFlow, the chatbot can understand the numerous variants of the test sentences, which improves entity detection significantly. Dialogflow uses rule-based grammar matching [19] to detect user intent. It compares user input to the predefined training sentences for all intents to identify the best matching intent. Although Dialogflow improves the intent detection module, it cannot detect phrases where the user should provide meaningful sentences. For example, if the user wants to create a contract when the chatbot asks what they like to create, then the input should be *"Create a contract"* or *"I want to create a contract"*, not only *"A contract"*. Thus, a user must elaborate while instructing the chatbot at each instance. If enough information is not provided, then the chatbot may generate incorrect code (for instance if a user wants to create a contract and does not include the word create or any of its synonyms, the chatbot may end up considering an intent to edit an element) or the user will not be able to complete building the desired use case.

D. *Input Sanitization*

While editing, deleting, and reading elements, the user must specify the entity name. If the user makes mistakes during this process, the chatbot will not be able to detect user intent correctly. As mentioned in Section III-B, we used Levenshtein Distance [17] to find the most similar string. Dynamic programming is used to implement the Levenshtein Distance algorithm, which is better optimized than a recursive approach. When a user asks to edit, delete, or read an element, the chatbot obtains the edit distance between the entity extracted and all predefined entities. After finding the closest string, the chatbot modifies the input and confirms the modification with the user. This process is also applied to the data type of the parameters. This ensures that the model instance generated is valid and the chatbot does not break during the code generation process (since we have predefined datatypes in the metamodel, a misspelled datatype would make the model invalid). This extra step makes the chat flow tedious but ensures accurate model editing.

E. *DSL Model Instance Construction*

The extracted sanitized intents are stored as an attribute-value pair intermediate model. Each pair represents concepts from the iContractML meta-model and the corresponding user input from the conversation. Listing 1 shows a snapshot of the DSL model instance structure.

Listing 1: Snapshot of the DSL model instance structure.

```
Contract: 'contract name'
Platform: 'target platform'
Participant {
    Name: 'participant name'
    List: Participant_List_None_0
    Parameter {
        Name: 'parameter name'
        Type: 'parameter type'
        Identifier: 'True or false'
    }
    ....
}
```

A visitor pattern traverses the intermediate model and generates a model instance that conforms to the iContractML meta-model syntax.

1) Model Validation: To ensure the validity of the model instance created by the user, we have implemented a set of validation rules in our framework. Examples of the validation rules are:

- A target platform must be specified.
- A contract name must be specified.
- A user must specify the type of the asset.
- A unique identifier must be specified for an asset and participant.
- The user must specify a valid relationship that relates to an existing object (participant, asset, or transaction).

The chatbot enforces these rules within the conversation with the user, where it explicitly asks the user to specify them. For example, for the rule *"A target platform must be specified"*, the chatbot asks the user for the target platform once the user has completed defining the smart contract use case.

The model validation ensures that the generated DSL model instance has all the required information for code generation.

F. *Code Generation*

To generate the smart contract code from the DSL instance model generated by the chatbot, we used Xtend [20] to write transformation templates for different blockchain platforms. The transformation template takes as input the generated instance model. It applies a set of transformation rules to generate a smart contract code for any of the three target platforms supported. The transformation templates used in this framework are based on the transformation templates of iContractML [5]. We made the design as modular as possible to reduce the chances of errors. This was done by creating a separate executable file for code generation, which was run by the chatbot only at the end of the conversation. This separates the code generation part and the DSL model construction part and ensures that any error in the model is rectified before it is fed to the code generation section. Table I illustrates the transformation template code snippets for generating Ethereum code from the corresponding DSL instance model.

TABLE I: Transformation table template

Concept	Abstract Syntax	DSL Model Instance	Transformation Code Snippet (Ethereum)
Smart Contract / File	SContract: Contract: name=ID Platform: platform=TargetPlatform	Contract: Contract2 Platform: Solidity	contract <<s.name.replaceAll("","_")

Participant	Participant:
	{Participant} 'Participant'
	(Creator : creator? T)?
		Participant Creator: T {
		Name: participant1
		List: None
			for (p: resource.allContents.filter(Participant)
			.toIterable){
			participant = participant + "struct" + .. }

Asset	Asset:
	Asset Type: type=ID;
		Asset Type: Struct{
		Name: asset1
		List: None
			for (a: resource.allContents.filter(Asset)
			.toIterable){
			asset = asset + "struct" + .. }

Transaction	Transaction:
	{Transaction} 'Transaction'
	(Mostrar : mostrar? T)?
		Transaction{
		Name: transaction1
		List: None
			for (tr: resource.allContents.filter(Transaction)
			.toIterable){
			transaction = transaction + "function" + .. }

V. Evaluation

In this section, we evaluate the proposed Chat2Code framework for auto-generating smart contracts. We have conducted a user experience survey to assess the usability of the chatbot.

1) Design: To evaluate the functionality and usability of the proposed chatbot, we conducted an online survey. We provided documented tutorials for the chatbot interface and a video to guide participants in using the chatbot. We also provided a basic overview of smart contracts and related concepts, accessible to non-technical users. We used the survey to answer three research questions:

- RQ 1. What is the overall experience of the participants based on their background?
- RQ 2. What are some current limitations and challenges with the chatbot?
- RQ 3. What are the possible improvements to improve the adaptability of the chatbot?

The survey questionnaire consisted of four parts:

- Demographics: The questions in this part focus on participants' segmentation, related to participants' education, background, gender, age, and work location.
- Pre-test: Based on the participants' segmentation, we followed up with pre-test questions (i) for developers to understand what tools they normally use to develop their applications and how comfortable they are with these tools, and (ii) for non-technical participants, to investigate whether their work involves participation in programming related tasks and the type and level of this involvement.
- Functionality and Usability: In this part, we adopted the Unified Theory on Acceptance and Use of Technology (UTAUT) introduced by Venkatesh et al. [21] as a technology acceptance model. From this model, we have considered six theories of technology adoption that fit with chatbots: (i) performance expectancy, (ii) effort expectancy, (iii) attitude toward using technology, (iv) facilitating conditions, (v) self-efficacy, and (vi) behavioral intention to use the framework. The list of the questions of this part is given in Table II.
- Post-test: In this part, we asked the participants about the overall experience and whether they have any feedback regarding the chatbot.

2) Survey Respondent Recruitment and Statistics: In this paper, we aim to analyze the functionality and usability of the chatbot framework from different users' perspectives. We contacted potential participants in multiple ways. We broadcast our survey in research survey forums to increase our reach. In addition, we sent emails to smart contract developers on GitHub and asked our colleagues in the industry to help broadcast our survey to their friends and colleagues interested in participating in it. In total, we received 46 responses.

Out of the 46 respondents, 2 of the participants are doctoral students, 7 hold a Master's degree, 12 are Master's students, further 15 hold a Bachelor's degree, while 10 participants are still pursuing their Bachelor's degrees. Only 9 participants (19.6%) are familiar with MDE.

To better understand the participants' experience, we divided the survey respondents into different demographic groups: developers with blockchain background, general developers, and non-programmers. Among these 46 participants, 34.78% are developers with a blockchain background, 36.96% are general developers only, and 28.26% are non-programmers. 69.7% of the developers are familiar with 1 to 5 programming languages. On the other hand, 27.3% of the developers know between 5 to 10 programming languages, while 3% know only one programming language. Most developers are familiar with programming languages: python, java, Matlab, and Go. 48.5%

TABLE II: List of questions included in the functionality and usability part.

Metric	ID	Question
Functionality	Q1	Were you able to generate the smart contract given in the test case?
	Q2	How many times did the chatbot respond in the wrong way?
Performance Expectancy	Q3	How useful do you think the chatbot is?
Effort Expectancy	Q4	My interaction with the chatbot is clear and understandable
	Q5	What do you have to say about the amount of information that had to be typed for generating the code?
Attitude toward using technology	Q6	How would you rate the chatbot in terms of the interest instilled by the chatbot?
	Q7	What were the reasons behind the chat being boring?
Facilitating Conditions	Q8	I have the resources necessary to use the chatbot
	Q9	I found the video tutorial helpful
	Q10	How good was the chatbot in giving some clarity about the working of smart contracts?
	Q11	The chatbot is compatible with other platforms I use
Self-efficacy	Q12	How many times did you refer to the tutorial/ video in order to build the chatbot.
Behavioral intention to use the system	Q13	I plan to use the chatbot in the future

of the developers feel comfortable with a new programming language within 2 to 4 weeks, while 42.4% of them get familiar with a new programming language within less than two weeks.

Of the developers with blockchain background, 62.5% possess in-depth knowledge about smart contracts and cryptocurrency. These developers faced many challenges with current smart contract languages, namely a lack of documentation and support (leading to errors in their code), and the completely new syntax that differs based on the underlying blockchain platform.

A. Findings

In this section, we discuss the result of evaluating the functionality and usability of the chatbot. Table III summarizes the survey results for the different metrics.

1) Functionality: In terms of functionality, we asked the participants to follow a predefined case in the tutorial. The chatbot's main functionality is generating smart contract codes from the user's specifications. 78.3% of the participants were able to generate the final code of the smart contract, while the remaining 21.7% were able to generate only the instance model. We also evaluated the number of times the chatbot responded incorrectly or could not detect the user's intention through the conversation. The chatbot responded incorrectly more than once but less than five times for 45.7% of the participants. 28.3% of the respondents countered chatbot errors more than five times but less than ten times. Only a few users (10.9%) faced more than ten chatbot errors, while the chatbot responded incorrectly only once for 15.2% participants through the conversation.

2) Performance Expectancy: This refers to the extent to which participants believe the system is useful. We asked the participants how useful the chatbot was before using it (as an idea) and afterward. The results show that 97.8% of the participants thought that the idea of generating codes/smart contracts from the conversation is useful and helpful (evaluated five and above, where seven is very helpful/useful). Hence, the idea is appealing to a large percentage of users. After using the chatbot, we asked the participants again to evaluate its usefulness. 93.5% of the participants still evaluated the chatbot as useful (evaluated five and above), where only 4.3% of the participants rated the chatbot as not useful.

3) Effort Expectancy: This refers to the effort required from users to use the chatbot. To assess this, we evaluated how clear and understandable the interaction with the chatbot was. The results show that 82.6% of users scored it from 5 to 7, where 7 indicates that the interaction was very clear and understandable. 10.9% of the participants gave it a neutral score of 4. Only three participants (6.5%) indicated that the interaction with the chatbot was not clear. We also asked users about their impression of the amount of data required to type (i.e., conversation length) to generate the code. 47.8% of the users indicated that the amount of typing was acceptable (evaluated 1 or 2, where one is acceptable), while on the other hand, 37% thought there was too much typing (evaluated at 4 or 5, where 5 is too much). 15.2% were neutral and indicated that the amount was not too much/just right.

4) Attitude Toward Using Technology: This is a theoretical measure of participant attitudes using a tool/system. We asked participants to rate the chatbot in terms of how interesting they thought it was to interact with the chatbot. 80.4% rated the chatbot between the scores 1 to 3 (where 1 is very interesting), while 15.2% rated the conversation with the chatbot as boring (the scores are 5 or 6). Issues that affected the participants' attitudes included typing a lot and repetitive messages from the chatbot.

5) Facilitating Condition: This refers to the extent that participants can acquire the required skills and knowledge from the technical infrastructure to use the tool. We measured whether we provided users with enough resources and tutorials to help them use the chatbot. 97.8% of participants indicated that the platform offered the necessary resources. All participants found the video tutorial highly beneficial. However, 69.6% of respondents expressed concerns about the chatbot's compatibility with the Windows operating system, due to lack of support of the current version of the Xatkit framework.

In assessing the impact of the provided resources on participants' knowledge, results show that 67.4% of participants had no prior knowledge of smart contracts before using the chatbot. Among them, 32.6% gained extensive knowledge, while 28.2% gained some clarity regarding the concept through the chatbot. Therefore, a tool that can generate codes from natural language can be a medium of learning and thus can bridge the gaps between various fields.

6) Self-Efficacy: This measures the user's ability to use the chatbot without needing support from a technical person or

TABLE III: Summary of the survey results.

Metric	Question	Results
Functionality	Q1	78.3% generated the code 21.7% only generated the DSL
	Q2	The chatbot responded incorrectly: - once for 15.2% - between one and five times for 45.7% - between five and ten times for 28.3% - more than ten times for 10.9%
Performance Expectancy	Q3	93.5% thinks chatbot is useful
Effort Expectancy	Q4	The interaction with the chatbot: - was clear for 82.6%-was not clear for 6.5%
	Q5	The typing amount: - is acceptable for 47.8% - is too much for 37%
Attitude toward using technology	Q6	80.4% rated the chatbot as interesting 15.2% rated conversation is boring
	Q7	The chatbot is boring due to: - There is a lot of typing - Repetitive messages
Facilitating Conditions	Q8	97.8% indicated that the platform provides the required resources to use the chatbot.
	Q9	All the participants indicated that the video tutorial was helpful
	Q10	69.6% indicated that the chatbot is compatible with other platforms they use
	Q11	32.6% already knew about smart contracts 32.6% gained a lot of knowledge in smart contract 28.2% got some clarity regarding smart contracts
Self-efficacy	Q12	The participant referred to the tutorial: - once for 13% - between one and five times for 34.8% - between five and ten times for 41.3% - more than ten times for 10.9%
Behavioral intention to use the system	Q13	56.52% will likely use the chatbot in the future 30.44% will unlikely use the chatbot

tutorial. 13% of the participants referred to the tutorial only once while using the chatbot. 34.8% required tutorial help from one to five times during the conversation, while 41.3% referred to a chatbot more than 5 times but no more than 10 times. Some participants (10.9%) were unable to interact with the chatbot easily and referred to the provided tutorials more than ten times. One possible reason for the less usage of the tutorial is the video tutorial. This shows that a basic visual introduction (like video) to technology can greatly reduce the requirement for technical support.

7) Behavioral Intention to Use the System: Participants were asked about their potential future usage of the chatbot, based on their experience with the system. Responses were rated on a scale of 1 to 7, with 1 indicating very unlikely and 7 indicating very likely. Results showed that 56.52% of participants expressed a positive likelihood of using the chatbot in the future, while 30.44% were not inclined to use it again. The rest 13.04% were uncertain about using it again. After analyzing the responses, most participants who said they were unlikely to use the system again were non-programmers.

B. Discussion

In this section, we discuss the overall user experience with the chatbot and the factors that affected or improved it. These factors were extracted from participants' comments and suggestions in the survey.

1) What is the overall experience of the participants based on their background?: Figure 4 depicts the overall experience of the participants with the chatbot based on their background. We have presented three metrics - usefulness, likeliness (of using the bot again), and overall experience. For measuring usefulness, we studied the performance expectancy scores, and for likeliness, we considered behavioral intention. In the case of the overall experience, we asked the users to rate the chatbot on a scale of 1-10. The values depicted by the bar graphs are the normalised average score of that metric in a particular category. The normalised score (NS_{ij}) is given by:

$$NS_{ij} = \frac{\textit{Sum of scores of users in category i}}{N_i} \times \frac{1}{Max_j} \quad (1)$$

where $i \in$ [*Non-programmers*, *Programmers without blockchain background*, *Programmers with blockchain background*], $j \in$ [*Usefulness*, *Likeliness*, *Overall experience*], Max_j is the maximum score provided by a user in the metric j and N_i is the number of participants in category i.

The results show that the normalized score of the overall experience is similar for the three different user types. However, for the likeliness metric, it is lower for non-programmers compared to the other two participant types. This is expected as most non-programmer participants did not require programming in their work or daily life. It should be noted that the people of all three categories had similar overall experiences and found the chatbot almost equally useful. Moreover, the usefulness score generally is more than the likeliness and overall experience metrics. The derivations as mentioned above from the figure indicate that such a tool would be helpful for people of all backgrounds.

Furthermore, we asked the participants to identify the challenges that affected their experience. Moreover, we have also obtained the possible improvements from them for future enhancement of the chatbot.

2) What are the current limitations and challenges with the chatbot?: To answer this question, we considered the participants' responses to the open-ended question, "anything else that you would like to comment about your experience with the chatbot". We have coded the participants' answers and classified them into positive, neutral, and negative groups, as shown in Table V. The negative codes mostly represent the limitations and the challenges of the chatbot, which include chatbot errors and usability issues.

Most of the chatbot's errors resulted from user input errors, limitations in detecting the user intention, or technical questions or responses that may have frustrated the user. Some user input errors failed to generate the smart contract codes in some cases, for example, referring to non-existent variables such as participant, asset, or transaction. The chatbot did not notify the users of these errors, adding confusion. The input detection limitations resulted from the sentences' sensitive structure that the NLP engine expected. Moreover, the NLP engine sometimes requires a full sentence, which increases the amount of user effort required to define a use case. This has caused some frustration and negatively affected the user experience.

One of the main limitations many participants pointed out was the typing required to create the smart contract. This increased the likelihood of error and made the chatbot more complicated. Furthermore, minor issues such as the level of detail of the chatbot questions/responses or the long messages from the chatbot decreased the clarity and understandability of the interaction with the chatbot.

3) What are the possible improvements to improve the adaptability of the chatbot?: This question was answered based on the analysis of the participants' suggestions and comments. We have categorized the suggestions into user experience enhancement, interface enrichment, and chatbot functionality improvement.

One important way to improve user experience and decrease input error would be to reduce the amount of typing the system requires. This may include providing template contracts, adding an auto-completion mechanism, and implementing drop-down lists for choosing from a set of constructed sentences. It would also be helpful to provide an intuitive way to track progress as the contract is written so the user remains oriented within the larger process. A side-by-side GUI visualization might make that progress clear. Moreover,

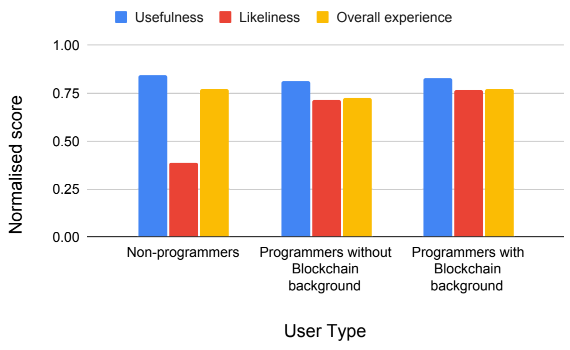

Fig. 4: Metrics to understand expectations and overall experience of the users.

to avoid user input errors, an advanced input sanitization algorithm must be used to detect possible spelling mistakes or detect when the user modifies or uses non-existing variables. Furthermore, improving the NLP algorithm and increasing the corpus of training sentences will help boost the usability of the chatbot. To make the chatbot more accessible to non-technical users, the level of questions/responses of the chatbot should be simpler, and if that is not possible, it should be defined in simple terms as a note.

VI. Threats to Validity

In this paper, we used smart contract development as a running example to automatically generate code from conversational syntax. A threat to the external validity of our research is whether or not our selected running example is general enough for our approach to be extended to other traditional domains of software development such as web development, object-oriented programming languages, Internet of things (IoT), etc. We maintain that smart contract development is complex and there are not many experienced developers in this field [14], which serves to demonstrate the motivation of this work. However, there is still a need to investigate the other software development domains to assess the generalizability of the proposed approach.

We surveyed 46 participants to evaluate the implemented chatbot framework. A threat to validity might have been that we missed potential users who might have assessed our solution differently from those in our study. To reduce this threat, we targeted participants from different backgrounds with different levels of experience (smart contract developers, general developers, and non-programmers). This diversity of backgrounds helped us reflect on real-world conditions of smart contract development. Nevertheless, the number of participants was small, and we need to survey on a larger scale for better feedback, which is the next stage of our project. Furthermore, survey respondents may have provided biased answers based on what we want to hear for several reasons. To help in obtaining unbiased answers, we allowed the respondents to be anonymous, if they so chose.

TABLE IV: Related works comparison

Related Work	Generating Conceptual Model	Query Model	Create Model Instances	Chatbot Development	Input Sanitization	Model Validation	Traceability	Code Generation	Evaluation Method
[9]	✓		✓	✓					
[10]	✓								
[22]				✓					
[23]		✓							
[23]				✓					
Our approach		✓	✓	✓	✓	✓	✓	✓	✓

TABLE V: Participants' responses codes statistics for RQ2.

Participants' responses codes		# of participants
Positive	great experience	8
	gained knowledge	4
Neutral/ no comments	-	17
Negative	chatbot's errors	9
	usability issues	8

VII. Related Work

This section presents the related work of using chatbot with MDE whether it was to generate conceptual models, query models, or developing chatbots. Furthermore, we present the related work of auto-generating smart contract artifacts from chatbots or NLP. In general, there are few works done in the field of chatbot and MDE [10, 22, 23, 24]. Table IV summarizes the related work and compares it with our approach. Most of the discussed related work focuses on generating, querying models, or developing chatbots, while in our approach, we focused on querying and creating instances from models. Furthermore, we applied our approach to auto-generating codes from chatbots, in particular smart contract artifacts. There is also related research work in the field of using NLP to domain modelling such as in [25, 26, 27]. However, these works focus on analysing text (description) instead of interactive conversation as in our case.

VIII. Future Research Directions

Enhancing NLP algorithms is a vital aspect of improving chatbot technology. By improving NLP, chatbots can generate more accurate code from user conversations. This can be achieved by utilizing more sophisticated machine learning algorithms and incorporating additional sources of information such as knowledge graphs. One promising area of research that shows potential for improving NLP is using large language models (LLMs). These models can significantly enhance the effectiveness and efficiency of chatbots integrated with MDE frameworks. They improve understanding of user queries and commands, leading to a more engaging conversational experience. Additionally, LLMs continuously learn from user interactions, allowing the chatbot to enhance responses and adapt to user preferences over time. As the models receive more training data, their language generation capabilities become more precise and context-aware, resulting in highly accurate and contextually appropriate responses. Therefore, using LLMs can be a promising direction for improving NLP algorithms and ultimately enhancing the performance of chatbots in generating code.

IX. Conclusion

In this paper, we investigated the use of chatbots as an interactive alternative approach to concrete syntax for modelling frameworks to enable code generation from chat conversations. We proposed an approach for auto-generating platform-independent codes from conversational syntax. We showcased our methodology with smart contract development and implemented a chatbot framework, for modelling and developing smart contracts. Furthermore, we evaluated the chatbot framework in terms of usability, and functionality based on a user experience study. The results show that the overall user satisfaction depends on the participant's background. However, 79% of the participants in all groups had an above-average overall experience in using the framework.

This work is considered an initial step towards making coding accessible to a wider community of domain experts by enabling code generation from chat conversations. An approach that can help us reap the benefits of unified modelling without limiting our ability of expression. The current approach and implementation require further improvements to achieve it's ultimate goal. In future work, we plan to extend the approach to other case studies in software development such as mobile development, IoT, and other domain specific and product line development scenarios.

Acknowledgment

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), [funding reference no. RGPIN-2023-05484] and the Icelandic Centre for Research (Rannís),[funding reference no. 218202-051].

References

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[2] Iftikhar Azim Niaz. Automatic code generation from uml class and statechart diagrams. *Graduate School of Systems and Information Engineering., University of Tsukuba, Ph. D. Thesis*, 2005.

[3] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM (Dissertation/Demos/Industry)*, pages 56–60, 2018.

[4] Juha-Pekka Tolvanen and Steven Kelly. Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 711–719. IEEE, 2016.

[5] Mohammad Hamdaqa, Lucas Alberto Pineda Metz, and Ilham Qasse. icontractml: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In *Proceedings of the 12th System Analysis and Modelling Conference*, pages 34–43, 2020.

[6] Mohammadali Gharaat, Mohammadreza Sharbaf, Bahman Zamani, and Abdelwahab Hamou-Lhadj. Alba: a model-driven framework for the automatic generation of android location-based apps. *Automated Software Engineering*, 28(1):1–45, 2021.

[7] Xuan Thang Nguyen, Huu Tam Tran, Harun Baraki, and Kurt Geihs. Frasad: A framework for model-driven iot application development. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 387–392. IEEE, 2015.

[8] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Romina Spalazzese. Model-driven engineering for mission-critical iot systems. *IEEE software*, 34(1):46–53, 2017.

[9] Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, and Juan de Lara. Towards conversational syntax for domain-specific languages using chatbots. *J. Object Technol.*, 18(2):5–1, 2019.

[10] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Flexible modelling using conversational agents. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 478–482. IEEE, 2019.

[11] Jack Cahn. Chatbot: Architecture, design, & development. *University of Pennsylvania School of Engineering and Applied Science Department of Computer and Information Science*, 2017.

[12] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. Challenges in chatbot development: A study of stack overflow posts. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 174–185, 2020.

[13] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[14] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 2019.

[15] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

[16] Shweta Suran, Vishwajeet Pattanaik, and Dirk Draheim. Frameworks for collective intelligence: A systematic literature review. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[17] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

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

[19] Navin Sabharwal and Amit Agrawal. Introduction to google dialogflow. In *Cognitive virtual assistants using Google Dialogflow*, pages 13–54. Springer, 2020.

[20] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[21] Viswanath Venkatesh, Michael G Morris, Gordon B Davis, and Fred D Davis. User acceptance of information technology: Toward a unified view. *MIS quarterly*, pages 425–478, 2003.

[22] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. Model-driven chatbot development. In *International Conference on Conceptual Modeling*, pages 207–222. Springer, 2020.

[23] Sara Pérez-Soler, Gwendal Daniel, Jordi Cabot, Esther Guerra, and Juan de Lara. Towards automating the synthesis of chatbots for conversational model query. In *Enterprise, Business-Process and Information Systems Modeling*, pages 257–265. Springer, 2020.

[24] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, Gwendal Daniel, and Jordi Cabot. A model-based chatbot generation approach to converse with open data sources. *arXiv preprint arXiv:2007.10503*, 2020.

[25] Rijul Saini, Gunter Mussbacher, Jin LC Guo, and Jörg Kienzle. Domobot: a bot for automated and interactive domain modelling. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10, 2020.

[26] Mohd Ibrahim and Rodina Ahmad. Class diagram extraction from textual requirements using natural language processing (nlp) techniques. In *2010 Second International Conference on Computer Research and Development*, pages 200–204. IEEE, 2010.

[27] Marcel Robeer, Garm Lucassen, Jan Martijn EM Van Der Werf, Fabiano Dalpiaz, and Sjaak Brinkkemper. Automated extraction of conceptual models from user stories via nlp. In *2016 IEEE 24th international requirements engineering conference (RE)*, pages 196–205. IEEE, 2016.