2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) | 978-1-6654-2484-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/MODELS-C53483.2021.00061

From Prose to Prototype: Synthesising Executable UML Models from Natural Language

Guus J. Ramackers
Leiden Institute of Advanced Computer Science
Leiden University
Leiden, The Netherlands
g.j.ramackers@liacs.leidenuniv.nl

Pepijn P. Griffioen
Leiden Institute of Advanced Computer Science
Leiden University
Leiden, The Netherlands
p.p.griffioen@umail.leidenuniv.nl

Martijn B.J. Schouten
Leiden Institute of Advanced Computer Science
Leiden University
Leiden, The Netherlands
m.b.j.schouten@umail.leidenuniv.nl

Michel R.V. Chaudron
Department of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
m.r.v.chaudron@tue.nl

***Abstract*—This paper presents a vision for a development tool that provides automated support for synthesising UML models from requirements text expressed in natural language. This approach aims to simplify the process of analysis - i.e. moving from written (and spoken) descriptions of the functionality of a system and a domain to an executable specification of that system. The contribution focuses on the AI techniques used to transform natural language into structural and dynamic UML models. Moreover, we envision a 'human-in-the-loop' approach where an interactive conversational component is used based on machine learning of the system under construction and corpora of external natural language texts and UML models. To illustrate the approach, we present a tool prototype. As a scoping, this approach targets data-intensive systems rather than control-intensive (embedded) systems.**

***Index Terms*—UML, MDA, requirement text, natural language processing, model driven engineering, executable specification, transformer architecture**

I. Introduction

The digital society requires organisations to automate increasingly complex functionality in highly interlinked software systems. Such digital products and services can no longer be static but need continuous evolution as new opportunities and challenges arise.

Technologies such as Model Driven Architecture (MDA) coupled with the Unified Modeling Language (UML), Domain Specific Languages (DSLs), or Model Driven Engineering (MDE) provide ways to express specifications that can be transformed into executables. However, while these technologies are accessible to IT experts, they are difficult to understand by domain experts and end-users. As a result, these approaches do not enable rapid feedback and communication between stakeholders, resulting in systems that do not fully satisfy business requirements and are difficult to modify.

To engage domain experts and end-users effectively in the communication process around requirements specification and logical design, they need to be addressed in the languages they understand. Firstly, system requirements are foremost expressed by domain experts in natural language documents [16]. Secondly, evaluating whether a specification is indeed 'correct' and 'complete' is done by viewing or using the result in the form of a prototype of the system under development [28].

The Prose to Prototype (P2P) project addresses these issues by investigating how textual requirements documents can become first-class citizens in designing digital products and services. It aims to process them with state-of-the-art natural language processing (NLP) techniques to map them to UML models. In particular, we aim to synthesise models that can be executed as a prototype immediately.

Due to the inherent ambiguity of natural language, previous efforts in this area have typically restricted the input text of such tools to overly structured language - in effect, reverse translated versions of UML models [27] [10]. Such textual expressions of requirements do not effectively enable the stakeholder communication process as they do not reflect real-world requirements documents. We aim to address this issue by deploying several novel NLP and AI approaches to augment traditional syntactic Part-of-Speech (POS) tagging techniques.

In our approach, we utilise recent advances in pre-trained unsupervised language models. The usage of these models brings us closer to addressing the ambiguity issue of natural language. However, they can never eliminate this problem conclusively. Therefore, we anticipate the need for an interactive human-in-the-loop approach to further reduce ambiguity. Such an approach is not solely needed for ambiguity reduction.

Another challenging aspect of requirements texts is that they are nearly always incomplete, particularly in the initial phases of the requirements elicitation process. A human-in-the-loop component needs to address not only ambiguity but also incompleteness. It will do so by having a conversation based on the full context of the evolving model by asking the modeller user to specify the solution to multiple available

978-1-6654-2484-4/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS-C53483.2021.00061

options. In addition, it will suggest new extensions to the model based on data mining of existing corpora - whether in textual or in (UML) model format. The resulting final requirements texts and models will be fed back into a corpus to create a domain-specific 'self-learning' element to the tool environment.

The approach outlined above will enable natural language usage to play a more effective role in mapping requirements text to formal UML-based system specifications. These should be obtained faster than traditional MDE development processes and tools, and they should be of higher quality in terms of completeness and ambiguity reduction. However, they would still lack an effective evaluation mechanism that involves domain experts and end-users, who are rarely UML experts.

Therefore, we have created an initial implementation of an integrated tool component that can interpret the generated UML model and present it as a runnable prototype. The user can execute the specification in order to validate it. They can enter 'live' data rather than look at static 'wireframes'. The end-user often discovers missing properties, relationships, operations, or larger missing elements, such as classes, by entering data. The user can try out any modifications (with the help of interactive tool functionality). These are reflected in the UML model to enable immediate execution of the evolved specification in the prototype.

This paper is structured as follows. Section II provides relevant background on natural language processing and transformation approaches from text to UML. It also describes the background of our interactive tool components. Section III describes the vision of the P2P project and presents its full architecture. Section IV discusses the status of the architecture components currently under development. Section V elaborates on the future directions of the project in terms of components and issues to be addressed, and Section VI summarises the unique aspects of our approach.

II. Background

There are a number of existing approaches to transforming textual requirements into UML models. Several NLP tools support these transformations in different aspects of the process, and a few of them include limited interaction with the user to improve the outcome. The following section reviews existing literature in this area.

A. Natural language processing

NLP enables an intuitive mode of interaction between humans and software development tools. It has received significant attention from industry and research in the last decade. NLP has gained further traction in the last five years as a result of the introduction of 'transformer' models, which have improved the effectiveness of several NLP tasks.

To create a new representation of an existing natural language text, referred to as sequence to sequence modelling, Recurrent Neural Networks (RNN) based architectures have traditionally achieved the best results. However, they cannot process information that is longer than a sequence (or across compound sentences), and therefore cannot model the context of occurrences in sentences [15]. To address this issue, attention mechanisms were introduced to highlight or downweigh elements of input sequences based on a calculated attention score. This allows models that use these mechanisms to only focus on the important segments in sequences, while discarding irrelevant parts.

Vaswani et al. pioneered a novel architecture based solely on these attention mechanisms: the transformer architecture [30]. This architecture makes use of an encoder-decoder structure that maps the input sequence from symbols to continuous representations, and subsequently generates an output sequence of symbols from the continuous representations. Both the encoder part and the decoder part make use of attention mechanisms.

Various models using the transformer architecture have surpassed other neural models in performance for natural language understanding (NLU) and natural language generation (NLG) tasks [32]. Most notably, the Bidirectional Encoder Representations from Transformers (BERT) model has seen significant success in modelling and capturing contextual clues in texts by taking both preceding and subsequent contexts into account [29].

B. The transformation from Text to UML Models

Requirements are an essential aspect of software development; they enclose the functionalities and prerequisites of the software being developed. Depending on the type of software, different analysis models for requirements are used. Requirements typically start in a textual format, and are subsequently transformed into analysis models. This transformation process can be done manually, but it is very time-intensive. Therefore, several approaches have been proposed to automate the text to model transformation process.

An overview of different transformation approaches between requirements and analysis models can be found in the paper of Yue et al., in the form of a systematic review [33]. The approaches differ in terms of input, output, transformation steps, intermediate model, automation, and intervention.

Overmyer et al. developed the Linguistic assistant for Domain Analysis (LIDA) that helps modellers identify objects, and their attributes and methods, using POS tagging. Refining the identified objects, attributes and methods, and creating associations between them, still has to be done manually by the modeller [22].

In contrast, Perez-Gonzalez and Kalita developed Graphic Object-Oriented Analysis Laboratory (GOOAL), which first regulates the input text before constructing a model from requirements descriptions. However, this tool is limited to process simple problem domains, and requirements texts need to be structured before feeding them into the system, limiting its usefulness in practice [23].

The Class Model Builder (CM-Builder) proposed by Harmain and Gaizauskas also has this problem: even though the tool uses a more sophisticated pipeline of NLP techniques,

such as tokenisation, POS tagging and parsing, it is still dependent on hard-coded sentence patterns to indicate aggregation relations [13].

Building on previous implementations, More and Phalnikar utilize domain ontology techniques combined with NLP techniques to extract classes in their desktop tool called Requirements Analysis to Provide Instant Diagrams (RAPID). Still, the issue remains that the input text is limited to a certain set of rules before being processed [19].

Following the road of extracting classes and attributes using predefined patterns, Azzouz et al. use over 1000 patterns in their tool called the Automatic Builder of Class Diagram (ABCD) [5].

In 2016, Narawita and Vidanage developed UML Generator, which extends a rule-based approach with a Weka model to recognize relationship types and multiplicity, pioneering the use of machine learning in the process of identifying parts of diagrams [20].

aToucan is an NLP tool proposed in the research of Yue et al. It can automatically transform use case models to UML analysis models such as class, sequence and activity diagrams. The use cases are modelled following the Restricted Use Case Modelling approach, which is designed for this automation purpose. The models are formalised with the Stanford Parser, and, by using a set of rules, transformed into a UML analysis model [34]. The approach is limited by the input models, which have to follow the modelling approach of Yue et al.

Sharma et al. present a solution to automatically generate UML activity and sequence diagrams from unstructured requirement descriptions. Their approach is based on grammatical, lexical and syntactic analysis of the descriptions using the Stanford Parser and Tagger. Based on the analysis, a set of frames are created, which are used to generate a sequence diagram, or an activity diagram [26]. The solution can handle unstructured texts, but the text must have certain keywords to trigger the frames used for the model generation. As a result, the input is limited to texts containing these keywords.

Arora et al. [2] present a rule-based approach for generating (domain level) UML class models from requirements documents in natural language. These rules are extracted from patterns in syntactic analysis. They find that many associations found by the rules are irrelevant, and suggest that future approaches should use a human-in-the-loop approach.

Osman has made initial steps towards automated abstraction and summarisation of designs [21].

Finally, Tang has developed an approach to generate UML class diagrams from requirements text using NLP toolkits such as NLTK and Stanford CoreNLP. This is the first tool to generate UML from requirements texts with extensive support for interactivity with the modeller user to refine and finalise the generated diagrams. Still, the problem of variety in real-world requirements texts remains: the tool still uses a list of keywords to link attributes and classes [27].

To conclude, previous efforts to assist UML modelling using NLP techniques have led to somewhat limited results, specifically due to requirements that the input text is overly structured and uses specific keywords. As such, they are not reflective of real-world requirements documents. Recent developments in NLP, specifically the introduction of transformer architectures, offer opportunities to improve both the robustness and the effectiveness of NLP systems in dealing with a wider variety of writing styles and formulation in requirements texts.

C. Combining Batch- and Interactive-Processes

Several approaches exist where a development tool acts as an interactive aid to the developer by recommending the next element to be added to a model. These approaches take (some) inspiration from source code-completion systems. As such, these human-in-the-loop approaches inherit the one-element-at-a-time mechanism for constructing a model from these systems.

Examples of approaches that apply such a mechanism in the context of modelling tools are the following. The system developed by Burgueno and Cabot [7] uses a recommender system that is trained on various texts and suggests potential next elements for a model. The modeller then needs to select the element to be inserted out of these suggested options. A similar approach is the one by Savary-Leblanc and LePallec [25]. Their approach uses a recommendation system based on a set of similarity rules that is trained on a large collection of UML models.

On the other hand, some approaches operate in batch-style; i.e. the modelling assistant takes a large piece of text - typically a text that includes requirements and/or domain descriptions - and then produces, without further interaction with the modeller user, a UML model. An example of this approach is the work by [2].

We combine the batch- and interactive-approaches in our architecture: we first apply NLP to a large text. Our finding is that text in natural language always contains ambiguities, noise, and omissions that make it impossible for any NLP approach to produce a correct and complete model automatically. Therefore, we present the resulting model to the developer for feedback and refinement. Our approach includes showing the model in a UML editor and utilizing a (simple) executable prototype of the resulting system. We believe that we can achieve a high level of automation in this way, and offer mechanisms for rapid feedback and refinement.

III. Vision

We envisage the combination of batch NLP processing with an interactive conversational human-in-the-loop approach in a modular tool architecture. An overview of the architecture can be found in Figure 1. The NLP pipeline consists of the following components:

1) *Speech to text:* Utilises advancements in transformer architectures for speech recognition, such as wav2vec 2.0 [3], to transcribe audio fragments that describe requirements. This allows for interviews with domain experts and end-users to be used as a source for UML modelling.

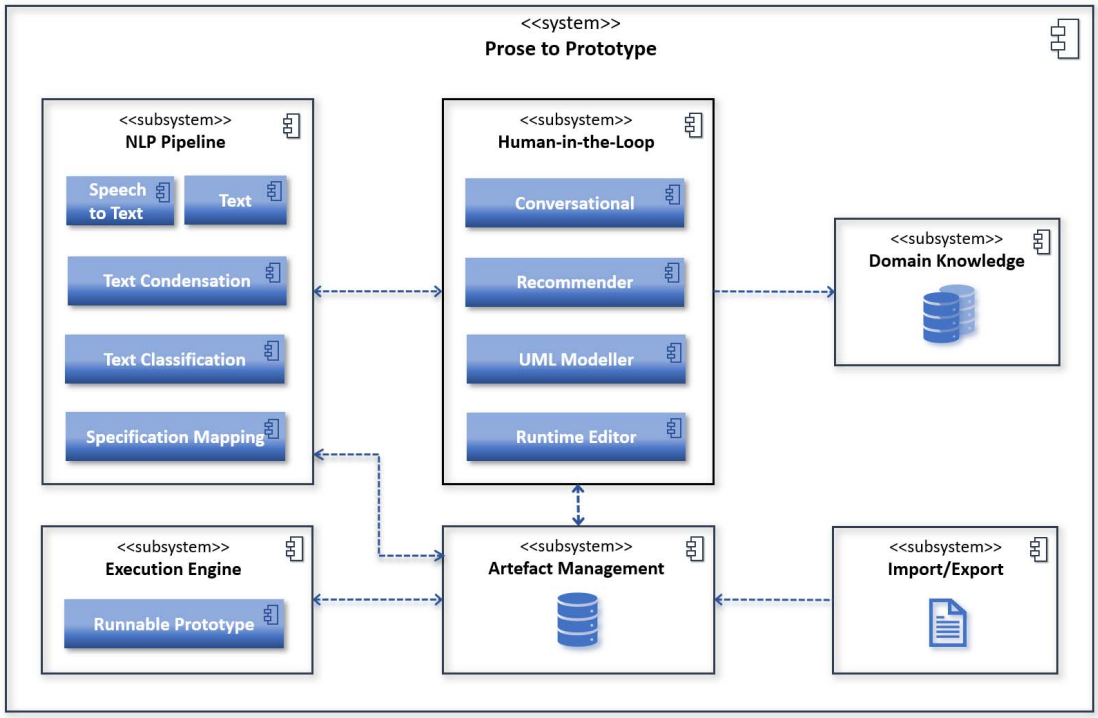

Fig. 1. Prose to Prototype Architecture

2) *Text Condensation:* Unsupervised topical summarisation models to remove fuzziness and variability in writing styles, combined with topic suggestions obtained by domain-specific data mining of open UML repositories.
3) *Text Classification:* Classification models such as VerbNet, combined with syntactical dependency analysis graph patterns obtained from SyntaxNet, for the targeting of text fragments into specific UML sub-model 'buckets';
4) *Specification Mapping:* The application of BiLSTM and BERT based models such as OpenIE and Semantic Role Labelling to the bucketed input text for Part-of-Speech (POS) tagging and keyword analysis to generate UML specification models.

The Text component is not mentioned in the pipeline above, but can be found in the architecture image in Figure 1. The Text component enables the pipeline to accept textual inputs in addition to the Speech to Text component. We can generate an application from natural language with this pipeline, but several other components are important to enhance our solution. We propose five additional components that enhance the application and enable users to interact with our solution. These components can be found in Figure 1, labelled 'subsystem'.

In straightforward cases, the pipeline will directly feed the execution engine component. This component enables the execution of the specification in the form of a *runnable prototype*. The execution engine interprets the UML model at runtime to provide a prototype application with 'live' data values such that end-users and domain experts can evaluate the system specification.

The human-in-the-loop component allows users to modify the outcome of the NLP pipeline by interacting with several levels of the pipeline. In this context, the 'user' is expected to be the modeller user where recommendations, suggestions, or clarification questions are focused on the UML representation of the specification, for instance, when using the UML modellers.

However, interaction regarding suggestions or questions pertaining to the domain is equally relevant to the domain expert - as long as it is phrased in natural language. For example, the system may query the domain expert whether a noun described in the text as having several properties may also have an unidentified property frequently found in similar elements, based on data mining of documents or models in the same domain. The human-in-the-loop component consists of the following sub-components:

1) *Conversational* component allows users to converse with the system on several levels, such as clarifying ambiguities or adding missing information.
2) *Recommender* component proposes instances that might interest the user, such as classes, attributes, or actions in the same domain models.
3) *UML modeller* enables the modeller user to visualise and change the generated models from a visual perspective.
4) *Runtime Editor* component allows the user (end-user,

domain expert or modeller user) to make changes to the UML model with easy to use dialogs to add missing fields or operations during the execution of the prototype. Any changes are immediately reflected in the running prototype.

The usability of tools for modelling has frequently been mentioned as a key factor in adopting modelling practices in software development [31].

The domain knowledge component contains existing UML models and requirements texts organised by business and application domain. Based on this data, the human-in-the-loop component will make recommendations and have conversations based on domain knowledge. The domain knowledge will be updated over time to ensure recommendations and conversations will be improved. The aim is to reuse existing data sources such as the Lindholmen dataset [14], MAR [17] and GitHub. Data sources should be configurable in order to be extended in future. As the format of the various data sources will inevitably differ, the domain knowledge component needs to provide an API to transform the data into the standard internal model and text format.

The artefact management component is a large storage component that provides a common API for other components to access UML model data and other artefacts. Several artefacts will be stored here, such as UML models, requirement texts, code snippets, and more. The objective of the component is to collect artefacts and find links between the different artefacts, which will allow our solution to make better decisions and, therefore, prototypes.

Finally, the import/export component allows the modeller user to import the specification models generated by the system into other tools for application implementation. We aim to support XMI files to utilise development tools that support this UML interchange format. Low-code platforms could be enabled by means of a mapping to their proprietary metadata format.

The vision of the P2P solution shows a promising way of defining models, where stakeholders are continuously up to date on the specification model and its prototype applications. The prototypes generated with the P2P solution will not be 100% correct as a result of the NLP transformation alone, as language is inherently complex and ambiguous. For this reason, the P2P solution contains interactive techniques that enable tool users to fit the models to their needs.

IV. Current Status

An initial prototype of the P2P tool environment has been developed as a proof of concept. It is constructed using a three-tier technical architecture:

- a web-based front-end where the tool user interface components execute. The user interface components are developed in Django and Typescript. A browser-based front end allows for easy deployment and offers access to the different stakeholders without requiring a complicated setup on a local machine.
- an application server where NLP processing takes place. The front end communicates through an API to a middle-tier implemented in Python, with some wrapped Java components. As the processing requirements of the NLP and AI components grow, we expect the middle tier to be spread across multiple machines in a cloud infrastructure going forward.
- a back-end database tier used to store requirements texts, diagrams, UML models, and Python code snippets added for model scripting. The database also stores the instance data that end-users add when executing the runnable prototype.

The first implementation activity of the P2P project started in 2018. As discussed in Section III, the architecture consists of several components, organised into sub-systems. In the remainder of this section, we will briefly describe the current status of the project in terms of components that have been implemented or are currently being worked on. The planned components whose implementation has not yet been started will be further discussed under future directions in Section V.

A. NLP pipeline components

The *Text input* component has been implemented as part of the current UML class transformer. Texts can be loaded, edited, and stored in the artefact component. It is also possible to configure and invoke (free) online speech-to-text conversion services (e.g. as provided by Microsoft and Google) by loading an audio recording. However, initial results with these services have been disappointing so far, leading to numerous errors in the output text. In addition, the online editing functionality is limited in its formatting capability.

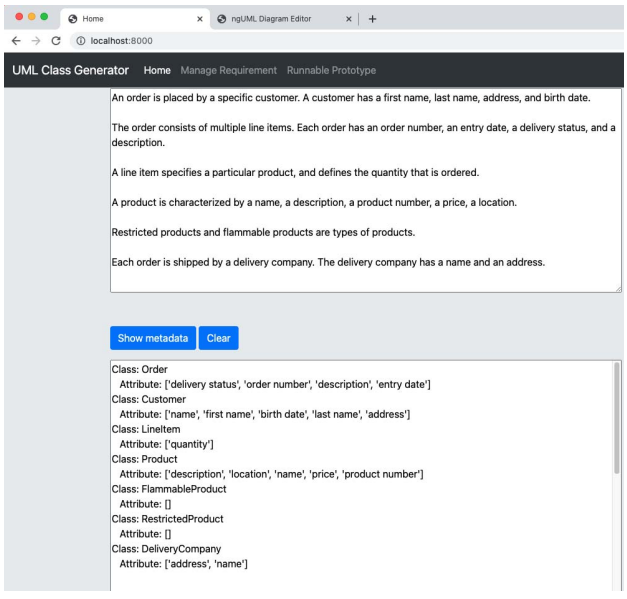

Fig. 2. Screenshot of the Natural Language View

The *Text condensation* step is in development as a master thesis project. Currently, a running prototype uses a transformer model for abstractive summarization of small pieces of requirements texts to reduce the amount of unneeded information in the texts. A current limitation is that irrelevant topics are not filtered out of the summary. To address this issue, a topical attention mechanism as introduced by Bahrainian et al. [4] will be implemented, extending our vision of human interaction with the pipeline: the user can select or deselect topics of interest in the requirements text, which will then be used to generate the abstractive summary.

The *Text classification* component is under development within the same project. It aims to create a framework for a data mining-based mechanism to enrich requirements texts by using machine learning combined with large datasets of UML files. Currently, to extend the effectiveness of generated class models, these large datasets are used to create a binary classifier to classify observed objects as either classes or attributes, a problem that has been identified as early as [12]. One current limitation that needs to be addressed is the issue of text format mappings due to domain naming conventions. For instance, 'customer name' is found as 'CUSTNAME', 'CustomerName', and other variants. A similar issue exists for domain-specific attribute data type mappings.

The *Specification mapping* has a finished prototype that can extract classes, attributes and relationships from natural language text and generates a UML class specification. It is built using Stanford CoreNLP [18] and OpenIE [1] and developed by Tang [27]. An example of the transformation of text to UML class models can be found in Figure 2. This figure shows the requirements text input area at the top, and a readable version of the extracted UML model beneath it. This information is also stored in the UML model in the middle tier and back-end.

Although this implementation successfully removes the requirement of forcing the domain expert to phrase overly structured requirements text, it has limitations in dealing with variations in informal writing styles and in dealing with real-world requirements documents that typically mix structural and behavioural elements. Initial results from the *Text condensation* and *Text classification* components show promising results in addressing these issues, respectively.

In addition, the current implementation is limited to structural class models only. Work is currently under way to extend this with a transformation component focused on UML activity models. An objective of the activity extraction from requirements text is to use ML transformer models in combination with the more traditional NLP parsing approaches.

B. Execution engine component

The execution engine component provides a runtime environment for incrementally creating prototypes based on the UML model [11]. These runnable prototypes are created in Python using the Django framework and execute in a 3-tier architecture, with the front-end running in a web browser. This runtime environment enables i) the services to run the

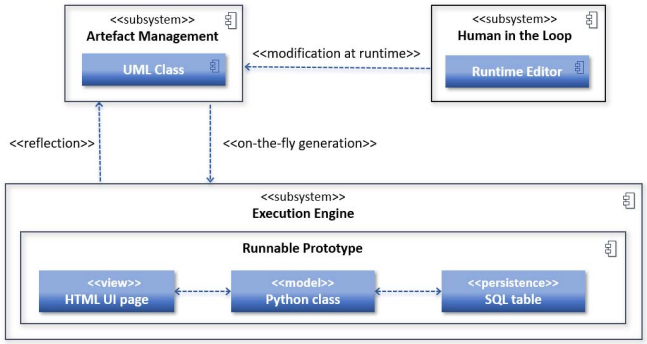

Fig. 3. Mapping of the UML class to elements in the runnable prototype

prototype, i.e. to enter or remove instance data, and ii) to incrementally generate the Django code when a modification is made to the UML model. This runtime environment performs on-the-fly reflection on the model. Hence, if any changes are made to the model, these are immediately reflected in the running prototype.

By default, classes in the UML model are interpreted as data types in the prototype. Then for every data type, the execution engine creates a table in the database, a middle-tier for retrieving data from and entering data into the database, and a UI page that provides a view where the end-user can perform CRUD operations on these data. This is illustrated in Figure 3. If a class is meant to contain algorithmic/computational logic, this can be done by manually adding Python code to this class. These code snippets are stored with the UML model and inserted at runtime.

The *runnable prototype* enabled by the execution engine component allows users to interact with the generated specification and enter instance data. This is an effective way for end-users and domain experts to evaluate the completeness of the specification. An example of a list view of an 'order application' is shown in Figure 4.

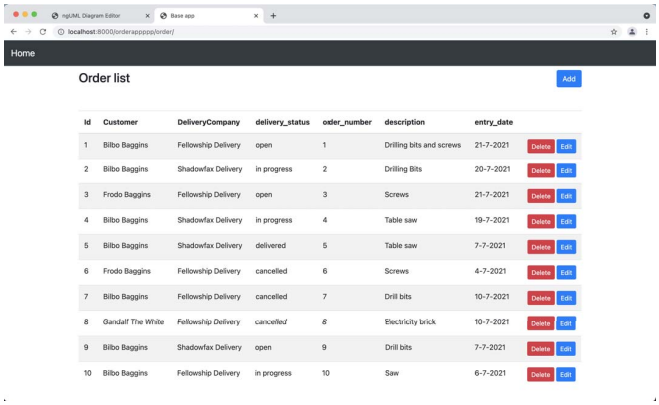

Fig. 4. Screenshot of the runnable prototype showing an 'order application'

The current implementation provides a straightforward user interface only. A current project is underway to expand the

visual web page layout to provide a more wireframe oriented visualisation. These representations are not static wireframes.

Another limitation of the current runtime environment is that it is based on the structural class model only. A second project will provide metadata-driven dynamic execution of these wireframes based on the UML Activity models. The goal is that requirements texts that describe process aspects such as business processes, workflows or application flows can be transformed into UML models. The resulting UML model will be interpreted at runtime to drive the dynamic execution of the prototype. This will extend the functionality of the prototype beyond simple CRUD operations.

It should be noted that the goal of the execution component is to provide an integrated tool capability to verify the specification model. Its goal is not to provide a (low-code) development environment. A project to provide bi-directional mapping functionality to XMI and a low-code platform is in its initial stages.

C. Human-in-the-loop components

The human-in-the-loop component currently consist of two implemented components that enable the modification of the UML model after it has been generated by the NLP components.

The *UML modeller* component provides a diagrammatic way of interacting with the model in a drag and drop web environment and is focused on the UML expert developer. In this application, the modeller user can create, read, update and delete the entities. It also has a built-in component for structured editing of Python code snippets inserted for model scripting. The application currently visualises class and activity models.

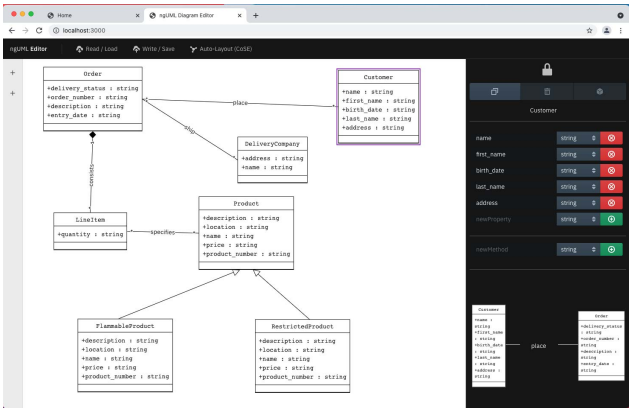

Fig. 5. Screenshot of the Class Modeller

The modeller enables the UML expert to visually interact with the NLP pipeline outcomes and make updates where the transformation from text was incomplete. Figure 5 shows a visual class diagram representation in the modeller. The current implementation has limitations in its auto-layout functionality, resulting in lines overlapping on the diagram and incomplete grid management. A bachelor project has been initiated to implement a combination of generic graph management with UML specific diagram heuristics.

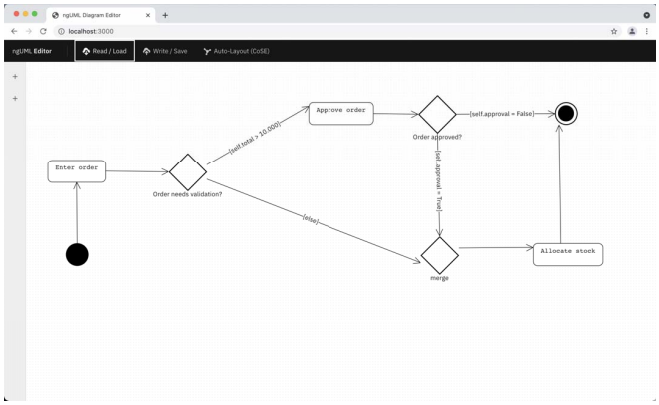

Fig. 6. Screenshot of the Activity Modeller

Work is currently in progress to add a first behavioural UML model to the tool environment in the form of Activity models. It will use a similar approach of combining an NLP transformer and a visual diagrammer. Figure 6 shows a screenshot of this activity modeller.

The UML class model can also be updated using dialogues of the *runtime editor* while the prototype is running, as illustrated in Figure 7. It provides a dialog oriented user interface to modify the UML class model. It is focused on usage by the end-user who wishes to try out any modifications when running the prototype. Any changes, such as adding a property to a class, adding a relationship between two classes, or larger changes such as adding a new class, are immediately reflected in the prototype after a browser page reload. Any changes will carry forward the instance data that the user has entered through automatic database migration in the background.

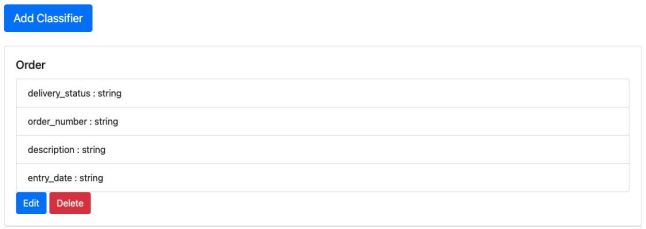

Fig. 7. Screenshot of the runtime model modification dialog for the definition of an 'order' class in a running prototype

D. Artefact management component

Within the P2P application, several types of data are used. The objective of the *artefact management component* is to manage these instances and make them available for query-based retrieval by other components. The collection of these instances enables us to reuse this data in future. Currently, the artefact management component manages requirements texts, UML models, UML diagrams, and Python code snippets. The

artefact management component is the basis for the metadata-driven prototype execution and also stores any instance data that the user enters into the prototype.

V. Future Directions

This section describes components of the architecture that need to be addressed going forward and issues that need to be resolved in order to validate our approach effectively.

To support complete coverage of UML models that are relevant for the requirements gathering process [10], we envisage adding *UML Use Case and UML Component models*. This requires extending the UML meta model, adding NLP language transformers, adding visual diagrammers, and, lastly, extending the executable prototype with additional functionality based on the extended meta model.

Adding a use case-based component will also provide an opportunity to address an omission from our approach. Currently, the focus of our approach has been on 'classic' UML based requirements. However, agile-based methods deploy user stories and user scenarios. These are subtly different in two ways: i) they mix structural and behavioural requirements in one document, and ii) they typically focus on 'instance level' descriptions (rather than generic or 'type level' requirements). Concerning the former, similar issues occur in traditional requirements documents, and we hope to address them through the *Text classification* component. The latter, however, requires generalising instance-level descriptions and composing multiple user stories into a single type level description.

Furthermore, several components need to be added to complete the human-in-the-loop interactive component of P2P architecture. In particular, the recommender component and the conversational component.

An interesting approach to applying *recommender* technology in UML context is described in Savary-Leblanc et al. [25]. We have initiated collaboration with the authors with the intention of reusing their recommender API. The technology will recommend entities such as classes or activities to the user. This will require configuring it to the requirements domain, as well as configuring relevant data mining sources.

Previous work in applying *conversational* chatbot like technology in the domain of UML models is described in Pérez-Soler et al. [24]. We will investigate if such an approach can be extended to cover contextual conversations based on causality chains in complex models and across external domain models. We have experimented with using voice as a modality for input to modelling tools on the input side [8].

Import/export functionality should enable validated specifications to be exported in XMI format for full implementation. A wide variety of both commercial and open-source UML tools support this format. These tools can then be used to generate code from the specification in the target languages they support to implement the application. Low-code development tools could similarly be targeted by mapping to their proprietary meta models. We will investigate if the Eclipse Modeling Framework (EMF) can be reused and customised to implement our import/export functionality.

In addition to adding the components above, two areas in our architecture require refactoring and extension to support incremental, iterative development in the context of a model-based tool with an NLP front-end.

Requirements gathering is inherently an incremental process, where new sections of requirements text are added to existing documents. Our current implementation does not support this. As a result, any modifications and additions that have been made to the existing UML output will be lost. Supporting incremental development will require diff/merge functionality between current and new requirements documents and driving the language transformers only with the changes that have been made to update the UML model.

That, however, only solves part of the problem. If changes are made to the UML model after transformation that impacts the functionality of the system, then these should be reflected in the textual requirements documents. A simple example is adding an attribute to the UML model, which should be reflected in the textual requirements. If this does not happen, the textual requirements do not form a complete representation of the specification. This requires the ability to update the requirements document based on changes to the UML model.

Going 'back' from UML models to text is helpful in two ways: first, it aids stakeholders that do not have the knowledge to understand technical models [6], and second, it keeps documentation in software projects up-to-date, even when project details change. Therefore, a future research direction that extends this project would be to reverse the process: going from "prototype to prose", where the models form the basis to generate requirements texts. With models such as GPT-3, generated natural language texts are getting almost indistinguishable from human-authored text [9]. We will investigate if this technology could be utilised in the P2P project to enable roundtrip, incremental NLP based UML modelling. If successful, it would also enable the option of automatically generating a textual explanation of any UML model.

Finally, significant issues remain to make our current tool implementation suitable for practical validation of the core NLP and AI components, which are the focus of our research. To do so, our prototype needs to be made publicly available, allowing others to interact with the tool and save models. This should provide an opportunity for experimentation and critical feedback. However, public availability will require several implementation issues to be addressed. In particular:

- Robustness. Automated scenario-based testing needs to be added to the development environment to address the brittleness of the current prototype;
- Scalability. The environment has only been tested on small scale publicly available requirements examples. We hope to obtain larger scale real-world requirements samples from industrial partners;
- Error Management. Errors or missing values in the generated UML model may lead to exceptions in the execution component (leading to a Python debugging environment, which is unacceptable for end-users). Rigorous constraint

checking is needed both at the model level, as a precondition for running the execution engine;
- Deployment. Web deployment will be important to engage with research and testing partners going forward. Currently, the prototype tool environment has been developed on GitHub but has only been deployed on local machines. Its architecture should enable deployment to a cloud infrastructure, but issues are likely to arise.

We want to make an open repository of requirements texts with associated UML models available to the community in the long term, where others could contribute. If successful, this would open up an opportunity for research into supervised ML approaches in the area of mapping natural language to UML.

VI. Conclusion

We presented a vision for automatically generating a UML specification from natural language requirements - either in text or in spoken form. We believe the unique aspects of our approach are:

- The ability to transform unstructured requirements text into UML specifications;
- An advanced NLP approach that includes steps for text condensation, text classification and specification mapping that supports real-world requirements texts;
- Offering both batch and interactive mechanisms for synthesising a UML model, including visual modellers;
- A data mining component that enables suggestions to the user;
- Generation of an executable prototype which enables rapid user validation.

Once further components have been implemented and the issues discussed have been addressed, we believe our prototype development tool can be used to verify that the techniques deployed enable more rapid development of high-quality specifications. In addition, they will enable domain experts and end-users to be more effectively involved in the requirements specification process.

References

[1] G. Angeli, M. J. J. Premkumar, and C. D. Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 344–354, 2015.

[2] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer. Extracting domain models from natural-language requirements: approach and industrial evaluation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 250–260, 2016.

[3] A. Baevski, H. Zhou, A. Mohamed, and M. Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *arXiv preprint arXiv:2006.11477*, 2020.

[4] S. A. Bahrainian, G. Zerveas, F. Crestani, and C. Eickhoff. CATS: Customizable Abstractive Topic-based Summarization. 2021.

[5] W. Ben Abdessalem Karaa, Z. Ben Azzouz, A. Singh, N. Dey, A. S. Ashour, and H. Ben Ghazala. Automatic builder of class diagram (ABCD): an application of UML generation from functional requirements. *Software: Practice and Experience*, 46(11):1443–1458, 2016.

[6] H. Burden and R. Heldal. Natural language generation from class diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–8, 2011.

[7] L. Burgueño, R. Clarisó, S. Gérard, S. Li, and J. Cabot. An NLP-based architecture for the autocompletion of partial domain models. In *International Conference on Advanced Information Systems Engineering*, pages 91–106. Springer, 2021.

[8] R. Capilla, R. Jolak, M. R. V. Chaudron, and C. Carrillo. Design decisions by voice: The next step of software architecture knowledge management. In *International Conference on Human-Centred Software Engineering*, pages 166–177. Springer, 2020.

[9] R. Dale. GPT-3: What's it good for? *Natural Language Engineering*, 27(1):113–118, 2021.

[10] O. Dawood and A.-E.-K. Sahraoui. From requirements engineering to uml using natural language processing – survey study. *European Journal of Engineering and Technology Research*, 2(1):44–50, Jan. 2017.

[11] R. Driessen. UML class models as first-class citizen: Metadata at design-time and run-time. *Leiden University. Leiden Institute of Advanced Computer Science (LIACS)*, pages 1–42, 2020. BSc thesis.

[12] G. Fliedl, C. Kop, and H. C. Mayr. Recent results of the NLRE (natural language based requirements engineering) project. *EMISA Forum*, 24(1):24–25, 2004.

[13] H. M. Harmain and R. Gaizauskas. CM-builder: an automated NL-based CASE tool. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 45–53. IEEE, 2000.

[14] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use UML: Mining github. In *Proceedings of the ACM/IEEE 19th Int Conference on Model Driven Engineering Languages and Systems*, MODELS '16, page 173–183. ACM, 2016.

[15] D. Hu. An introductory survey on attention mechanisms in NLP problems. In *Proceedings of SAI Intelligent Systems Conference*, pages 432–448. Springer, 2019.

[16] C. Lemaigre, J. G. García, and J. Vanderdonckt. Interface model elicitation from textual scenarios. In P. Forbrig, F. Paternò, and A. M. Pejtersen, editors, *Human-Computer Interaction Symposium*, pages 53–66, Boston, MA, 2008. Springer US.

[17] J. A. H. López and J. S. Cuadrado. Mar: a structure-based search engine for models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 57–67, 2020.

[18] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proc. of 52nd meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[19] P. More and R. Phalnikar. Generating UML diagrams from natural language specifications. *Int. Jnl. of Applied Information Systems*, 1(8):19–23, 2012.

[20] C. R. Narawita and K. Vidanage. UML generator –use case and class diagram generation from text requirements. *International Journal on Advances in ICT for Emerging Regions (ICTer)*, 10:1, 1 2018.

[21] M. H. Osman, M. R. V. Chaudron, and P. Van Der Putten. Interactive scalable abstraction of reverse engineered UML class diagrams. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 159–166. IEEE, 2014.

[22] S. P. Overmyer, L. Benoit, and R. Owen. Conceptual modeling through linguistic analysis using lida. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 401–410. IEEE, 2001.

[23] H. G. Perez-Gonzalez and J. K. Kalita. Gooal: a graphic object oriented analysis laboratory. In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 38–39, 2002.

[24] 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.

[25] M. Savary-Leblanc, X. Pallec, and S. Gérard. A recommender system to assist conceptual modeling with UML. *International Journal of Software Engineering and Knowledge Engineering*, 2021.

[26] R. Sharma, S. Gulia, and K. K. Biswas. Automated generation of activity and sequence diagrams from natural language requirements. In *ENASE - Proceedings of the 9th Int. Conference on Evaluation of Novel Approaches to Software Engineering*, 2014.

[27] T. Tang. From natural language to UML class models: An automated solution using NLP to assist requirements analysis. *Leiden University. Leiden Institute of Advanced Computer Science (LIACS)*, pages 1–114, 2021. Thesis Master ICT in Business, Leiden University.

[28] L. Teixeira, V. Saavedra, C. Ferreira, J. Simões, and B. Sousa Santos. Requirements engineering using mockups and prototyping tools: Developing a healthcare web-application. In S. Yamamoto, editor, *Human Interface and the Management of Information. Information and Knowledge Design and Evaluation*, pages 652–663, Cham, 2014. Springer International Publishing.

[29] M. O. Topal, A. Bas, and I. van Heerden. Exploring transformers in natural language generation: Gpt, bert, and xlnet. *arXiv preprint arXiv:2102.08036*, 2021.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[31] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial adoption of model-driven engineering: Are the tools really the problem? In *International Conference on Model Driven Engineering Languages and Systems*, pages 1–17. Springer, 2013.

[32] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[33] T. Yue, L. C. Briand, and Y. Labiche. A systematic review of transformation approaches between user requirements and analysis models. *Requirements Engineering*, 16:75–99, 6 2011.

[34] T. Yue, L. C. Briand, and Y. Labiche. aToucan: An Automated Framework to Derive Uml Analysis Models from Use Case Models. *ACM Transactions on Software Engineering and Methodology*, 24, 2015.