



Universiteit
Leiden
The Netherlands

Opleiding I&E

A RUNTIME ENGINE FOR INTERPRETING UML ACTIVITY META DATA FOR PROTOTYPE EXECUTION

Taro Spruijt 2599325

First supervisor: Dr. Guus Ramackers
Second supervisor: Prof. Dr. Joost Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

10/08/2022

Abstract

Model Driven Engineering is a growing practice aiming to enhance software development from the generation of models. Whereas previous research has thoroughly studied the rationale, users, challenges, usage and industries of Model Driven Engineering none have reviewed a runtime engine for prototype execution. Particularly, UML activity diagrams and interpreting their subsequent meta data for prototype execution is of interest. As such, this paper aims to tackle exactly this. While a project containing static pages in which users create activity diagrams and enter data exist, this paper adds behavioural and workflow to enable prototype execution.

This was done by making users link pages they create to nodes of an activity diagram in order to display these pages. As a result these pages can be displayed in the order relevant to the activity diagram. By making users link pages to nodes a sequential order of nodes was generated with their respective pages. Subsequently, all pages will be shown in order of the nodes of the activity diagram when a user wishes to execute a prototype. In addition, users are able to manually move on to next nodes and browse through the pages by clicking a “next page” button.

Even though an engine for prototype execution was made, full functionality to interpret activity diagrams is not entirely implemented. Specifically, decision, fork and join nodes have no functionality yet. Another essential component which has not been included in the runtime is the interpretation of edges between nodes. Future research should extend the runtime engine by adding interpretation for all types of nodes of an activity diagram and the interpretation of edges between nodes. Lastly, this research and development is part of the ngUML/Prose to Prototype Software Development Environment project at Leiden Institute of Advanced Computer Science (LIACS).

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Task definition	1
1.3	Approach	1
1.4	Goal	1
1.5	Contribution	1
1.6	Thesis overview	2
2	Background and Related Work	2
2.1	Unified Modelling Language	2
2.1.1	Activity diagrams	2
2.2	Current state	4
2.3	Desired state	5
2.4	Related work	7
2.4.1	Rationale behind MDE	7
2.4.2	MDE's industries	8
2.4.3	Users of MDE	8
2.4.4	Challenges for MDE	8
2.4.5	Usage of MDE	10
3	System Design	11
3.1	Logical Design	11
3.1.1	Requirements	11
3.1.2	Overview	11
3.2	Technical Design	12
4	Worked Examples	12
4.0.1	Page creation	13
4.0.2	Workflow example 1	17
4.0.3	Workflow example 2	23
5	Conclusion	24
	References	26

1 Introduction

1.1 Introduction

Unified Modelling Language (UML) is a set of modelling standards widely used as the norm to create diagrams. However, making diagrams manually may be timely. As a result, automation of the process of developing diagrams could greatly reduce the amount of time and resources needed to model and visualizing software projects. As a consequence, time previously allocated to developing models could now be spent elsewhere. This may result in more efficiency within software engineering.

Most diagrams should usually not be used as static artefacts. Namely, diagrams are often used to describe the behaviour and functionality of a prototype at runtime. This means that diagrams are not just used as an image to view, but diagrams are used to show how a certain prototype or software element should behave under certain circumstances. This way, the creator of a diagram can easily show a developer for example how they want some software to behave and function at runtime.

1.2 Task definition

Previous research has already created the ability to automatically create activity diagrams. This was done by taking text input from a user specifying their requirements which are translated into Python classes using Natural Language Processing. Currently, the existing project has no functionality to interpret UML class metadata or prototype execution. There are only static pages where the user enters data. This needs to be extended with behavioural aspects for workflow and application flow to be able to execute more real life prototypes.

1.3 Approach

Namely, automatic execution at runtime for processes is not integrated yet. Automatic execution at runtime enables users to perform tasks without any lag. In addition, the final prototype no longer consists of static pages but now also supports linking pages within an application or workflow. As such, users are able to easily generate activity diagrams and perform tasks. This will be done by adding interpretation to activity nodes.

1.4 Goal

Thus, the goal of this research is to develop software which is able to perform processes at runtime. That is, the aim of the software is to be able to take user inputs specifying application or workflow processes used to make a UML activity diagram after which the process specified must be executed at runtime. However, there is no backend which can automatically execute the processes yet.

1.5 Contribution

This report will contain two main details.

- Firstly, a description of the software created will be given.

- Secondly, examples of the software created will be shown.

1.6 Thesis overview

Furthermore, this report is part of a bachelor thesis at the Leiden Institute of Advanced Computer Science and is supervised by Guus Ramackers. Also, the report is divided into five chapters. Firstly, this chapter contains the introduction; Section 2 describes both background information needed to understand this research and research related to this work; Section 3 discusses both the logical design of the system as well as the technical design; Section 4 describes the created software and shows examples of its functionalities; Section 5 discusses challenges regarding the research, concludes and discusses potential further research.

2 Background and Related Work

2.1 Unified Modelling Language

Unified Modelling Language (UML) is a standard which is often used by diagram makers to which different forms of diagrams should conform to. For example, flowcharts (charts which visualize the steps of a process) can be visualized in an endless amount of ways. In order to align the interpretation of readers of diagrams, UML has rules which describe how certain diagrams should be visualized so making and reading diagrams is easier.

2.1.1 Activity diagrams

The main component of UML which this report studies are activity diagrams. Activity diagrams are a type of diagram which describes the steps a system must perform [act]. In essence, activity diagrams portray the flow of activities. An example of an activity diagram can be seen in Figure 1. In order to display the connection between activities, activity diagrams exist of nodes which are connected to each other using directed arcs.

There are different forms of nodes in an activity diagram. For this project the following nodes can be used to create an activity diagram with.

- Initial node: this node shows where the activity flow starts. It is denoted by a fully black circle (see Figure 2).
- Action node: the action node displays an action which must be performed. It is drawn as a rectangular box with text of the action to be performed within the box (see Figure 3).
- Decision node: the decision node is used when a condition must be checked. For example, “is the order total above 10000?” The node is denoted by a diamond with the text for the decision underneath it (see Figure 4).
- Merge node: the merge node is used to close after a decision. The merge node takes inputs from the paths created after a decision node and brings these paths back together. The node is denoted by a diamond with the text for the merge underneath it (see Figure 5).

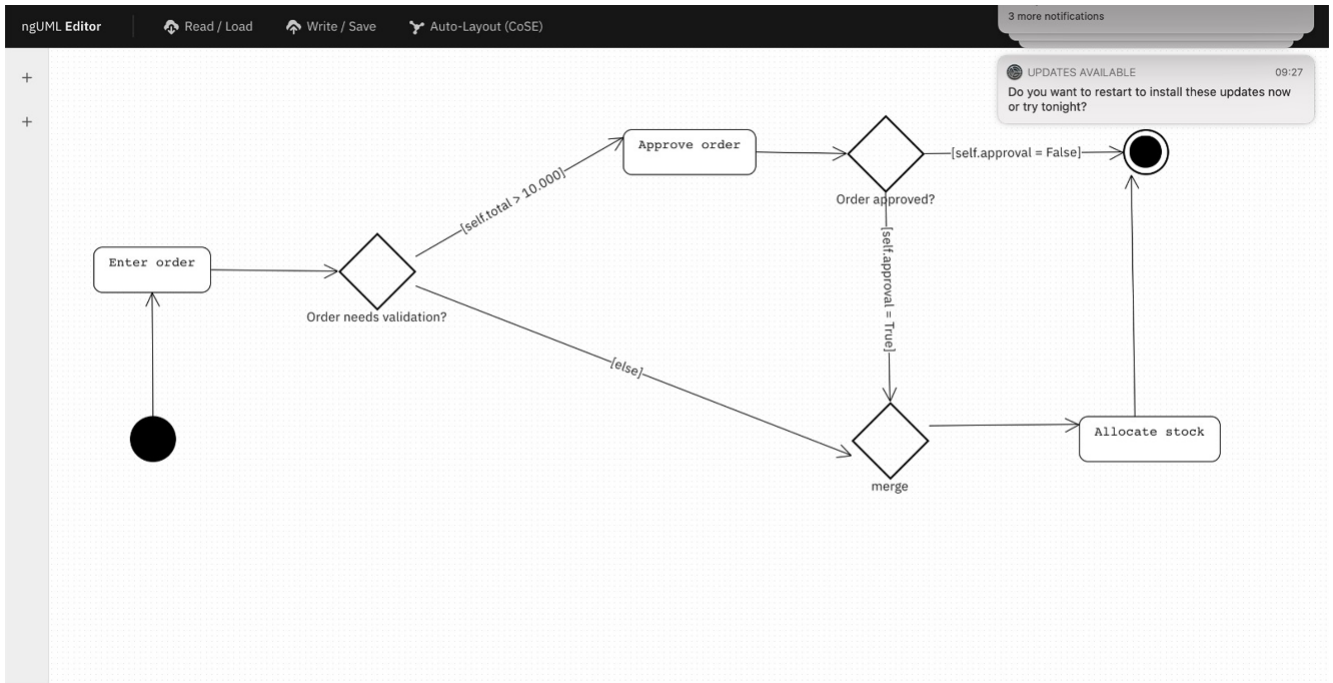


Figure 1: Example of an activity diagram

- Fork node: the fork node splits the behaviour into parallel flows or actions. It is denoted by a thick black vertical line with the text for the fork underneath (see Figure 6).
- Join node: the join node connects the flows or actions after a fork node has split the behaviour into parallel flows or actions. The flows or actions are brought back together by a join node. The join node is denoted by a thick black vertical line with the text for the join underneath (see Figure 7).
- ActivityFinal node: the activity final node is used to stop all flows in an action. It is denoted by a circle with a thick black dot in it a thick black vertical line (see Figure 1)
- FlowFinal node: the flow final node denotes the exit of a system. The difference between a flow final node an activity final node is that a flow final node ends all tokens that arrive in it and has no effect on other flows and actions, whereas an activity final node ends an entire flow or action.



Figure 2: An initial node

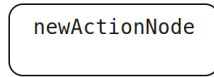


Figure 3: An action node

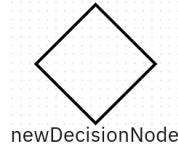


Figure 4: A decision node

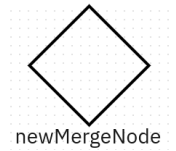


Figure 5: A merge node

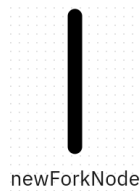


Figure 6: A fork node



Figure 7: A join node

2.2 Current state

Currently, there are two main components to the system: the backend and the editor. Moreover, there are static pages in the backend in which users are able to enter data (see Figure 8).

First, users enter requirement text on this page. For example: "a product is characterized by a name, a description, a product number, a price and a location." This will result in a Python class: Class: Product Attribute: ['description', 'price', 'location', 'name', 'product number'] (see Figure 8). As a result, a class diagram as seen in Figure 9 below will be generated in the editor. Subsequently,

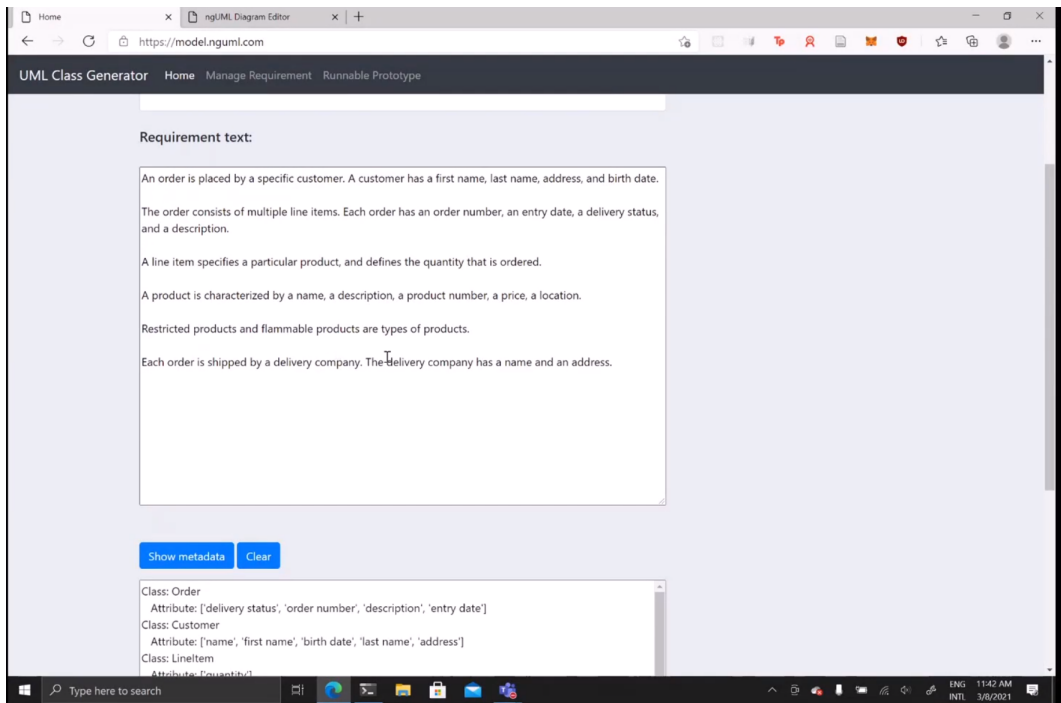


Figure 8: The requirement text

models can be created (see Figure 10). Also, the properties of the classes (i.e. the attributes) can be filled in (see Figure 11) with live data.

Second, users create an activity diagram manually. However, there is no flow to execute processes at runtime. That is, there are only static pages in which users are able to enter data. There is no ability to automatically perform the actions described in the activity diagram yet. Thus, the pages must be extended with behavioural aspects to be able to execute real life prototypes.

2.3 Desired state

In order to automatically perform prototypes, workflow between pages must be added. As such, the desired state is a software which is able to interpret process specifications at runtime in order to add pages, which are part of an application or workflow. As a result, pages must be shown at runtime for each node in an activity diagram. For example, see Figure 1.

In this activity diagram the user starts by entering an order. As such, an enter order page must be shown to the user. When the user has clicked the enter order button a decision node is reached. Then, depending on whether the user needs to approve the order, a merge node is reached or the user must validate the order. If the order is validated the merge node will also be reached. Then, stock must be allocated. Afterwards the activity flow will be finished if the order was not approved or stock was allocated.

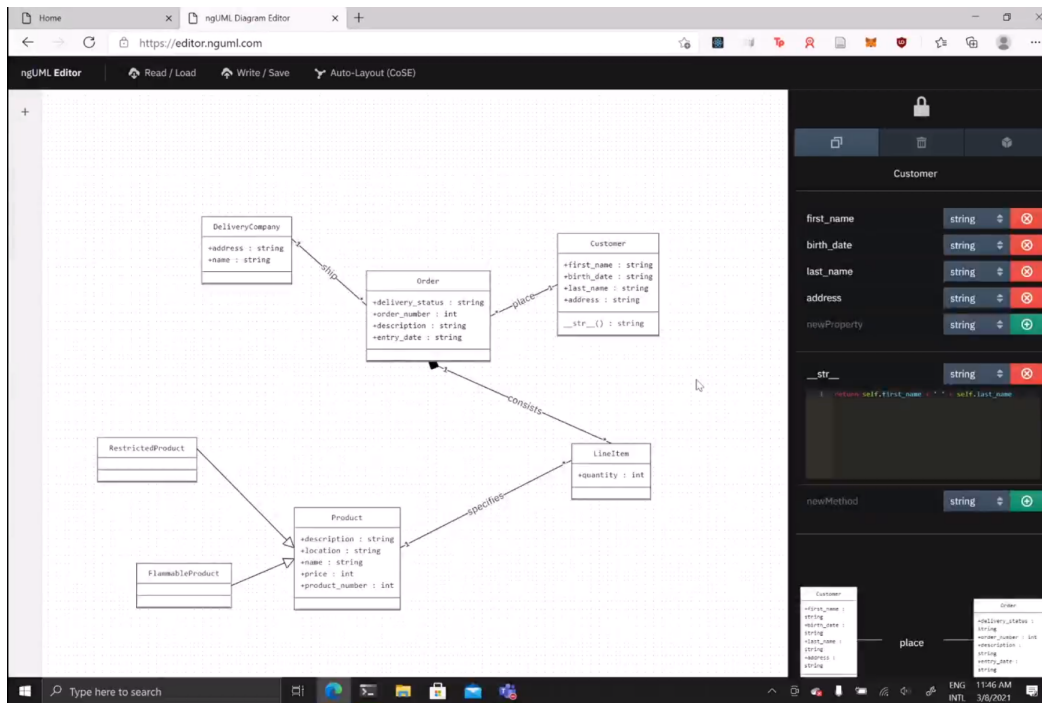


Figure 9: Class diagram of the product order requirements

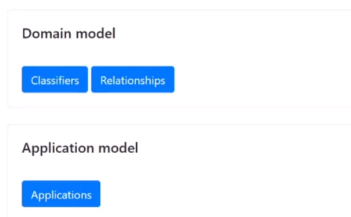


Figure 10: The models which can be created

Add Product

description

name

price

product_number

Figure 11: Data which can be filled in

2.4 Related work

Whereas this paper analyzes the usage and creation of executable prototypes, other research has been done on similar subjects. Namely, two main similar subjects have already been studied quite thoroughly. These two topics are executable specifications and model driven engineering. Executable specifications are the primary category where the subject studied in this paper falls under. Nevertheless, model driven engineering is not of much lesser importance. From here on out executable specifications and model driven engineering will be denoted by MDE.

In order to fully understand the importance and relevance of this paper’s research, the current state of the art can be evaluated. For this, five different topics are key. The main questions which are answered by other research are: “why is MDE used?”, “where is MDE used?”, “by whom is MDE used?”, “what are the challenges with MDE?” and “how is MDE used?.” When these questions have been answered, it should be clear to see why MDE is an important topic of study.

2.4.1 Rationale behind MDE

First, previous research has determined a rationale behind the usage of MDE. It has been studied what the central points are which users consider when adopting MDE and the potential usage MDE has in the 22 years from 2022. As such, a clear reasoning can simply be identified for the usage of MDE in the present as well as predict the future trend of MDE adoption.

The first paper explaining the why-question of MDE, discusses the acceptance of MDE in industry [MGSF13]. In this paper, Stefanescu et al. studied four large companies aiming to discover the most important reasons for adopting MDE. For this, data was collected covering tool evaluations, interviews and a survey. It was found that perceived usefulness, ease of use and the maturity of the tools were the key components to employ MDE.

The second paper covering the why-question proposes a potential future trend with regards to MDE usage [MAB⁺14]. In order to test the relevance of MDE in the future, Mussbacher et al. designed an experiment over the duration of a week. In this week, Mussbacher et al. united with 15 MDE experts to discuss challenges and future opportunities of MDE. In the end, four main challenges of MDE were identified. These four challenges identified are: Cross-Disciplinary Model Fusion, Personal Model Experience, Flexible Model Integration and Resemblance Modeling – From Models to Role Models.

Firstly, Cross-Disciplinary Model Fusion explains that MDE knowledge should be used across multiple disciplines. For example MDE should focus more on artificial intelligence, databases, the semantic web and human-computer interactions. Secondly, Personal Model Experience explains the task to make modeling and the use of models directly benefit the individual. Thirdly, Flexible Model Integration depicts “how software models should be structured to provide value when developing systems that flexibly address many concerns simultaneously” as stated by Mussbacher et al.. Lastly, Resemblance Modeling – From Models to Role Models shows that modeling should not be done in a too high level of abstraction.

2.4.2 MDE's industries

Second, it has been studied in what industries MDE is beneficial. A paper by Mohagheghi et al. does this by reviewing a case of three large industrial participants of a research project which aims at finding new techniques to use MDE on the development of large and complex systems [MGS⁺11]. As such, Mohagheghi et al. give a summary of the findings. As stated by Mohagheghi et al. these findings are that the participants think MDE is primarily convenient in “providing abstractions of complex systems at multiple levels or from different viewpoints, for the development of domain-specific models that facilitate communication with non-technical experts, for the purposes of simulation and testing, and for the consumption of models for analysis, such as performance-related decision support and system design improvements.”

Not only did the participants find MDE useful for the development of domain-specific models, they also found “a methodology useful and cost-efficient if it is possible to reuse solutions in multiple projects or products from the industrial perspective.” On the other hand, developing reusable solutions required extra effort and sometimes had a negative impact on the performance of tools. Mohagheghi et al. conclude that merging different tools in a seamless development environment requires several transformations which requires higher implementation effort and increases complexity.

2.4.3 Users of MDE

The third topic to be covered is the users of MDE. This topic has been studied in two ways. Firstly, an use case was studied of Motorola's [BLW05]. Secondly, an industry-wide study was performed to analyze the usage of MDE in practice [WHR14]. In the first paper studying Motorola, Baker et al. convey their experiences within Motorola. Baker et al. do this by analyzing their experiences gained over more than 15 years in deploying a top-down approach to MDE within Motorola.

Due to Motorola striving for lower development costs despite higher system complexity, Motorola has been using MDE for a while. For this, rigorous models are created throughout the development process in order to introduce automation. Baker et al. conclude that “through the coordinated and controlled introduction of MDE techniques, significant quality and productivity gains can be consistently achieved and the issues encountered can be handled in a systematic way.”

The second answer to the question included an industry-wide study to learn by whom and where MDE is used. In this study, 450 MDE practitioners were surveyed. Interviews were also performed with these practitioners and 22 other practitioners. Whittle et al. conclude that developers barely ever user MDE to generate entire systems despite MDE being more commonly used than usually believed. Instead, developers apply MDE to key parts of a system.

2.4.4 Challenges for MDE

The penultimate topic to be covered is the challenges of MDE. Primarily, a study done by Bucchiarone et al. [BCPP20] describes findings of two events in Germany and Italy where the state of the art, research and practice were discussed. Experts from industry, academics and the open-source community gathered in these events to discuss the changes over the last ten years of MDE at the

time of the meetings (2017 and 2018). The grand challenges found were split into two groups: technical challenges and social and community challenges.

Under the technical challenges foundation challenges and domain challenges are the two main sub challenges. Under the social and community challenges social aspects and community aspects are the two main sub challenges. Bucchiarone et al. have illustrated the challenges found during the events in the chart seen in Figure 12.

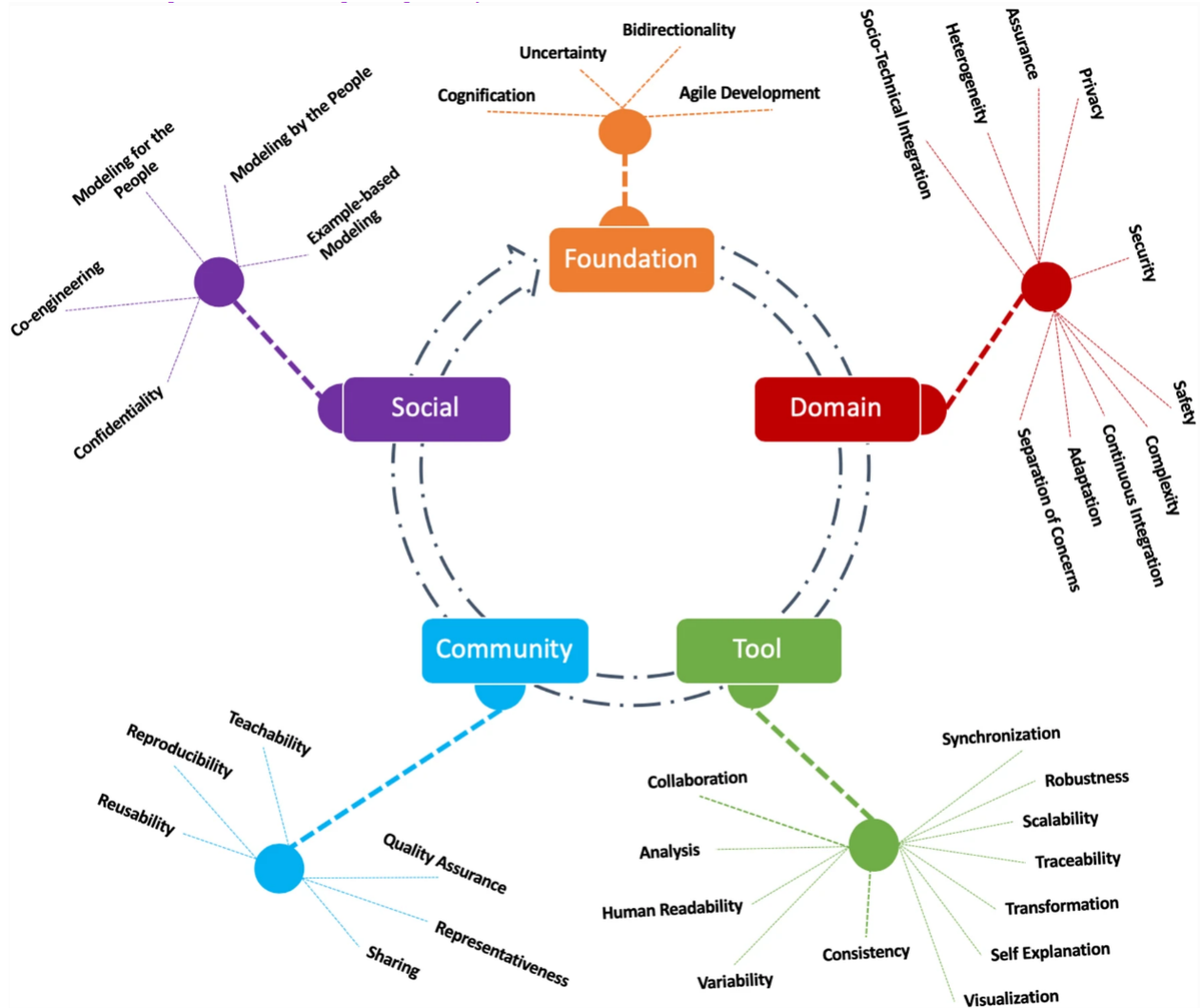


Figure 12: Challenges of MDE. Courtesy of Bucchiarone et al.

Furthermore, a study done by Mohagheghi et al. reviewing quality control in MDE might be of particular interest when analyzing challenges for MDE [MA07]. In this study, Mohagheghi et al. both lay out quality goals in MDE and emphasize that the quality of models is affected by the quality of modeling languages, tools, modeling processes, the knowledge and experience of modelers and the quality assurance techniques applied. They also state that well-formedness and precision

are key factors in MDE.

Due to quality control being such a main component MDE, Mohagheghi et al. think that more research on quality control may aid in the adoption of MDE for complex system engineering. Using this reasoning, it is clear to see that quality control is a fundamental challenge in MDE. Properly tackling this challenge may induce the promotion of adoption of MDE.

2.4.5 Usage of MDE

Finally, the question of how MDE can be used will be reviewed. The first paper that does this uses MDE for model-checking tools [BTJ⁺21]. Besnard et al. use a UML model interpreter so software requirements can directly be translated into UML. In the approach of Besnard et al., formal requirements are encoded as UML state machines. As such, it is stated that executable UML specifications are able to model “either a Büchi automaton or an observer automaton, and is synchronously composed with the system, to follow its execution during model-checking. Formal verification can continue at runtime for all deterministic observer automata used during offline verification by deploying them on real embedded systems.”

A second study displaying the usage of MDE has set out a framework for MDE [Ken02]. A distinction was made first between Model Driven Architecture (MDA) and MDE. The paper’s framework for MDE aims at creating a point of reference within MDE and MDA. The most important topics covered by this article are the organisation of the modelling space and how to locate models in that space, different kinds of mappings between models, why process and architecture are tightly connected, the importance and nature of tools, the need for defining families of languages and transformations and for developing techniques for generating/configuring tools from such definitions.

The third article discussing the usage of MDE looks at software estimation [GGN94]. Namely, Gong et al. present a software estimator which measures execution time, program memory size and data-memory size for a specification executing on a processor. The main idea of using a software estimator was to enable quicker exploring of large design spaces in software/hardware systems. Designers or partitioning tools have a trade off between hardware with software implementation for the entire part or only a part of the system under design. As such, automatic software estimation is central in this process.

The last paper tackling the usage of MDE aims to translate specifications in natural language to executable specifications [SF96]. For this, Schwitter et al. use Attempto Controlled English (ACE). ACE is a subset of natural language which is still expressive enough to be able to use for natural language, but can be accurately and efficiently processed by a computer. ACE is used to translate specifications discourse representation structures and Prolog. As a result, a knowledge base is generated which can be queried in ACE and executed for simulation, prototyping and validation of the specification.

2.4.5.1 MDE usage and prototype execution

This project slots in the MDE usage in such a way that a new fashion of MDE is studied. Namely, this project analyzes MDE usage at runtime. That is, by creating software which is able to interpret data entered into an activity diagram model. An user generates an activity diagram and creates pages after which the created pages can be displayed in the correct order according to the nodes of the activity diagram. As such, the need to manually generate and display pages is removed and executing a prototype is made more effective and simple.

3 System Design

This section will examine the design of the system created to execute prototype specifications at runtime. This will be done by dissecting the system design into two subjects. Firstly, the logical design will be reviewed. Secondly, the technical design will be discussed. The logical design subsection will analyze the logical aspects behind the code written. That is, an explanation on the design of the system will be given on the level of the code to understand how parts of the software relate to each other and cooperate. In the technical design component, the same will be done as for the logical design, but for a more high-level view of the system.

3.1 Logical Design

3.1.1 Requirements

The requirements of the logical design lie in the execution of the prototype. The most important requirements is the ability for a user to automatically go through the pages they assign nodes in an activity diagram. For example, a user may create an activity diagram with two connected action nodes and link two page to these action nodes. Then, the required functionality is that the user is able to go through the pages as indicated by the activity diagram.

Furthermore, the software must be able to handle the desired functionality for the nodes explained in section 2.1.1. However, this project does not implement functionality for the decision, fork and join nodes. One of the key requirements is that the pages which the user linked to nodes are shown in the correct order and take the user's input to execute the prototype. As such, taking the user's input from the page creation and linking should lead to the execution of the prototype. The switching between pages can be done through a "next page" button.

3.1.2 Overview

View the activity diagram given in Figure 1. A user creates pages and links them to the nodes. For example, the user assigns page 1 to the Enter Order node, page 2 to the decision node, page 3 to the approve order node and page 4 to the allocate stock node. When the user clicks start prototype and the button communicates with the backend to show page 1. When the user is finished on page 1, they can click the next page button to go the next page. Now, a decision node is reached and the user must determine whether their condition is satisfied or not.

Depending on the user's input, either page 3 is reached or the merge node. If the user reaches page 3 and they approve the order, the merge node is reached and the user reaches page 4. If the user disapproves the order the finish of prototype is reached. Else, the merge node is reached in the case of order approval or no need for validation and page 4 is shown. After allocating stock the user reaches the finish of the prototype. Due to the fact that there is a decision node, the software designed in this project is unable to handle this specific prototype.

3.2 Technical Design

The technical design consists of html files and Python files communicating with each other. Also, the entire project is built with Django. Mainly, the python files use functions calling upon each other as a result of receiving a request from a user pressing the start prototype button from Figure 32. When the request is sent from the button, a function is called to handle this request. In this function, three main functions are called. Firstly, a function called `page_node_rela()` which creates a dictionary of node ids and their corresponding page ids, where the keys are the node ids and the values the page ids.

Secondly, a function called `node_order()` is called. This function returns a dictionary of node ids and the type of the node this node is, where the keys are the ids of the nodes and the values are the types. For this, finding the initial node of the activity diagram is done first. From the initial node, the edge from the initial node is checked and the adjacent node is found. This node is added to the dictionary and now the adjacent node for this node is found. This process goes on until the final node is found.

Lastly, a function called `node_has_page()` is called. This function takes the dictionaries obtained from the `page_node_rela()` and `node_order()` functions and checks for the nodes in the `node_order` dictionary whether they have a page linked to them. If a node has no page linked to them, the node is removed from the `node_order` dictionary because there is no page which has to be shown for that node. Currently, only functionality for the action, merge, initial and activityfinal nodes have been implemented.

When all of this has been determined, the main function checks whether the request it got from the html was get or post. If the request was get it means that the start button was pressed and it shows the first page by looking up the first node in the `node_order` dictionary. If the request was post the user has clicked the next page button. The index of which page is looked up in the `node_order` is increased by one and the next will be shown by resetting the request method to get.

4 Worked Examples

To illustrate how the system works, two worked examples will be shown. First, worked example created by Bram van Aggelen of the creation of a page will be viewed. Then, an illustration of the workflow will be depicted. As a result, a clear understanding of the created software should be obtained.

4.0.1 Page creation

First view the worked example for the creation of pages. Remember that users start by entering their requirement text into the backend metadata extractor (see Figure 8). Consequently, users can create a class diagram from the metadata (see Figure 9). However, the class diagram which is automatically created without human intervention looks like the one seen in Figure 13. This diagram does not necessarily guarantee that the final application will work in the way it is supposed to. As such a few changes need to be made.

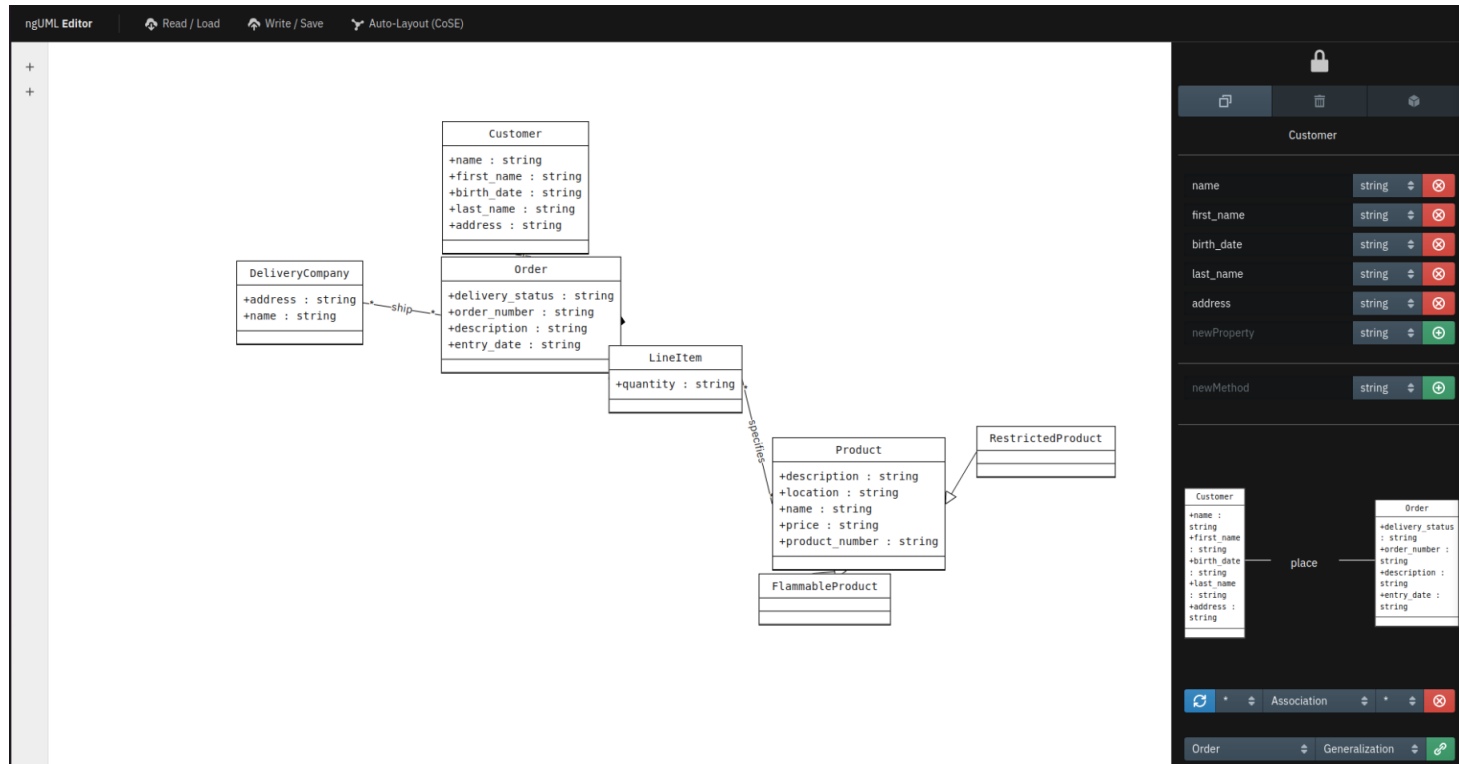


Figure 13: Class diagram directly created from metadata extraction

Namely, some relationships which are now modelled as many to many need to be modelled as many to one, due to the lack of implementation of many to many relationships currently. As such these changes must be made where the 1 in the changes showcases the 1 in the relationships:

- order – customer (1)
- order – deliverycompany (1)
- lineitem – product (1)

Now, view the backend again. Here, the relations and classes can be viewed and edited too (see Figure 10). The most important part in the backend to create pages, is the applications part. For this example, two applications were created. One application was created for a client and one

application was created for a business (see Figure 14). After the creation of the applications, the corresponding classifiers were added to the applications. In the applications, categories were created. Under the business application the category creation was created (see Figure 15) and 2 pages called product and delivery company were made (see Figure 16) in this category.

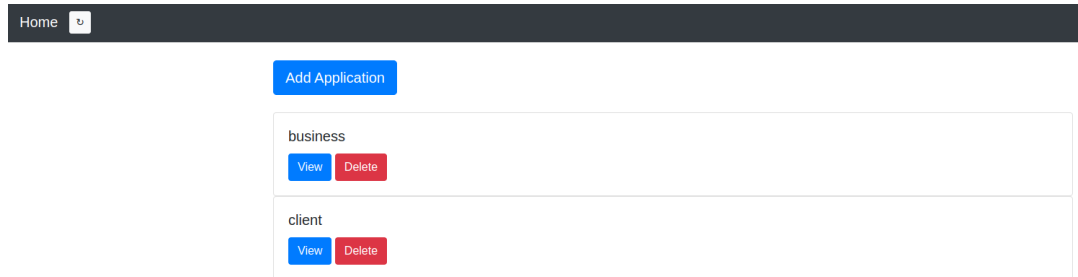


Figure 14: The business and client applications

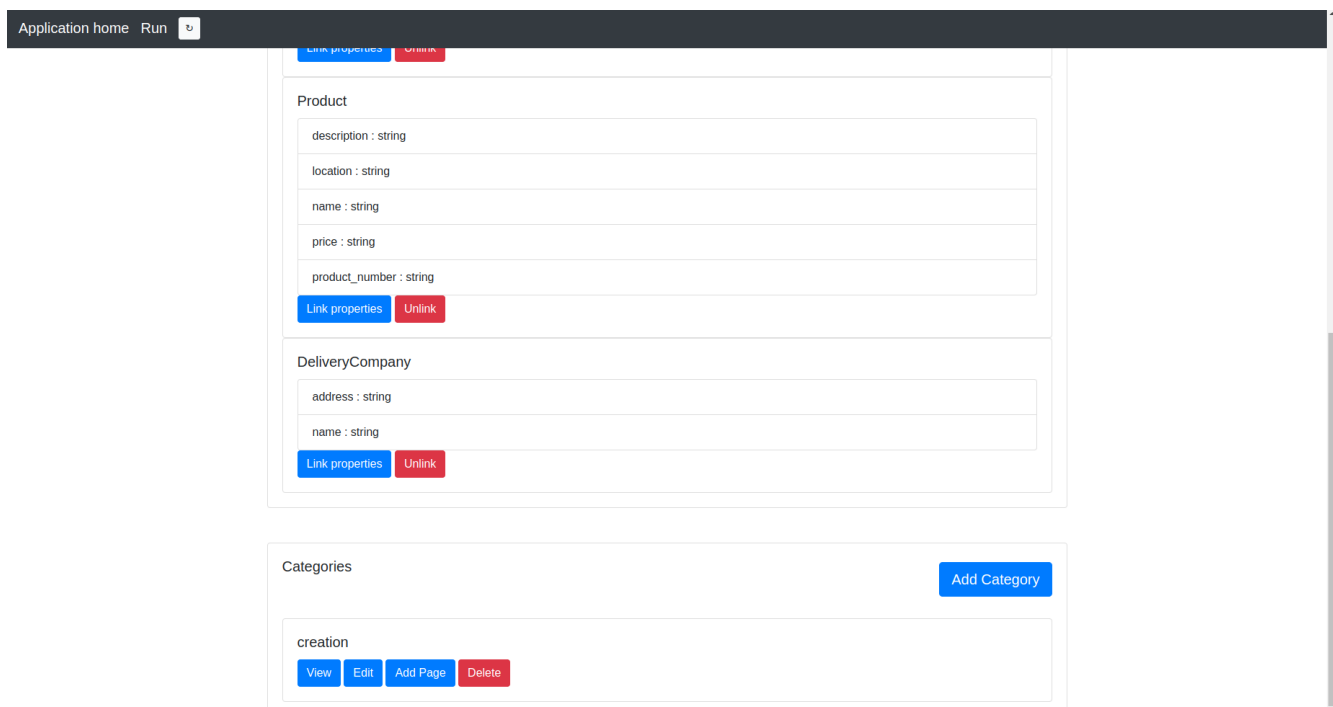


Figure 15: Category “creation” for the business application

Now that the pages have been created, the query tab from Figure 16 was accessed. In the query tab, the main model can be chosen, the toggle for “page for data insert” can be checked and a query can be specified to narrow down results (see Figure 17).

For this page, product was selected and the page creation option was chosen. After this was saved, the properties tab from Figure 16 was selected. In here the properties of product were used (see

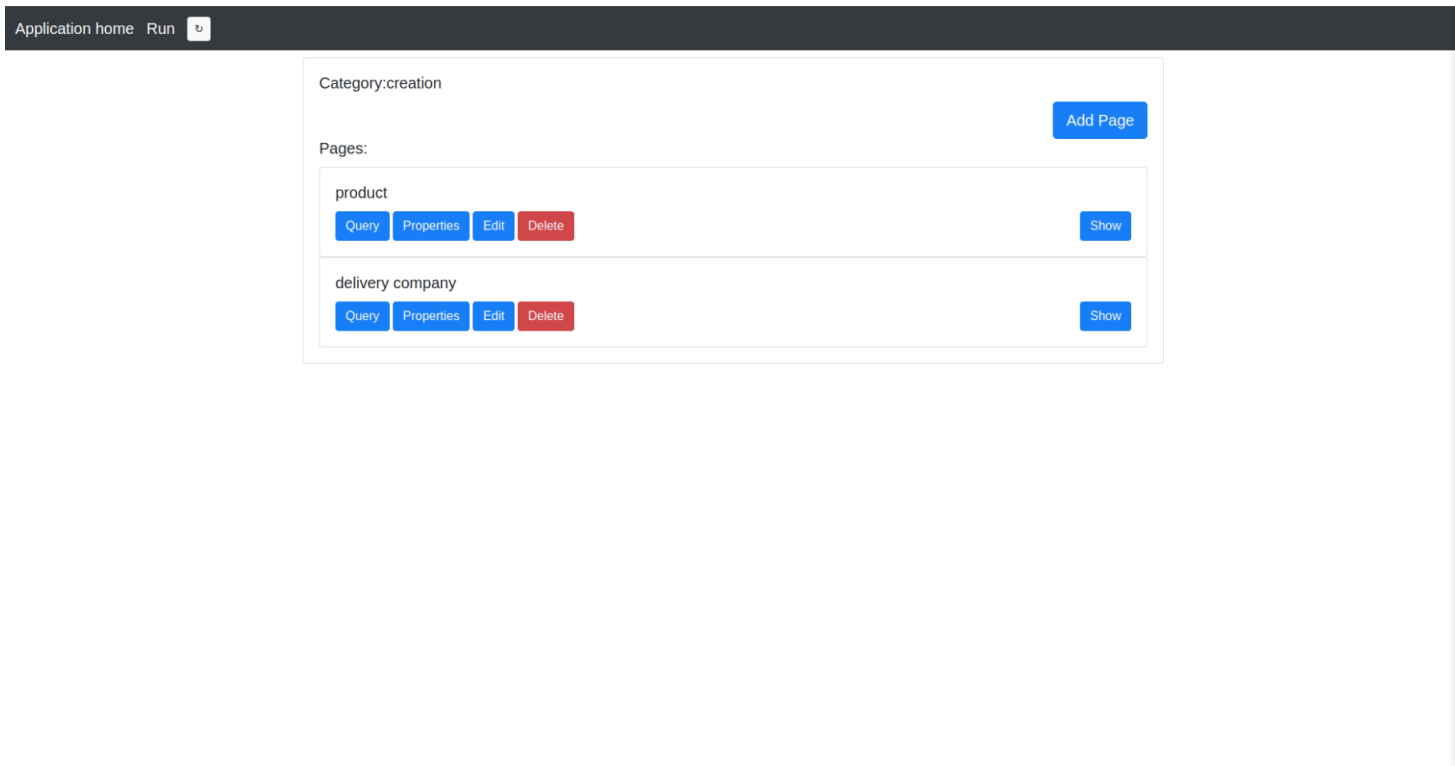


Figure 16: Product and delivery company pages created for the business application

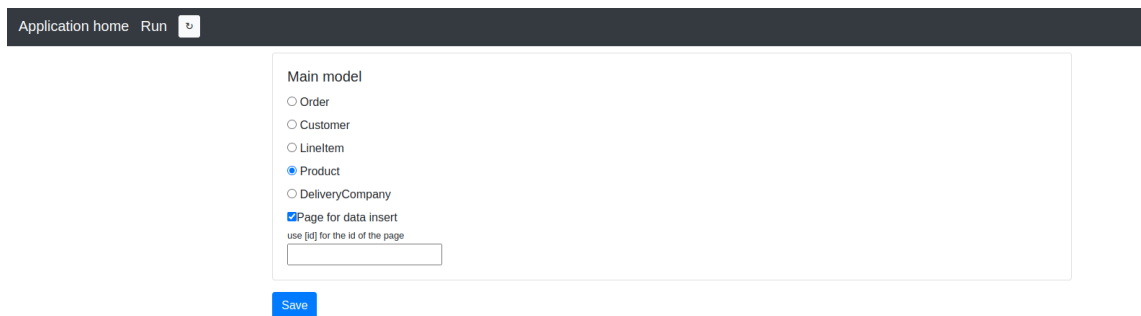


Figure 17: Query page

Figure 18).

Next, the html structure of the page was created through the edit tab of Figure 16. On this page (see Figure 19) there is the ability to insert data, insert a data table, insert an input field, insert an input model and insert a submit button.

For this example, insert input was chosen (see Figure 20) and the data of product was selected (see Figure 21).

Afterwards a save button was added to the page resulting in the final page seen in Figure 22. The

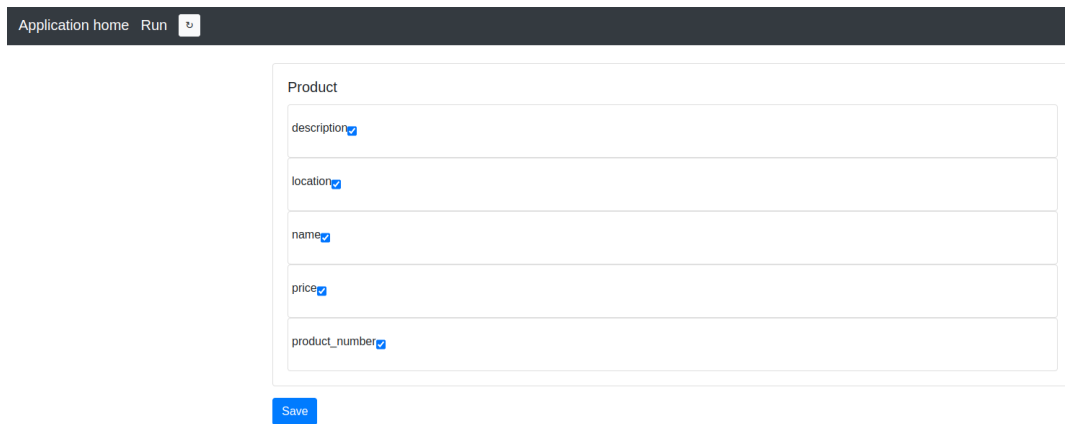


Figure 18: Properties page

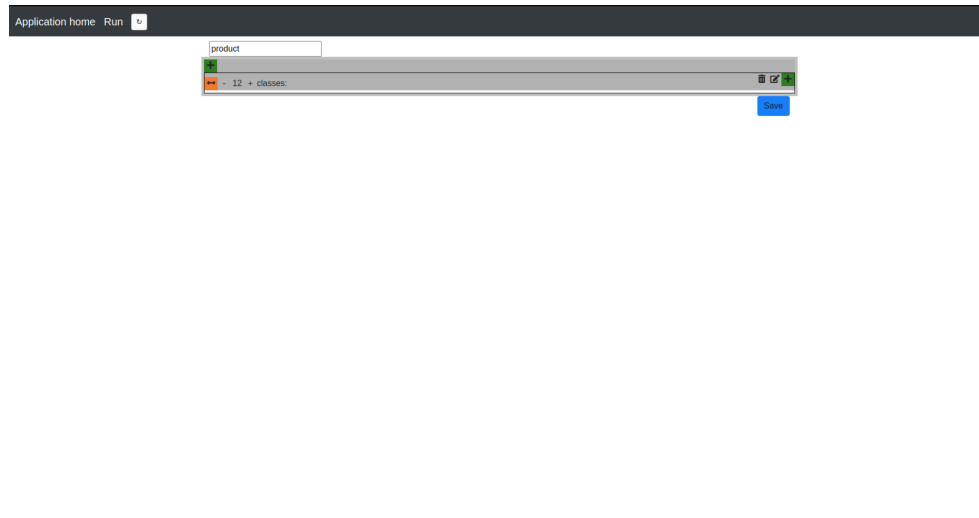


Figure 19: HTML page creator

add more button may be used in case there is need for more data to be entered. Remember, this page was created for the product (as seen in Figure 16). As such, the same steps were done for the delivery company afterwards. Next, a new category called overview was created (see Figure 23). In this category the order-overview page was created. The main model of this page was Order. The following properties were selected:

- All of Order
- Name of Customer
- Quantity from LineItem
- Name, price and product_number from Product

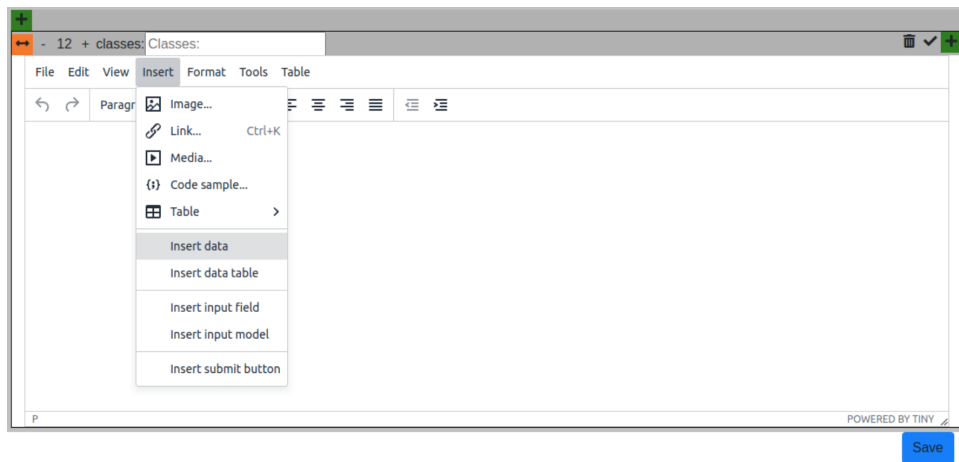


Figure 20: Inserting data in the HTML page creator

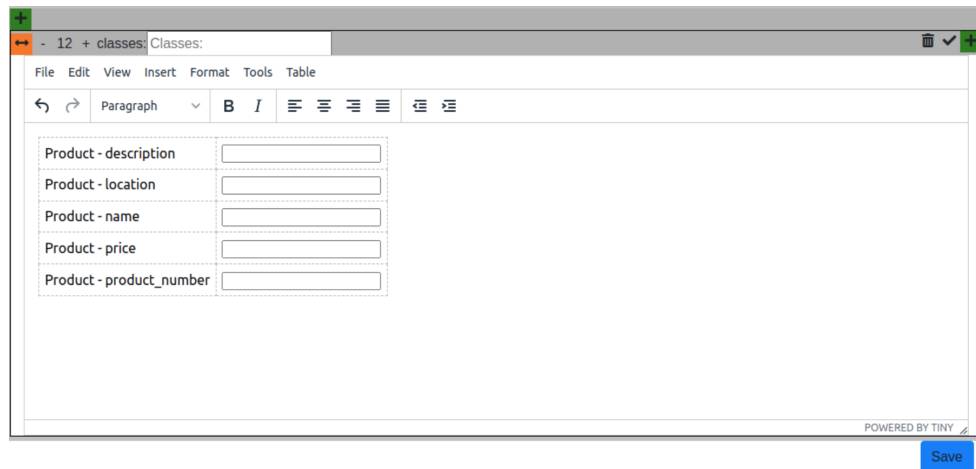


Figure 21: Inserting the product data in the HTML page creator

- All of Delivery Company

As such, the page in Figure 24 was created. Lastly, the Client application was accessed and the Order, Customer, LineItem and Product classifiers were added. Also, the category Order was made. In this Order category, a page create-order was made. For the main model, the Order classifier was used and inserted as data page. All possible properties were linked and the final page was created as seen in Figure 25.

4.0.2 Workflow example 1

Next, view an example of the workflow. For the workflow, the activity diagram from Figure 26 was created in the editor.

As visible, one initial node was created after which two consecutive action nodes come. The first thing to do in the editor is to create page names for the nodes of the activity diagrams. By assigning

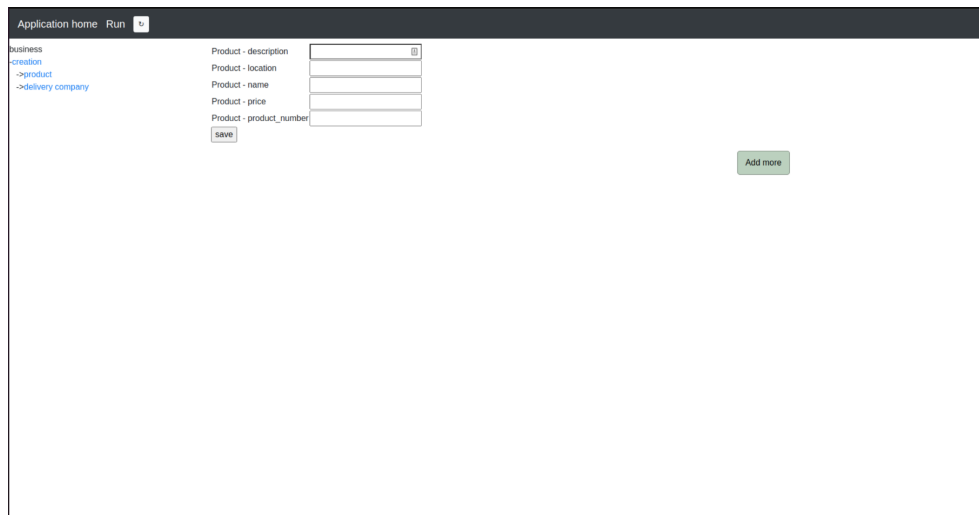


Figure 22: The page created after selecting the product data and creating a save button

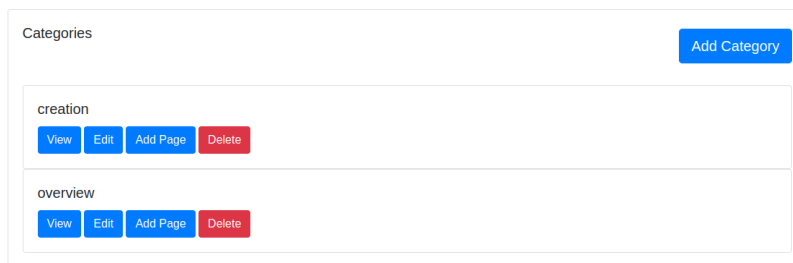


Figure 23: Addition of the category for overview

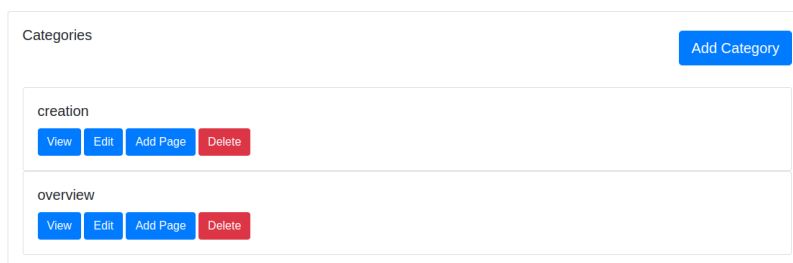


Figure 24: Page for the order overview

a page name to nodes, nodes can later be linked to pages created as seen in example of chapter 4.0.1. In Figure 27 and 28 page names were assigned to the nodes.

Following, a category called cat1 was created (see Figure 29) and two pages were created within this category. Next, the page1 and page2 were linked to pagename1 and pagename2 respectively (see Figure 30 and Figure 31). As such, action node 1 and page 1 are effectively linked to each other as well as action node 2 and page 2.

In order to run the prototype, the run prototype button (see Figure 29) must be clicked leading the

Figure 25: Creation of the create-order page



Figure 26: The nodes connected by edges

user to a page with a button to start the prototype execution (see Figure 32). As a result, the page linked to the first action node after the initial node will be shown. Since action node 1 was linked to pagename1 and pagename1 to page 1, page 1 will be shown for action node 1 (see Figure 33). When the next button is clicked, the page corresponding to action node 2 will be displayed. Since both the pages for action node 1 and 2 are empty, the pages will look like Figure 33. However, the id of the page is still different for both nodes. If a page has some data entered onto it, the difference between the pages can easily be spotted.

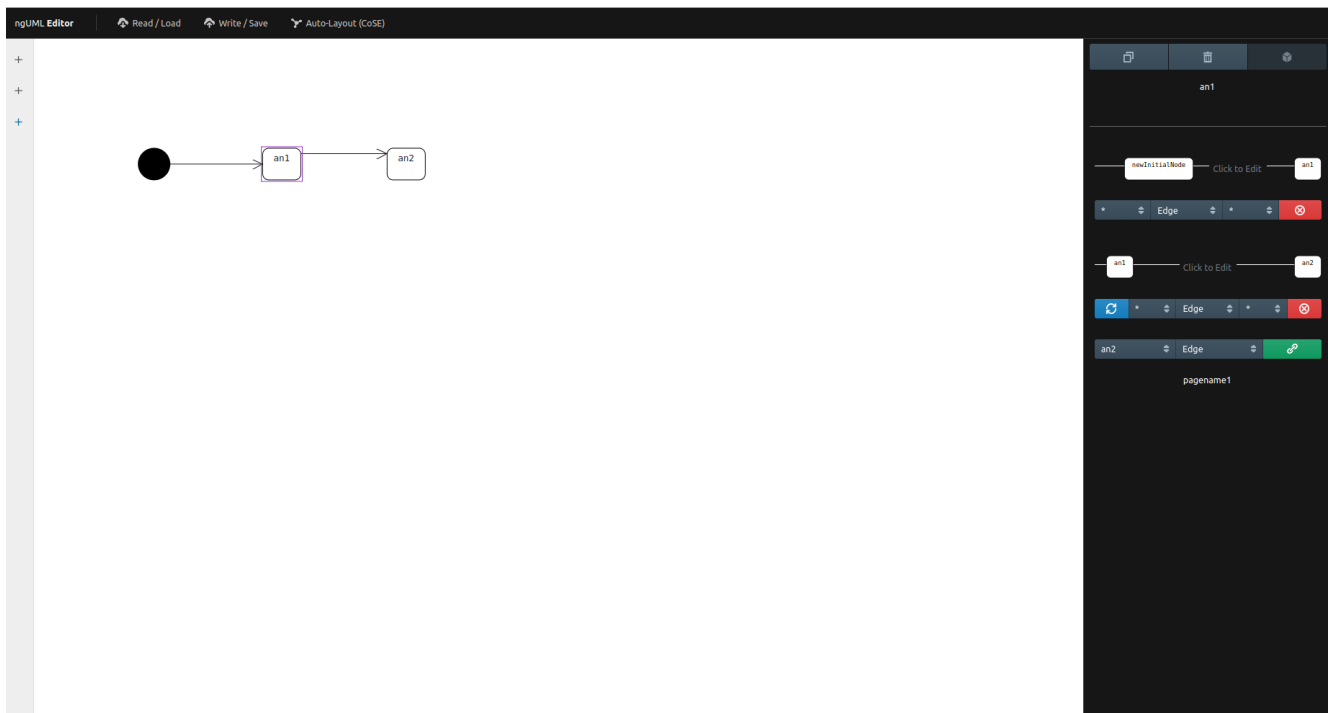


Figure 27: Addition of a page name for action node 1

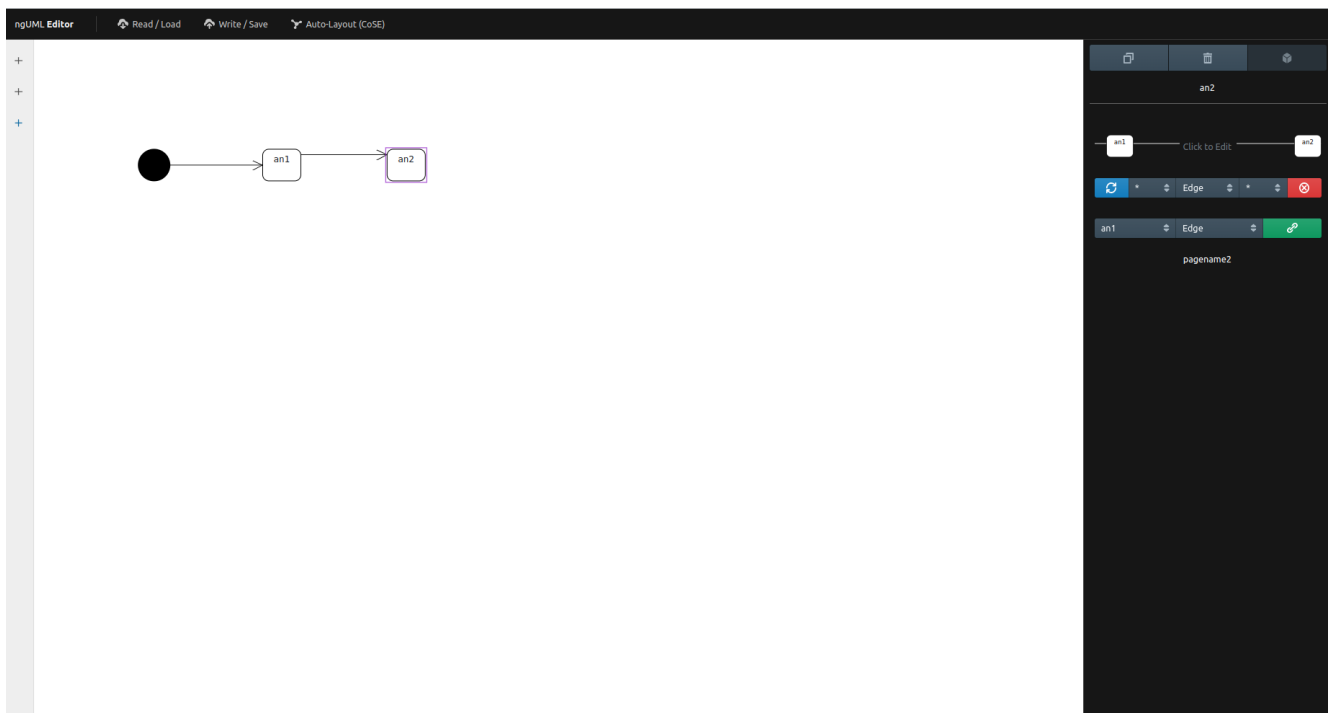


Figure 28: Addition of a page name for action node 2

Application home Run Next **Next page** 1

name : string

price : string

product_number : string

Link properties Unlink

DeliveryCompany

address : string

name : string

Link properties Unlink

Categories

Add Category

cat1

page1

page2

View Edit Add Page Delete

Prototype

Run Prototype

Figure 29: Addition of a category

Application home Run Next page 1

page1

Page Name

▼

 Save Back

pagename2


pagename1

Figure 30: Linking page 1 to pagename1

Application home Run Next page 

page2
Page Name

Figure 31: Linking page 2 to pagename2

Application home Run Next 

Prototype

Figure 32: Page to start the prototype execution

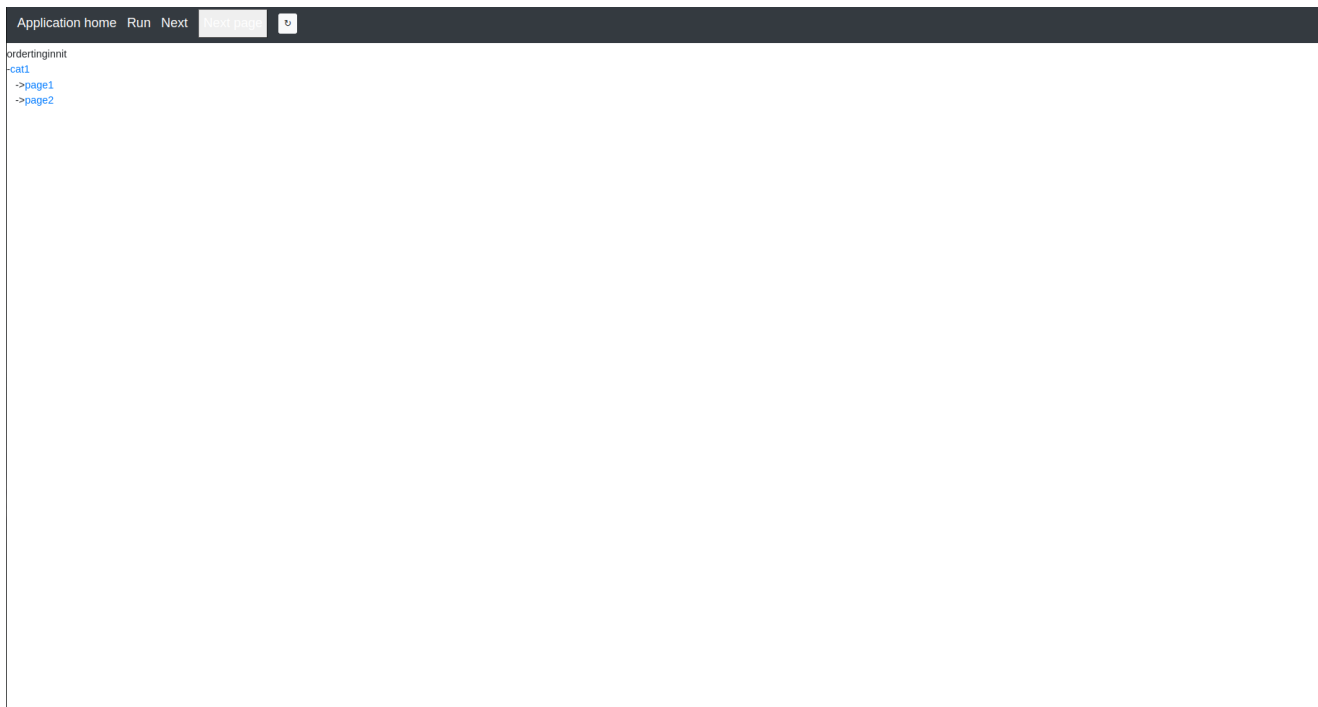


Figure 33: Page which is shown for the first action node. Note, since the page is empty the same page will be displayed for action node 2.

4.0.3 Workflow example 2

For the second example of the workflow, assume the same activity diagram as example 1 (see Figure 26). However, now page 1 must take the properties of the Order model and let the user fill in the data for these properties. The creation of this page can be seen in Figure 34. The second page must show the text “page number 2” and a submit. This page is created in the way seen in Figure 35. When going back to the start prototype execution page and clicking the start button the first page linked to the first action node will be displayed (see Figure 36). On this page, users are able to enter data for the properties of the Order model. Clicking the next page button sends a user to the second page (see Figure 37). Here, the text “page number 2” and the submit button are present.

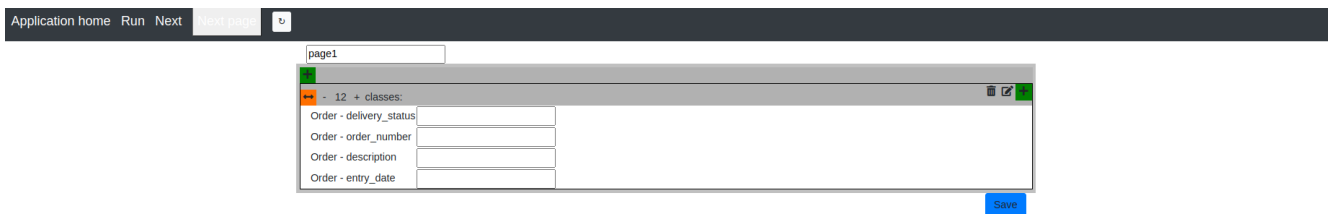


Figure 34: Page which must be shown for the first action node. The page must show data entry fields for the properties of the order model.

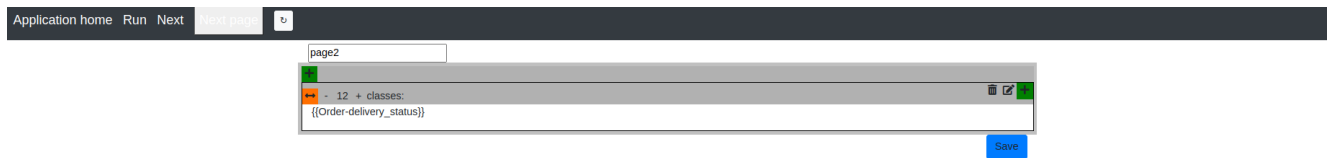


Figure 35: Page which must be shown for the second action node. This page must show the text “page number 2” and a submit button.



Figure 36: Page which is shown for the first action node. The page shows data entry fields for the properties of the order model which can be filled in by the user.

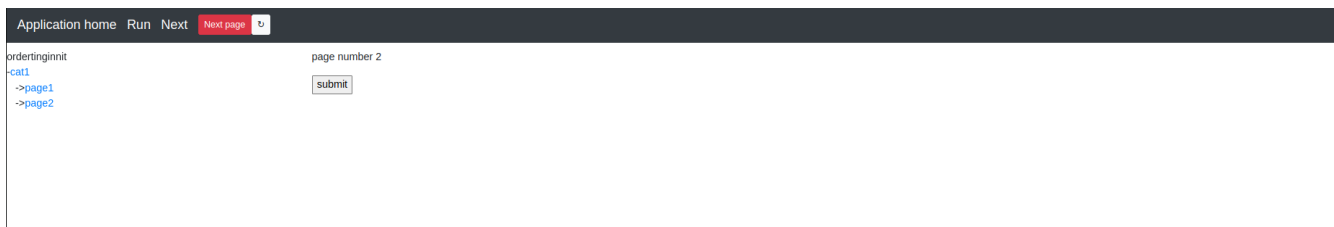


Figure 37: Page which is shown for the second action node. The page shows the text “page number 2” and a submit button.

5 Conclusion

This paper is part of ngUML at the Leiden Institute of Advanced Computer Science and looked at adding behavioural aspects to static pages in an UML interpreter to automatically execute prototypes at run time. Whereas previous research has already looked at the rationale, industries, users, challenges and usage of Model Driven Engineering and Executable Specifications (MDE) this paper adds to this by creating a run time prototype executor from metadata obtained by user requirements. The way this was done was by creating Python functions calling each other to showcase pages at run time.

This was done by extending a current existing project by the ngUML team existing of an editor and a backend. In the editor users are able to enter activity diagrams or class diagrams. This data is then sent to the backend and users can create pages. However, the activity diagrams were not able to be performed automatically yet. As such, Xue Jun Wang’s paper and this paper collaborated to

create the ability to execute prototypes. Whereas Wang’s work focused on linking pages to nodes of an activity diagram, this paper worked on the automatic execution of the flow of pages for an activity diagram.

Since the existing project was coded using Django, the software delivered in this paper consists of html files communicating with Python functions. The prototype execution was done by using three main functions. The first function determines all nodes in an activity diagram and the pages linked to them. The second function returns the order of the nodes. The third function removes all nodes without pages linked to them. Lastly, the first page is shown and if the user clicks the next page button the index of the node_order dictionary is increased by one and the next page is shown.

Nevertheless, the prototype execution does not have full functionality yet. Namely, functionality for decision, fork and join nodes have not been implemented yet. As such, future research should focus on adding the functionality for these nodes. Another limitation of this paper’s research was the lack of proper testing. Even though all errors have been removed, more potential options for prototypes should be tested to see whether they break the code. Furthermore, future research should focus on making cleaner code. For example, the current code takes all activity nodes in the database without looking at the id of the activity diagram they are part of.

Another problem with the execution engine is the inability to interpret the data on edges. For example, if an edge between nodes says “check for completeness” the user has to decide whether they want to move on the next page manually. The user clicks on the next page button so they have to check whether their condition is satisfied themselves. Future research may enable automatic interpretation. The last limitation is the lack of response when an activity diagram is incomplete for example. Future research should return a page which says that an user’s activity diagram is incorrect if it is missing edges for example.

References

- [act] Uml activity diagram tutorial.
- [BCPP20] Antonio Bucchiarone, Jordi Cabot, Richard Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19:1–9, 01 2020.
- [BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context — motorola case study. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, pages 476–491, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [BTJ⁺21] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. Unified verification and monitoring of executable uml specifications: A transformation-free approach. *Softw. Syst. Model.*, 20(6):1825–1855, dec 2021.
- [GGN94] Jie Gong, Daniel D. Gajski, and Sanjiv Narayan. Software estimation from executable specifications. *J. Comput. Softw. Eng.*, 2(3):239–258, mar 1994.

- [Ken02] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 286–298, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- [MA07] Parastoo Mohagheghi and Jan Aagedal. Evaluating quality in model-driven engineering. In *International Workshop on Modeling in Software Engineering (MISE’07: ICSE Workshop 2007)*, pages 6–6, 2007.
- [MAB⁺14] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. The relevance of model-driven engineering thirty years from now. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 183–200, Cham, 2014. Springer International Publishing.
- [MGS⁺11] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel Angel Fernández, Bjørn Nordmoen, and Mathias Fritzsche. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling*, 12:619–639, 2011.
- [MGSF13] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel Fernández. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18:89–116, 02 2013.
- [SF96] Rolf Schwitter and Norbert E. Fuchs. Attempto - from specifications in controlled natural language towards executable specifications, 1996.
- [WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.