Universiteit Leiden
The Netherlands

Opleiding Informatica & Economie

Generating Workflow enabled Prototypes from UML Activity models for Model-Driven Engineering

Maarten Verhoeff (s3350290)

First supervisor: Dr. G.J. Ramackers
Second supervisor: Prof.dr.ir. J.M.W. Visser

BACHELOR THESIS

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

July 14, 2025

Acknowledgments

I wish to express my gratitude to every individual whose contributions made this thesis possible. I owe special thanks to Dr. G.J. Ramackers and Prof.dr.ir. J.M.W. Visser for the many productive meetings we had, which greatly contributed to the direction of my thesis. I would also like to thank S.Y. van Dam for sharing his knowledge as technical architect of the AI4MDE project.

Generative AI usage

Generative AI was used during this thesis solely for rephrasing existing text to improve clarity and the flow of sentences. It was not employed to generate any new content. An estimate of 5% of the total text was revised using generative AI.

Abstract

Model-Driven engineering seeks to create and utilize models to automate and streamline software development. A persistent challenge for both academic tools as well as commercial low-code platforms is the automated generation of executable workflows from UML activity diagrams. To address this, we designed a solution that maps the metadata from a UML activity model to a data-driven executable format. The feasibility of this approach is verified by implementing it within the *Artificial Intelligence for Model-Driven Engineering* (AI4MDE) project. This project focuses on generating Django prototypes for data manipulation-focuses applications. Following the implementation, the generated application accurately reflected both the structural and behavioral aspects modeled in the UML diagrams. The generated data-driven workflow-aware application can guide different users through sequential processes with decisions, loops and parallel sections, and enable the system to execute automated tasks using custom code implemented by the user. User interviews confirmed the effectiveness of this approach, while also highlighting several opportunities for further enhancement.

Contents

1 Introduction

1.1 Context and problem statement

The Unified Modeling Language (UML) is a widely adopted standard in software development for specifying, visualizing and documenting the structure and behavior of software systems. Numerous efforts have been taken to not only use UML for documentation purposes, but also to directly generate applications from the UML, thereby bridging the gap between design and implementation. This approach is highly desirable as it can significantly reduce development time by minimizing the need for manual coding. In turn, this can lower overall development cost and enable organizations to adapt more quickly to changing market conditions. The research project *Artificial Intelligence for Model Driven Engineering* (AI4MDE) at Leiden Institute of Advanced Computer Science is one of these projects aiming to achieve this. It does this by providing the user with an intuitive design studio to create UML-based models, from which the metadata is used to automatically generate an application. The AI4MDE project is currently able to generate simple prototypes with multiple pages on which a users can perform Create/Read/Update/Delete operations. However, a significant limitation of the current implementation is its inability to generate workflow-aware prototypes due to the absence of the UML activity diagram in the generation process. This research aims to expand the AI4MDE project by extending the generation architecture to allow it to generate workflow-aware applications.

1.2 Research objectives

This research is an extension to the generation process, created by van Dam [vD24], Willemsens [Wil23] and de Hoogd [dH24]. This process makes use of the class and use case diagram from the Unified Modeling Language in combination with jinja2, a templating langauge, to generate a runnable prototype. Application components are also added by the user containing pages, sections and categories, allowing for a fronted to be generated to interact with the application. A detailed explanation of this work can be found in Section 2.4. A limited activity diagram editor is also implemented in the project, however this metadata is ignored in the generation process. To be able to make workflow-aware applications the metadata of this diagram will have to be mapped to a format which can be executed. This results in the first research question:

Q1: How can a mapping be defined from UML activity models to a low code execution platform?

The second research question investigates how the mapping from the UML activity models into an executable format can be integrated into the AI4MDE project. This leads to the following research question:

Q2: How can such a mapping be implemented by extending the prototype generator in the LIACS AI4MDE project?

To address this, the following issues will have to be investigated

- How can the current activity diagram editor be extended to include the metadata required for the enhanced generation process?

- How can the UML activity diagram metadata be parsed into the executable format mentioned in research question 1
- How can the generated user interface of prototypes be extended to expose workflow functionality to the end user in a user friendly fashion?

1.3 Overview

This research will first discuss the background information relevant to this research. This includes information about model driven development, UML and workflow engines. It will also provide an in-depth examination of previous work on this project. Afterwards, the methodology employed in this research will be explained. Following this, the conceptual architecture will present the proposed solution which will be implemented in the technical architecture. The new generation process will be evaluated using three different case studies, one of which will involve real users to assess the effectiveness of the solution in an real-world scenario. The research will conclude with a summary of the findings and a discussion of potential future extensions to the project.

2 Background & Related work

2.1 Model Driven Development

Model-Driven Development (MDD) refers to "the notion that we can construct a model of a system that we can then transform into the real thing" [MCF03]. One well-known realization of this approach is the *Model-Driven Architecture* (MDA), a framework proposed by the Object Management Group (OMG). OMG defines the goal of MDA as follows: "to derive value from models and modeling that helps us deal wit the complexity and interdependence of complex systems" [Gro14]. This is done by introducing a three layered modeling approach. First a *Computation Independent Model* (CIM) is made, which focuses on describing business concepts. After this, a *Platform Independent Model* (PIM) is created, capturing the systems structure and behaviour without committing to any specific technology. Finally the PIM model is used to create a *Platform Specific Model* (PSM) by refining it with technology-specific details. Advantages mentioned with using MDA include a faster development process, easier updates, greater consistency and improved stakeholder understanding.

2.2 Unified Modeling Language

The *Unified Modeling Language* (UML) is a modeling language used to visually represent the structure and behavior of a software system during the development process. UML was adopted by the Object Management Group in 1997 and has been maintained by them since, with the latest version being 2.5.1. According to official documentation [Gro17], UML consists of 14 different diagrams, divided into two main categories: 7 structural diagrams, which describe the static structure of a system, and 7 behavioral diagrams which capture dynamic behavior and interactions. The Class and Use Case diagram, which are already used during the generation of a prototype in the AI4MDE project, are examples of a structural diagram. The Activity diagram, which will be used to extend the AI4MDE project, is an example of a behavioral diagram.

2.2.1 Activity diagram

As mentioned before the Activity Diagram is a behavioral diagram within UML with its main purpose being to represent workflows within a system. It models the sequence of activities and the transitions between them [Gro17]. The main element within an activity diagram is the Action node. It represents a task or operation within the system. Between these action nodes, control flows define the sequence the action nodes are executed. This flow can be augmented using different control nodes such as decision, merge, fork and join nodes, which control branching, combining and parallel execution of the workflow. Additionally Swimlanes are used to organize the actions between different groups, typically representing different actors in the process.

2.2.2 UML in software generation

UML is mostly used as a visual aid in the design of software. Numerous efforts have been undertaken to automate this process by generating applications using these models. One of these efforts is the approach presented in the paper *Generating Java code from UML class and sequence diagrams*, which explores automating the generation fo Java code base on UML class and sequence diagrams.[PSDB11]. Even though this approach was successful in generating some code, it was unable to produce complete code due to the limitations of the sequence diagram, which focused on method invocation rather than the full implementation needed for comprehensive code generation.

Similarly, another method is described in the paper *An approach to code generation from UML diagrams*, which explains a method to convert a single UML diagram in to executable code [GM14]. This method however, once again, fails to generate complete code due to the limitation to one single UML diagram.

A common factor in both approaches is their reliance on a limited amount of UML diagrams, which prevents the generation of fully functional code. This limitation is further confirmed in the paper *Automatic code generation from UML diagrams: the state-of-the-art* by M.Mukhtar and B. Galadanci [MG18], which analyzes 40 different automatic code generation methods based on UML. Their main conclusion highlights a significant gap in the automatic transformation of UML diagrams into source code. One of the primary reasons is that a large majority of the papers focuses solely on generating source code from a single UML diagram, which as previously nodted, is insufficient for producing complete, executable code. The paper also noted that a significant amount of these approaches focused on generating Java code.

2.3 Workflow engines

According to the Workflow Management Coalition (WfMC), a workflow engine is:

A software service or "engine" that provides the run time execution environment for a workflow instance.

In their Workflow Reference Model, the WfMC outlines that such software is responsible for a range of tasks, including managing process states (such as starting, suspending or terminating instances), navigating between workflow activities (both sequential and parallel) and maintaining workflow-relevant data through execution [HH95].

Within the Django ecosystem, which the AI4MDE project utilizes, several workflow-related packages exists, such as `django-viewflow`, `django-joeflow` and `django-fsm-2`. These tools manage process flows and task transitions but are tightly coupled to model definitions, meaning that modifying workflows requires changes to the code. This would limit the flexibility of the generated prototype in environment where processes may change frequently. By storing the workflow in the database, the system becomes data-driven, allowing for easier adjustment to workflows, without altering the codebase. This approach enhances adaptability and enables modifications without the need for a redeployment. It today's fast changing business environments, organizations need system that enhance the dynamic capabilities of the firm. This is a concept which refers to the first ability to integrate, build and reconfigure internal and external competences to address rapidly changing environments [TPS97]. By making the workflow engine data-driven, the system supports these dynamic capabilities, making it better suited for real-world scenarios where processed evolve over time.

2.4 AI4MDE

2.4.1 Earlier implementation

One of the earlier implementations is the What You See Is What You Get editor, implemented by B. van Aggelen [vA22]. Using this editor the user was able create pages with input fields corresponding to the class attributes. This implementation however had a few shortcomings. The first one was the large amount of work required to create the pages, since each page had to be created manually. This manual effort significantly undermined the core objective of the AI4MDE project, of being able to automatically generate applications with little to no coding experience and the second one was not being able to generate the views and URL required to operate these pages.

2.4.2 User interface

To resolve these issues S.Van Dam [vD24], L. Willemsens [Wil23] and P. de Hoogd [dH24] added three new attributes to the metadata used during the generation of the prototype: categories, pages and section components. A section component is used to perform Create, Read, Update, and Delete (CRUD) operations on a specific model. These section compoents are displayed within the second newly introduce structure, the page. Each page, in turn, is linked to a category, which serves to organize multiple pages under a common grouping [dH24].

2.4.3 Template driven generation

Using Jinja2 templates the newly provided metadata is mapped to the desired Django output files following the Model-View-Template (MVT) pattern. The class models created by the user are mapped to a single models.py file in the *shared models* application to avoid duplication. For each application component one or multiple custom views are created. These views provide the functionality required by the section components as well as being able to render the pages on which the section components are displayed. HTML templates and styling are also creating using Jinja2, enabling automated UI generation. This approach significantly reduced the amount of work required to generate an application [vD24].

After these additions the AI4MDE project is capable of generating prototypes pages on which the user can perform CRUD operations, without too much work and technical expertise.

3 Research Methodology

This section outlines the approach used to develop a prototype generator capable of generating workflow-aware applications using UML activity diagrams. The research combines research by design and iterative development, structured around three progressively complex case studies.

3.1 Research by design

This research adopts a research by design methodology, in which the design process itself serves as a core method of inquiry. As defined by the EAAE research committee, research by design involves: "any kind of inquiry in which design is a substantial part of the research process", allowing new insights and knowledge to emerge through the act of designing [Hau11]. In this thesis the extension of the prototype generator serves as both the research process and the research object. The EAAE research committee also defines research by design as being validated through peer review by panels of experts, who collectively cover a range of disciplinary competencies addressed by the work. This will be done through iterative development as well as a test with different users, which is explained in the Section 3.2.

3.2 Iterative development

This research also employs an iterative development methodology, where the design of the prototype generator evolves through a series of incremental cycles. Each cycle begins with a basic use case, gradually increasing in complexity in each iteration. The process starts with a simple UML diagram that only includes control flows between activity nodes, gradually introducing more complex elements such as decisions and parallelism each iteration. Through each iteration, the system is refined based on feedback from the result of the previous cycle, making the generator more complete over time. The final iteration concludes with a test involving different users to evaluate the generator's functionality, usability and effectiveness in real-world scenarios.

3.3 Case studies

This research uses multiple case studies, serving as integration tests, to evaluate the generator's capability to produce workflow-aware applications from UML activity diagrams. These tests examine the generator as whole, including its interaction with the existing generation architecture, ensuring that the generator functions effectively. Each case starts with designing and preparing a UML activity diagram that defines the scope of the scenario under investigation. Feedback sessions with my academic supervisor and the project's technical architect serve as important qualitative input to refine and further extend the system. The analysis compares the intended behavior specified in the UML diagrams with the behavior observed in the generated prototype. Both case studies will focus on the same scenario: the process of placing an order at an online store. The case studies will involve three different actors, each responsible for completing different activities. These actors are: Customer, Manager and Order Picker.

3.3.1 Case study 1: Basic order fulfillment

The first case study focuses on establishing a basic control flow, where activities transitions from one to another, without the inclusion of parallelism or decision making. It starts with a customer placing an order, which is then approved by the manager. After approval the picked by the order picker, checked and finally shipped. Since this will be the first case study, emphasis will also be placed on integrating the workflow engine into the front-end of the generated prototype. The UML activity diagram containing this scenario can be found in Figure 13.

3.3.2 Case study 2: Conditional and parallel order fulfillment

The second case study will introduce decision making into the workflow, meaning the flow from one activity to another will depend on specific conditions or choices made during the process. Next to this parallel flows will be implemented. The main difficulty will be managing multiple activities that can be active simultaneously. This case study extends the basic workflow from Case study 1: when the order is not approved, a cancellation email is sent, terminating the process. If approved, the picking of the order is split between different tasks, such as creating the shipment, which are executed in parallel. Additionally if the order check fails the process loops back to the start of the order picking process. Two different UML activity diagrams were used for this case study, which can be found in Figure 17 & 21.

3.4 User interviews

In the final phase of this research, user interviews were conducted to evaluate the performance of the extended prototype generator. Participants were asked to create an application of their choice using the design studio, after which the generated prototype was evaluated. They were then interviewed about their experience with the design phase, the alignment between their intended models and the generated prototype as well as the overall practicality of using the tool in real-world scenarios.

4 Conceptual architecture

4.1 Workflow engine architecture

To be able to generate workflow aware applications a workflow engine will have to be created, which the application can interact with. The prototype should be able to trigger the workflow engine to start a new process or advance the progression of ongoing activities. Next to this the application will be able to read which processes are active as well as their progress. The components diagram in Figure 1 displays how the workflow engine will be integrated in the existing AI4MDE project. The interaction between the two subsystems depicted in the components diagram are further explained in section 4.1.2. The creation of the workflow as well as its data which it depends on is explained in section 4.1.1

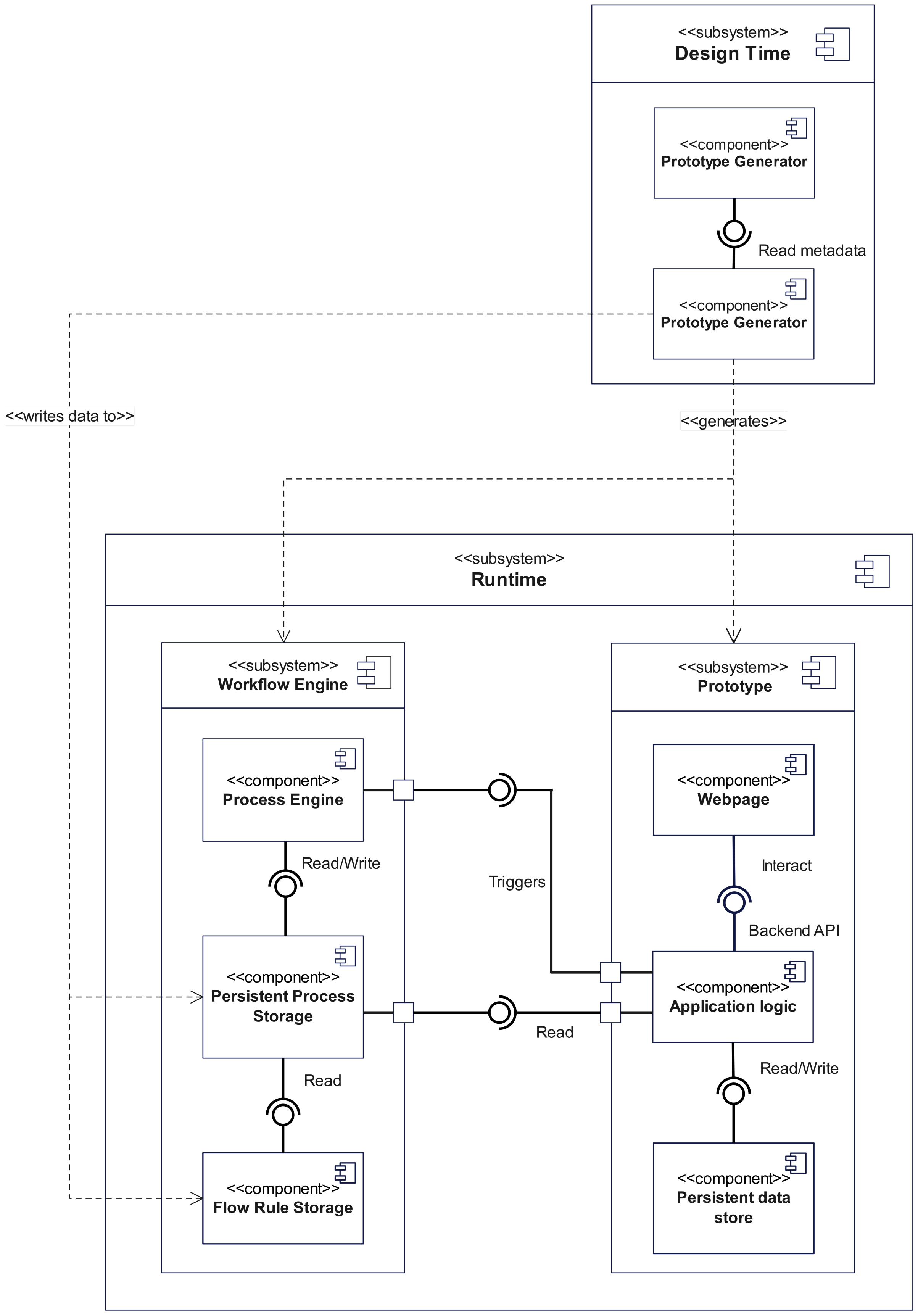


Figure 1: AI4MDE component diagram

As can be seen in Figure 1 the workflow engine relies on persistent process storage and a flow rule storage to store and keep track of processes. The metadata from the AI4MDE studio is not used directly, instead the metadata is taken and mapped to a new format as can be seen in Figure 2. This is further explained in subsection 4.1.1. Next to the reason mentioned in section 2.3 other benefits include:

1. **Business Insight**: The new format allows us to capture data from running and completed processes, enabling real-time monitoring and analysis. Processes can in turn be optimized using this data, ensuring better efficiency and improved decision-making.

2. **Runtime efficiency**: When using the raw metadata within the workflow engine it would have to be evaluated entirely after each step in the process. The new structure moves this to the generation phase and allows us to focus only on the relevant data at each step, thereby speeding up workflow execution.

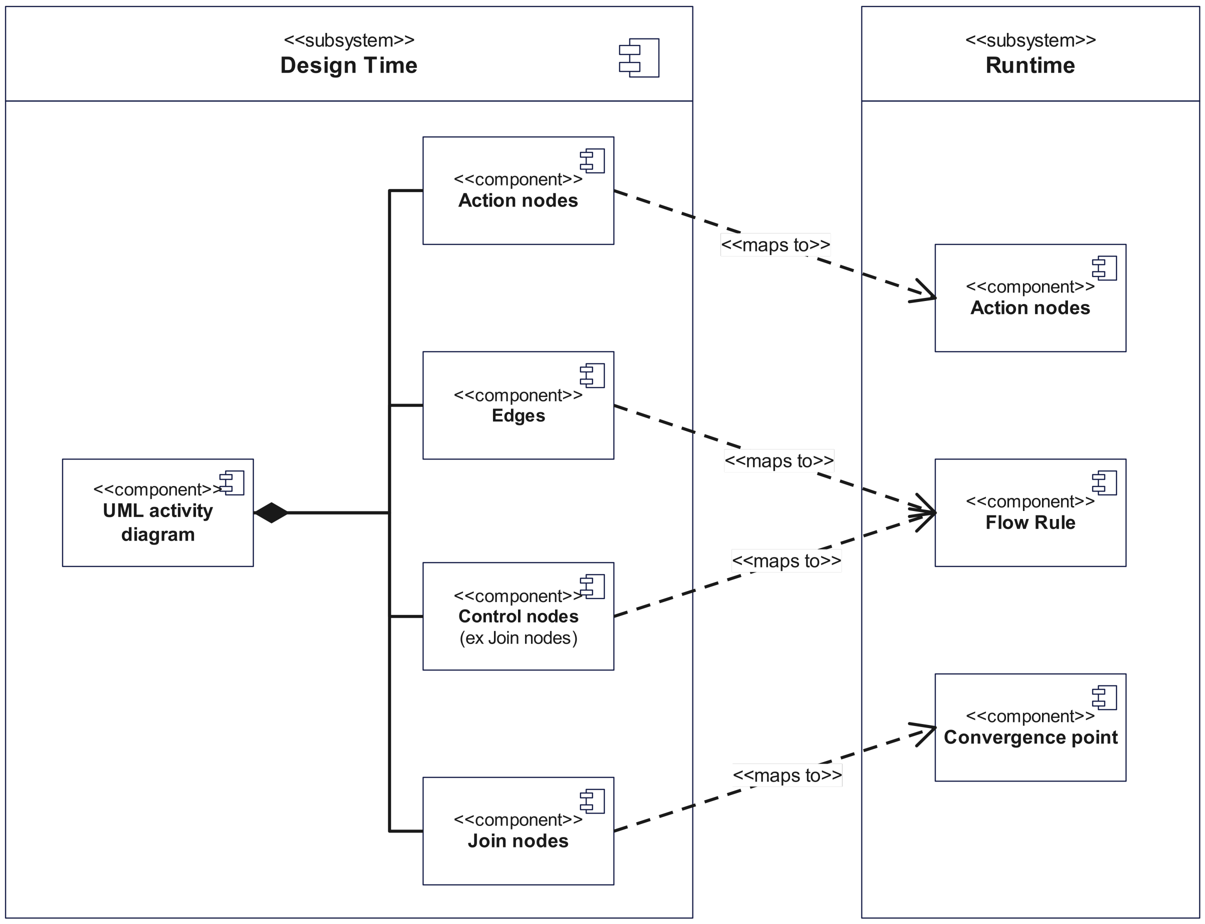


Figure 2: Metadata mapping

4.1.1 Prototype generation

The transformation from the activity diagram metadata, provided by the AI4MDE studio, to a structured format that can be used by the workflow engine is a crucial step in the generation process. The metadata represents the data behind the visual representation of a UML activity diagram, which, although effective for illustrating a process, is not suitable for execution by the workflow engine. This section will outline the mapping from the metadata, into a pre-defined format that ensures that the workflow engine can effectively, interpret and use to execute the modeled activities.

Workflow engine structure

The workflow engine will be responsible for storing the processes created by the user and keep track of the progress. To manage this, the workflow engine organizes the data using four classes, which will be added during the generation process of the AI4MDE project.

1. **Process**: This class stores the process the user created, gathering all the different nodes from the Action Node class into one process. It also holds the start point of the processes as references to a specific action node.

2. **Action Node**: Each node represents a specific action within a process.

3. **Flow Rule**: Governs the conditions under which the workflow transitions from one node to another. This is further explained in section 4.2.

4. **Active Process**: This class keeps track of the processes currently being executed in the application. It monitors progress by referencing the nodes that are currently active through the Active process Node class.

5. **Associated Model**: This class serves as a reference point to any other class within the system. This link to another class can be used to reference any data instance through the associated model instance from any class.

6. **Associated Model Instance**: Stores the specific instance ID of the model referenced by the active process. The ID the model belongs to is determined through the Associated Model class.

7. **Convergence Point**: This class holds the points where different parallel action nodes converges into one control flow. It holds the amount of control flows required to reach this point before a workflow is allowed to continue.

8. **Active Convergence Point**: Tracks the runtime status of a convergence point, storing the amount of parallel action that have been completed before allowing the workflow to proceed.

The class diagram depicted in Figure 3 illustrates these classes and their relationship within the workflow engine. Additionally the classes are divided between two core components: those used for the process storage, marked in blue, and those used during the workflow execution, marked in red. The division is illustrated in Figure 4.

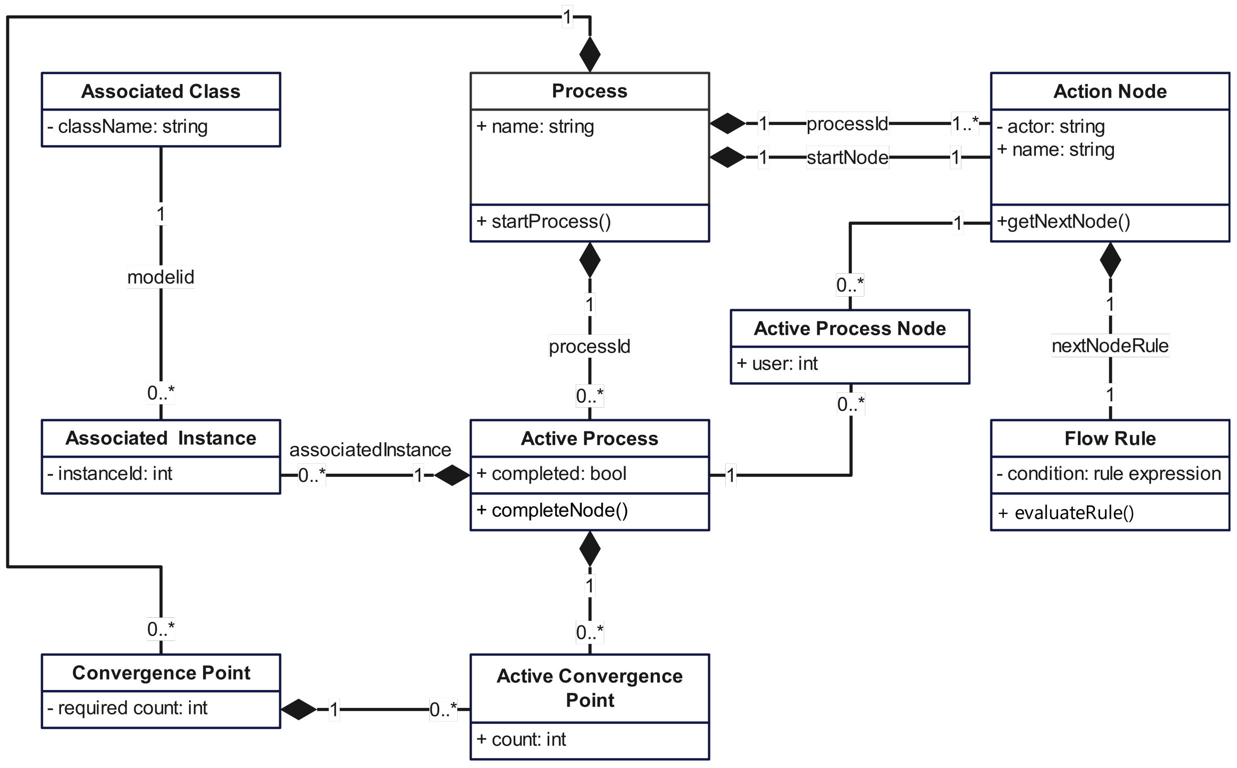


Figure 3: Workflow engine class diagram

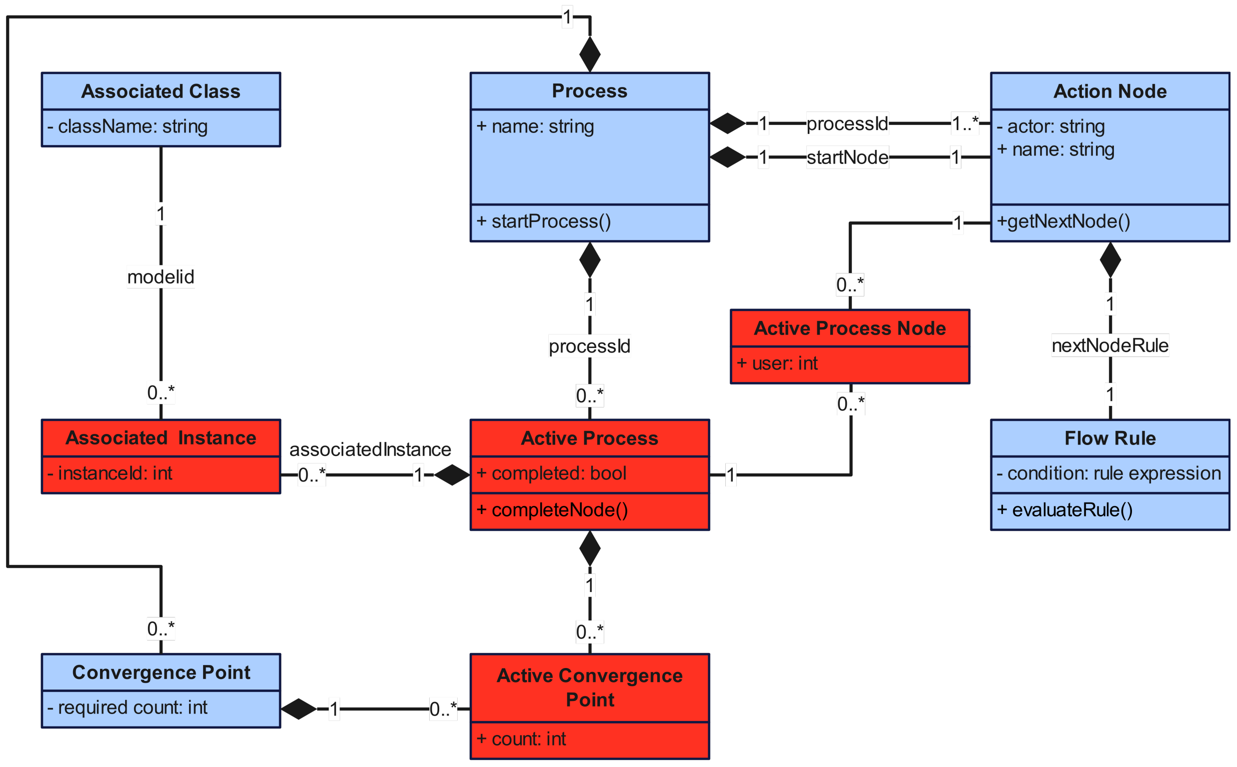


Figure 4: Class division between process storage (blue) and runtime execution (red)

Metadata mapping

With the structure of the workflow engine outlined, the next step is to describe how the AI4MDE activity diagram metadata is mapped to the structures highlighted in blue in Figure 4. Initially, the Associated Models will be created from the class diagrams in the metadata, after which the process mapping can begin.

The following process will be performed iteratively for each activity diagram in the metadata. The mapping starts by creating an entry in the Process class. Next, the Node class is populated using the action nodes in the metadata. Control nodes are skipped, since they do not represent an action that should be executed but instead define the flow of the process. An exception to this are the join nodes. For these the number of incoming edges are counted and an Active Convergence Point entry is made. The other control nodes are analyzed separately and mapped to Flow Rules. This mapping process will work by iteratively going over the created action nodes. For each action node an empty flow rule will be created. Next, the type of the target node will be determined. If the target node is an action node, it is assigned as the target of the flow rule, and the flow rule is considered complete. In the case the target node is another control node, the flow rule is further expanded based on its specific type. The method on how the flow rules are expanded based on each different type is described in section 4.2. Once the control node is added to the flow rule, its target node(s) are/is identified, and the same process is recursively applied until the flow rule is completed after which it is added to the Flow Rule class.

A successful execution of this mapping will result in processes existing of nodes which correspond to activities in the UML diagram, each of which is governed by a flow rule that determines the next node based on the context of the process. This mapping will take place during the generation process of the application, immediately following the creation of the structures. The activity diagram depicted in Figure 5 illustrates the mapping described above.

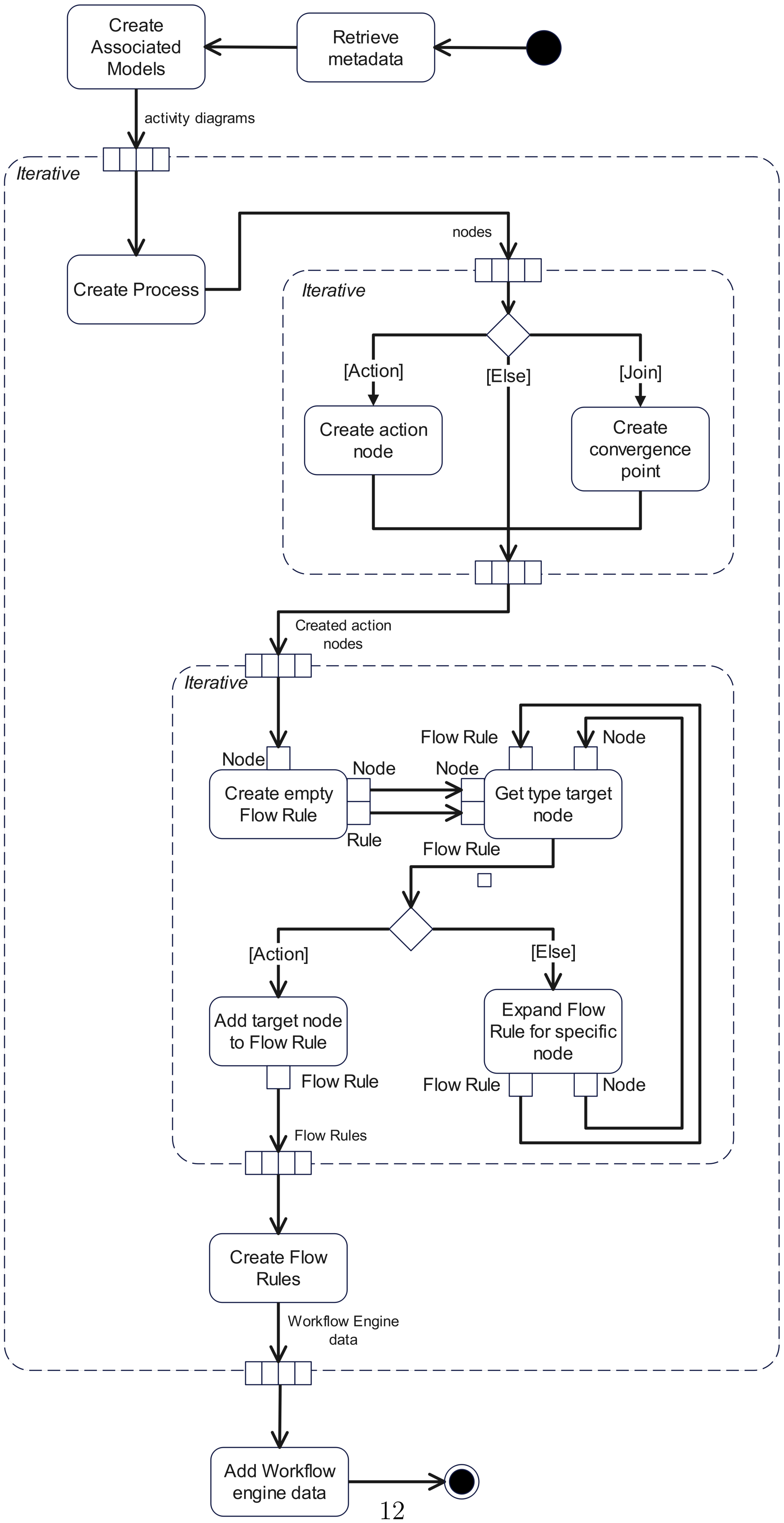


Figure 5: Activity diagram metadata mapping

4.1.2 Runtime execution

The workflow engine is a central component responsible for managing the life cycle of processes. It ensures that each process progresses according to predefined rules. When a new process starts, the engine identifies the starting point and activates the corresponding workflow. Throughout the execution of the workflow, the engine continuously tracks and updates the process context. This is done by linking any relevant information created during the workflow execution to the ongoing process. For example if a user adds items to an order, those items become part of the process context. This context is in term used to dynamically interpret flow rules to determine the next steps in the process. It also controls what information is displayed to the user, ensuring that only data relevant to the current state of the process is shown. The engine is also responsible for synchronization of different parallel paths, such as ensuring all parallel control flows converge before advancing to the next step in the process. A process is considered complete when all paths have been executed and no further actions remain. Section 5.4 will further explain how this is achieved by describing the interactions with the classes defined in Figure 3.

4.2 Flow Rule language

Flow Rule syntax
A Flow rule is defined as a set of possible transitions to the next action node:

$$\text{FlowRule} = \{T_1, T_2, \ldots, T_n\}$$

Each transition T is an entry with the following structure:

$$T = \{\texttt{next} : \textit{Target},\ [\texttt{condition} : \textit{Condition}],\ [\texttt{check} : \textit{Convergence point}]\}$$

Where:

- `next`: Specifies the target of the transition. It can be:
  - The next node,
  - the keyword `END`, indicating termination,
  - Another Flow Rule. In this case it is evaluated recursively using the same procedure.
- `condition` *(optional)*: A logical predicate that determines whether the transition is eligible based on the current context. It is defined by the following attributes:
  - `operator`: Specifies the comparison operator. Supported values are: `eq` (equal), `ne` (not equal), `lt` (less than), `le` (less than or equal to), `gt` (greater than), and `ge` (greater than or equal to).
  - `threshold`: A value against which is compared
  - `aggregator` (optional): An optional function to aggregate values if the condition is evaluated over a set. Supported values are: `SUM`, `AVERAGE`, `MAX`, `MIN` and `COUNT`.
  - `target_attribute`: The name of the attribute to be evaluated.

  - `target_class_name`: The class or object the attribute belongs to
  - `target_attribute_type`: The data type of the attribute being compared (e.g., numeric, categorical).

- `check` *(optional)*: The Convergence point t whose evaluation determines if a transition can be taken. When a transition either has no condition or its condition evaluates to true, the associated check (if present) decides whether the transition will be taken. If the check evaluates to false, no transition is taken, and the evaluation halts.

Flow Rule Evaluation

Whenever a flow rule is being evaluated, the set of transitions are sorted by the presence of a condition in a descending order. To evaluate a flow rule the following predicated will be defined:

$$\mathsf{Cond}(d_i) := \text{“Condition } d_i \text{ holds in the current context”}$$

$$\mathsf{Check}(c_i) := \text{“Convergence point } c_i \text{ is satisfied”}$$

In case the `condition` or `check` is absent, the corresponding predicate is will behave in the following way

$$\mathsf{Cond}(d_i) = \top \quad \text{if } \texttt{condition}_i \text{ is absent}$$

$$\mathsf{Check}(c_i) = \top \quad \text{if } \texttt{check}_i \text{ is absent}$$

The Flow Rule is evaluated by selecting the first valid transition based on the condition and check predicates. Let T_i be the i-th transition after sorting. Then:

$$i^* = \min \{i \mid \mathsf{Cond}(d_i) = \top\}$$

If such an index i^* exists, then the evaluation proceeds as:

$$\text{Evaluate}(T_{i^*}) = \begin{cases} \texttt{next}_{i^*} & \text{if } \mathsf{Check}(c_{i^*}) = \top \\ \bot & \text{otherwise} \end{cases}$$

If no such i^* exists then:

$$\text{Evaluate(FlowRule)} = \bot$$

This case should only happen when the user has made a mistake in modeling the activity. A worked out example of a rule evaluation can be found in Appendix A.1.1

5 Technical architecture

5.1 Metadata extension

The current implementation of the AI4MDE diagram editor already provides support for the creation of the majority of activity diagram elements, but lacks several features that are essential for generating workflow-aware applications. To address these limitations, extensions to the metadata are necessary. In the context of this project, metadata refers to the structured representation of the elements within UML diagrams as well as information about the interfaces that have to be generated. This metadata is in turn used to generate a working Django application.

Role-Based responsibility
Real-world workflows typically involve multiple actors, making it essential for activity diagrams to support the assignment of actors to specific user roles. As previously mentioned in section 2.2.1 swimlanes are used in activity diagrams to represents these role assignments and therefore will have to be added to the UML activity diagram editor. To support this, the metadata model of the activity diagram is extended. A new attribute called *actorNode* is introduced to the metadata of each action node. The *actorNode* attribute serves as a reference to an existing actor node within a use case diagram.

Automated actions
In many business processes, most activities are not performed by users but are instead automated by the system without requiring user intervention. An example of this is sending an email or calling an external API after finishing an order. The UML activity diagram modeling interface must support the distinction between manual and automated actions, as well as provide a way to define what automated actions should execute. Additionally, the interface should allow the user to specify when an activity is scheduled to start, making it possible to automate a workflow entirely. The metadata of the action node is extended with two new attributes. The first attribute, *isAutomatic*, is a boolean flag which indicates whether an action should be executed automatically by the system or manually by the user. The second attribute, *customCode*, is used to define the logic that should be executed when the action is marked as automatic. This attribute holds a block of Python code that is run by the workflow engine whenever it encounters an action node linked to this code during process execution. To be able to start a workflow at a scheduled time, the *scheduled* attribute was added to the initial node metadata. If the *scheduled* attribute is set to true, the *schedule* attribute must also be provided. The *schedule* attribute contains a CRON expression that specifies the timing for when an workflow should be initiated. This schedule data is used to configure the `django-crontab` package, which handles the automatic execution of scheduled actions within a Django application.

Context-aware user interface
The current user interface of the generated prototypes provides the user with a way to interact with all available data in the system, regardless of the workflow context. For example, when deleting an item, the interface displays all item instances, rather then restricting the view to only those linked to the current activity. A distinction should be made between pages that offer global access to data and those that are context-specific. To support this, the page metadata schema, originally defined by de Hoogd [dH24], was extended. A new attribute called *type*, which classifies each page as either *normal* or *activity*, was introduced. Based on this classification, application components such as views, URLs and templates are generated differently to ensure only data related to the current workflow process is displayed.

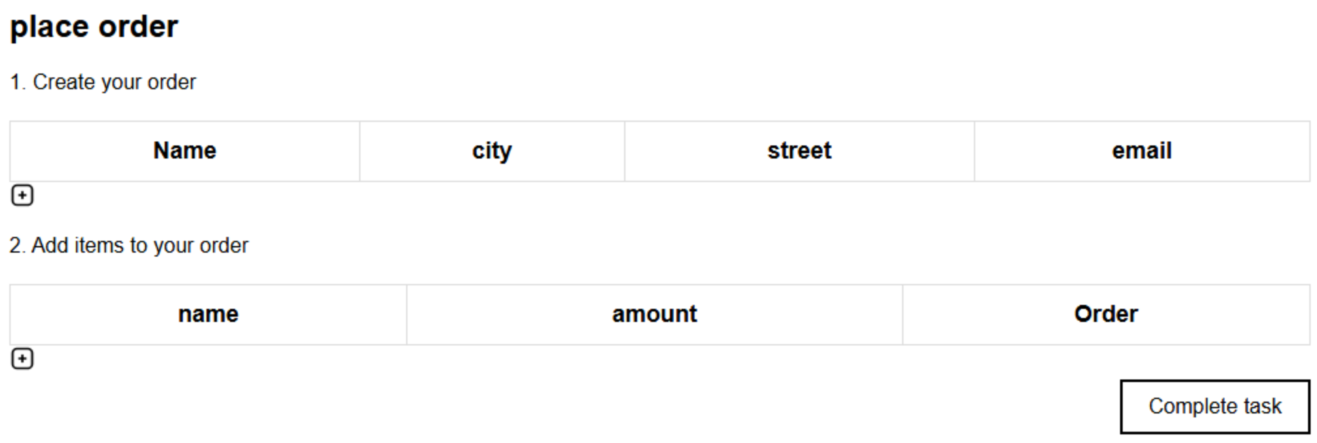
Management interfaces
The introduction of a workflow to the application brings the needs for workflow management. These pages should only be accessible by actors with elevated privileges. A distinction should be added to an application whether this actor has managerial rights and grant access to these pages accordingly. To facilitate this the boolean flag *managerAccess* was added to the interface metadata. These new pages are further explained in Section 5.2.

Conditions
Currently a decision node can be added to a UML activity diagram in the design studio. However, there is no way to define a condition for the outgoing edges of a decision node. To address this, the editor should prompt the user to specify a condition whenever an edge is created with a decision node as its source. The metadata of the `ControlFlow` edge was extended with the *condition* attribute. This attribute follows the same structure as defined in Section 4.2. Additionally, it includes an *isElse* attribute, which indicates whether the condition serves as an “else” transition, which is taken when no condition is met.

Improved diagram edges
During the creation of more advanced activity diagrams, it became apparent the existing method of generating edges was lacking. The system is only capable of creating straight edges between two points. This often caused the edges to intersect other lines or pass through nodes. To improve clarity the edge metadata in the design studio was extended to include *PositionHandlers*. The handlers define an x and y coordinate through which an edge must pass through, enabling the creation of angled edges and corners.

5.2 User Interface design

The workflow engine is reflected in the user interface through several key elements. Upon accessing the applications, the users are presented with the home page. This home page was extended to include a list of processes the user is allowed to initiate, as well as a list of pending actions that require their attention within active workflows. Upon clicking an item in either list, the user is redirected to the corresponding activity page, where they can perform the required task. A visual representation of the interface can be see in Figure 6.



place order

1. Create your order

Name	city	street	email

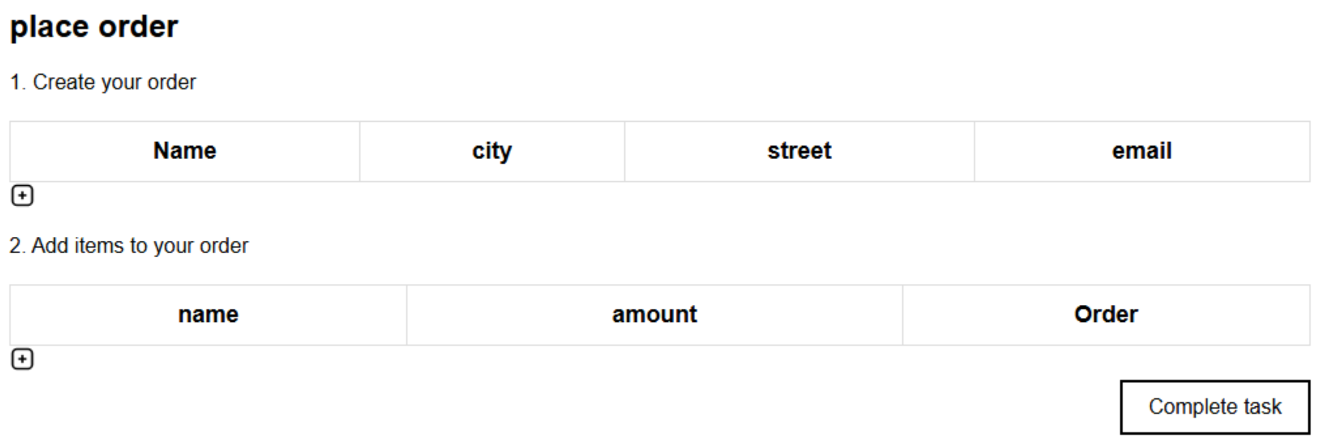
2. Add items to your order

name	amount	Order

Complete task

Figure 6: Prototype Home page

Once the tasks has been completed, the user can finalize the activity by clicking the *Complete Activity* button in the bottom right of the activity page, which marks the tasks as finished and updates the progress of the process accordingly. An example of an activity page can be found in Figure 7



place order

1. Create your order

Name	city	street	email

2. Add items to your order

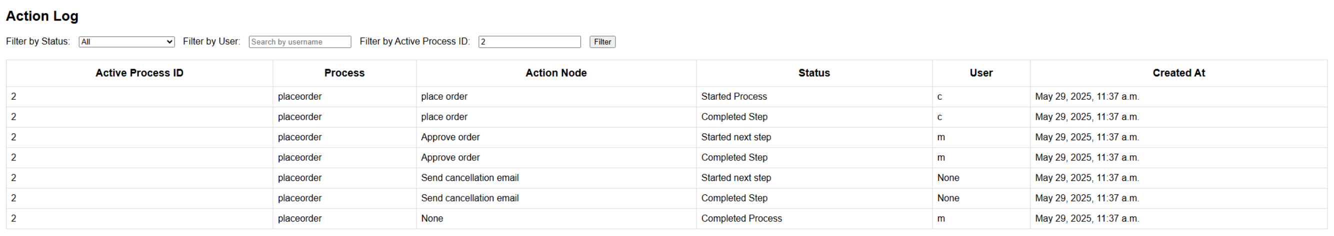
name	amount	Order

Complete task

Figure 7: Prototype Activity page

To improve flexibility of and the insight into the workflow engine, additional managerial pages have been introduced. The first of these pages is the action log page, which has an overview of all activity related events, such as when tasks where completed and which one are currently active.

The second page allows for the reassignment of activities to different users, enabling flexibility in task distribution. Examples of these page can be found in Figure 8 & 9.

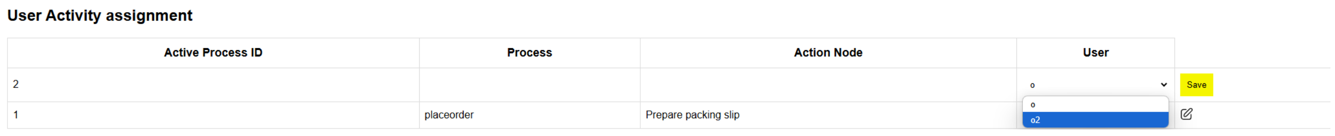


Action Log

Filter by Status: All Filter by User: Search by username Filter by Active Process ID: 2 Filter

Active Process ID	Process	Action Node	Status	User	Created At
2	placeorder	place order	Started Process	c	May 29, 2025, 11:37 a.m.
2	placeorder	place order	Completed Step	c	May 29, 2025, 11:37 a.m.
2	placeorder	Approve order	Started next step	m	May 29, 2025, 11:37 a.m.
2	placeorder	Approve order	Completed Step	m	May 29, 2025, 11:37 a.m.
2	placeorder	Send cancellation email	Started next step	None	May 29, 2025, 11:37 a.m.
2	placeorder	Send cancellation email	Completed Step	None	May 29, 2025, 11:37 a.m.
2	placeorder	None	Completed Process	m	May 29, 2025, 11:37 a.m.

Figure 8: Prototype Action log page



User Activity assignment

Active Process ID	Process	Action Node	User	
2			o	Save
1	placeorder	Prepare packing slip		

Figure 9: Prototype User assignment page

5.3 Prototype generation

This section will outline how the classes defined in Figure 3 will be implemented. In addition to this it will be explained how activity diagram metadata generated by the AI4MDE studio will be used to populate the aforementioned classes. First a new Django app called *workflow_engine* is created. This new app will be responsible for managing the storage of workflows and overseeing their progression. This implementation is chosen, because it makes the workflow engine easily accessible from each actor-specific application.

Class implementation

All classes, except the Associated Model class are added to a *models.py* file, which is copied to the *workflow engine* application during generation. These classes are defined as standard Django models. The Associated Model class, however, was not required during implementation. This is because it is replaced by Django's *ContentType* framework [Fou24b]. The *ContentType* framework provides a generic interface for working with your models by storing information about all models installed in your project. By making use of the *GenericForeignKey* provided by this framework, an *ActiveProcess* instance can be dynamically associated with any other model, without the need for predefined relationships. Additionally, an *ActionLog* class was introduced to enable the enable logging of user and system actions performed during process execution. The class diagram of the technical implementation can be found in Figure 10.

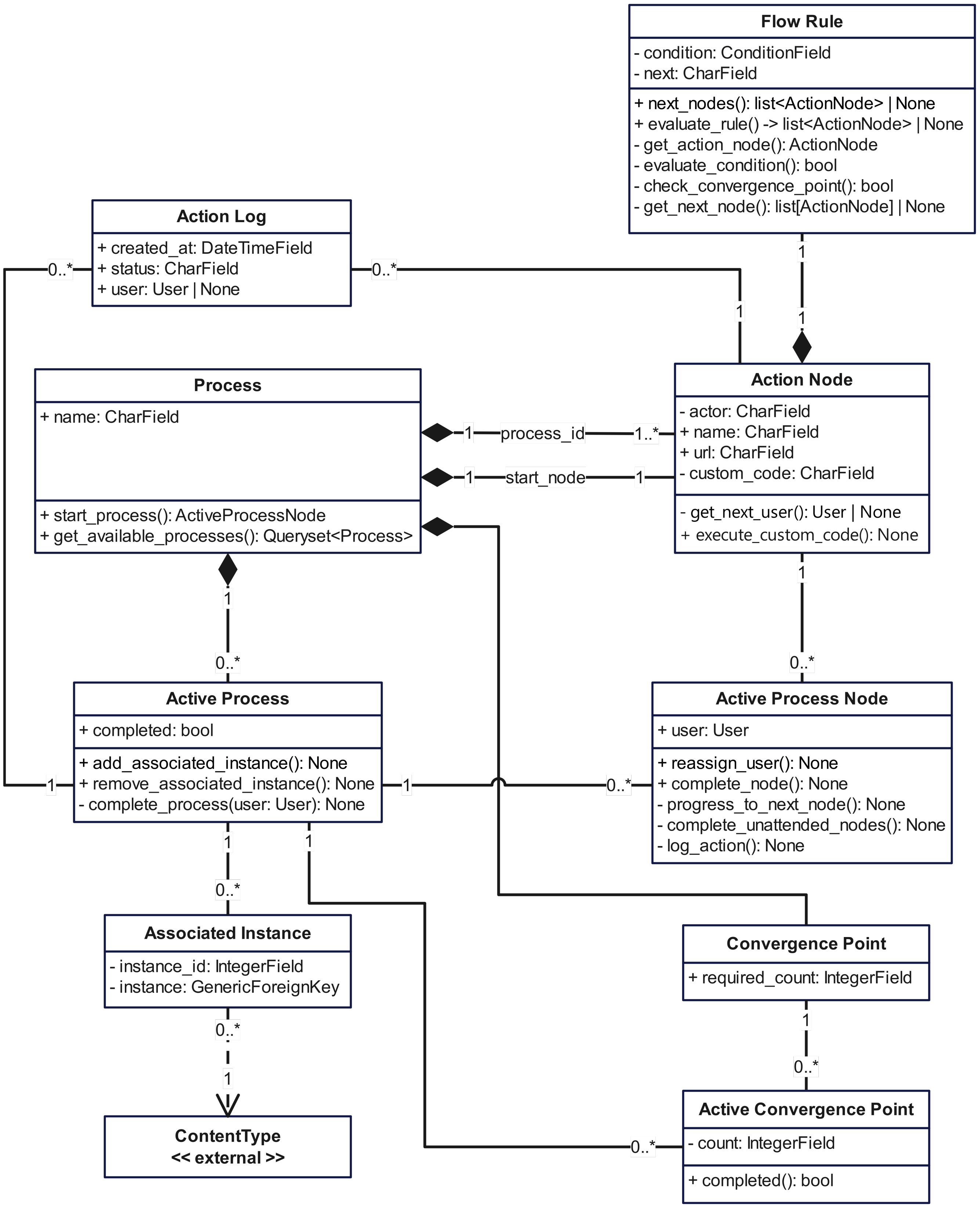

Flow Rule
- condition: ConditionField
- next: CharField
+ next_nodes(): list<ActionNode> | None
+ evaluate_rule() -> list<ActionNode> | None
- get_action_node(): ActionNode
- evaluate_condition(): bool
- check_convergence_point(): bool
- get_next_node(): list[ActionNode] | None
Action Log
+ created_at: DateTimeField
+ status: CharField
+ user: User | None
Process
+ name: CharField
+ start_process(): ActiveProcessNode
+ get_available_processes(): Queryset<Process>
Action Node
- actor: CharField
+ name: CharField
+ url: CharField
- custom_code: CharField
- get_next_user(): User | None
+ execute_custom_code(): None
process_id
start_node
Active Process
+ completed: bool
+ add_associated_instance(): None
+ remove_associated_instance(): None
- complete_process(user: User): None
Active Process Node
+ user: User
+ reassign_user(): None
+ complete_node(): None
- progress_to_next_node(): None
- complete_unattended_nodes(): None
- log_action(): None
Associated Instance
- instance_id: IntegerField
- instance: GenericForeignKey
Convergence Point
+ required_count: IntegerField
ContentType
<< external >>
Active Convergence Point
- count: IntegerField
+ completed(): bool
0..*
1
1..*

Figure 10: Technical Class diagram

Workflow engine population
The workflow engine's data is populated through the parsing of the activity diagram metadata, which is initially provided as JSON input. This JSON is first converted into a graph structure that represents the relationship between nodes and their transitions. The new structure is more suitable when further processing the metadata. The parsing process involves iteratively processing each activity diagram. Each diagram has a start node, which serves as the entry point for the graph. From this start node, the process recursively traverses through the nodes in the diagram, capturing the transitions between them. By following these transitions, the parsing process builds a graph structure, where each node is linked to its subsequent nodes based on the diagrams flow. The algorithm terminates in two cases:

1. **End of diagram**: The process stops when it reaches a final node or a node that has no outgoing edges.
2. **Already visited nodes**: The process also terminates when a node has already been visited during the recursive traversal. This ensures that the same node is not processed more than once, which is important when dealing with merge and join nodes, as well as when loops are present in the process.

Following the creation of the graph structure, the next step is to convert the graphs into their respective entries for the workflow engine. The entry for the process is created first, as this allows the subsequent creation of action nodes to correctly reference the process ID to which they belong. The action nodes are created by recursively traversing the graph structure and skipping any control nodes. Following the creation of the action nodes, the flow rules that define the transitions between these nodes are created. The conditions are formulated by evaluating the nodes outgoing control flows. The conditions are recursively extended until they either reached another action node or a final node, at which point the recursion terminates and the condition is returned. Once this is done for every activity diagram, a JSON object is returned containing all the entries for the workflow engine. This JSON object is copied to the migrations folder of the prototype, alongside with a migration file that utilizes this data to insert the entries into the database ensuring the workflow engine is filled with the appropriate data.

Views generation
When extending the view generation to accommodate activities, some problems with the previous generation method where revealed. The most notable issue was the creation of multiple views with largely overlapping functionality. For instance, when creating a page responsible for creating model instances three or more different views where created. One of these views was called when actually creating the instance, one was responsible for rendering the creation page by returning all instances of that type and the third mirrored the behavior of the second, but in addition only returned an extra boolean indicating whether the creation form should be displayed. Similar duplication in the render views were observed when generating update pages or incorporating custom model methods, where a new page render view was created for each method. This duplication became particularity problematic when integrating activity-specific views. For these views it is important not all instances of a model are returned, but only those associated with the current activity. This filtering must be consistently applied across all relevant views, which due to the duplication, had to be implemented in a large amount of separate view definitions. To address this issue the view

generation was rewritten to use Django Class Based Views (CBV) instead [Fou24a]. By using CBV the logic to add extra context, such as when to display a creation form, could be moved to a single class which the other views can inherit from. This reduced the amount of views required to at most two, irrelevant of how many custom methods where implemented. This transition to Django Class Based Views also made it possible to adopt Django's generic views, which further reduced the amount of code required per page. To ensure reusability of the parent view classes, they were implemented in the *views.py* file within the *shared_models* application.

Authentication
While initially optional, authentication became required after incorporating activity diagrams into the system's generation architecture. This is because this change introduced task assignment tied to specific users, making it necessary to verify a user's identity. Assigning tasks to individual users, rather than to general roles, ensures accountability and enables fair workload distribution. Without this, managing the workflow would become challenging, as the process would effectively operate as a black box, offering no visibility into who is responsible for which task.

5.4 Runtime execution

At runtime the workflow engine will be managing and executing the workflow processes. This is done by adding to or altering the classes marked in red in Figure 4. The engine has two primary endpoints responsible for process management. The first is responsible for initiating new processes. This is done by creating a new *Active Process* entry and assigning an *ActiveProcessNode* to it, corresponding to the start node defined in the *Process* configuration. The second endpoint is used to progress an *ActiveProces*. It takes an existing *ActiveProcessNode* as input and evaluates the flow rule associated with the *ActionNode*. Any new action nodes retrieved from this evaluation will be set as new active process nodes. How this evaluation is performed is explained in more detail in Section 5.5. To be able perform these evaluations, the workflow engine must have access to the process context. This is where the *AssociatedInstance* class is used. It leverages the Django's ContentType framework to generically link any model instance created during the process to the corresponding *ActiveProcess*, allowing flow rules to dynamically access and evaluate relevant data. This data is linked by adding an additional step in the views responsible for creating data within a process step. In some workflows, the next node can only become active once all parallel paths leading to a join have been completed. This is where the *ConvergePoint* and *ActiveConvergepoint* classes come into play. Whenever a control flow from one action node reaches a convergence point, the engine checks whether that *ConvergenPoint* has already been completed. This is managed through the *ActiveConvergepoint* class, which tracks how many control flows have arrived. Once the number of incoming control flows matches the expected amount, it is considered as complete and the workflow can continue. After all flow rules have been evaluated and the active nodes have been updated, the number of remaining active nodes is checked. If none are present, the active process is marked as complete. The execution logic described above is illustrated in Figure 11.

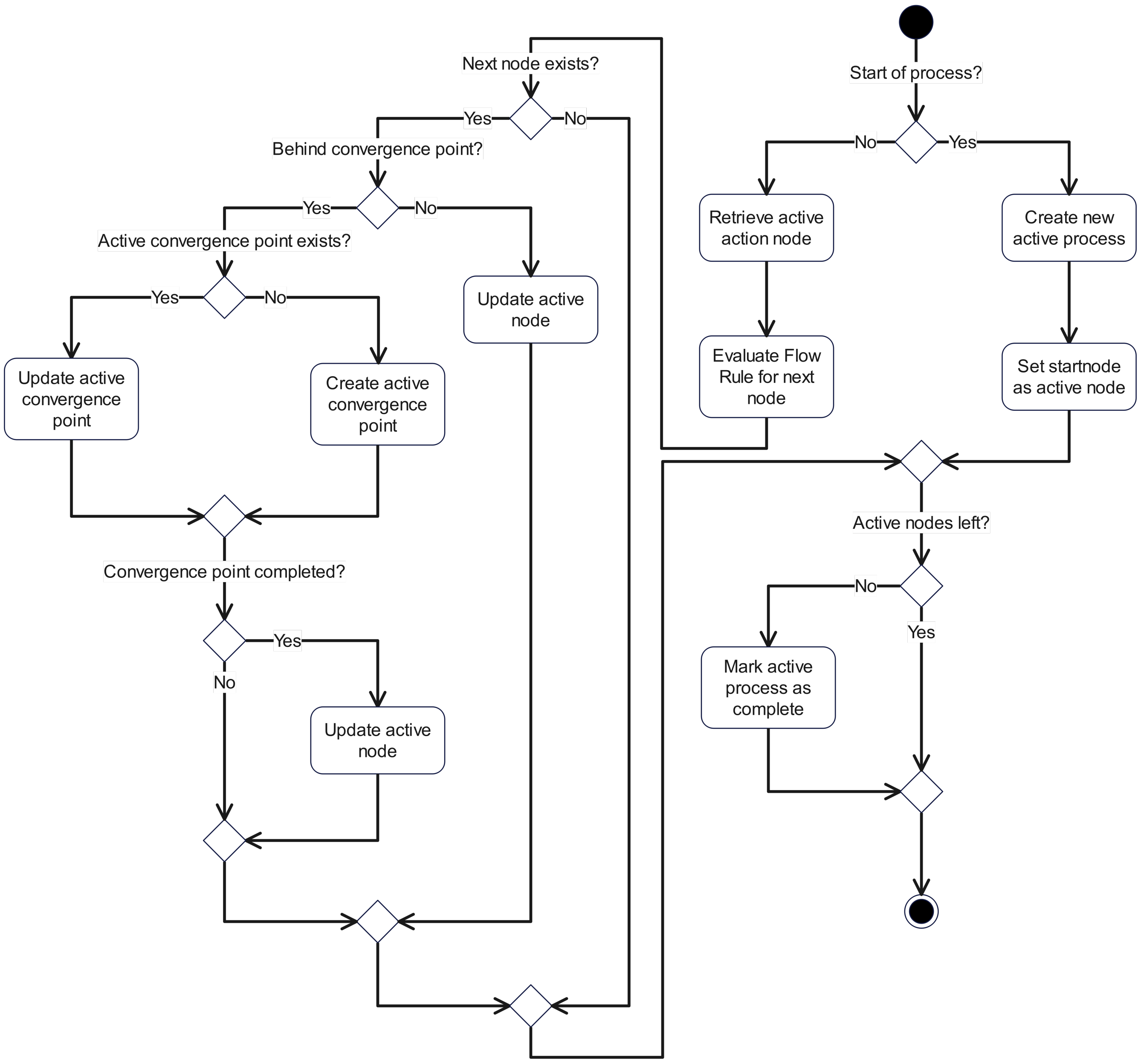

Start of process?
No
Yes
Retrieve active action node
Create new active process
Evaluate Flow Rule for next node
Set startnode as active node
Active nodes left?
No
Yes
Mark active process as complete
Next node exists?
Yes
No
Behind convergence point?
Yes
No
Active convergence point exists?
Yes
No
Update active node
Update active convergence point
Create active convergence point
Convergence point completed?
Yes
No
Update active node

Figure 11: Workflow engine runtime execution

5.5 Flow Rule evaluation

As mentioned in the previous section, progressing an *ActiveProcessNode* involves evaluating the *FlowRule* of the associated *ActionNode*. Each flow rule consists of two attributes: `next` and `condition`. The `next` attribute is used to define straightforward transitions within the workflows that do not depend on any runtime data and thus do not require any conditional logic. An example of such a flow is a direct transition from one action node to the next or from an action node to an end node. On the other hand, the `condition` attribute is used for more complex flows that require evaluation during runtime using the context of the active process. Conditions are stored in JSON format in the database and as mentioned in Section 4.2, consist of a list of possible transitions. To improve the usage of this data, a custom Django model field is implemented, which inherits from Django's built-in `JSONField`. This custom field automatically parses the raw JSON into a more structured Python representation when retrieving it from the database. The following pseudo-code outlines the steps involved in determining the next nodes to be set active in the workflow:

Algorithm 1 Determine the Next Node in a Workflow

```
Require: active_process, list of next_nodes
Ensure: List of action nodes to activate or None
 1: Sort next_nodes so those with conditions come first
 2: for each node in next_nodes do
 3:     if node has a condition AND _evaluate_condition returns False then
 4:         continue to next node
 5:     end if
 6:     if node.check exists AND _check_convergence_point is False then
 7:         return None
 8:     end if
 9:     Set next_value = node.next
10:     if next_value is a list of integers then
11:         return list of action nodes for each ID
12:     else if next_value is a list of nodes then
13:         return recursive call to _get_next_node with the list
14:     else if next_value is a single integer then
15:         return list with one corresponding action node
16:     else if next_value is "END" then
17:         return None
18:     end if
19: end for
20: return None
```

As can be seen in Algorithm 1, determining the next node(s) in the workflow requires evaluating both conditions and convergence logic. Specifically a transition can only be considered if a possible associated condition is met. For this the condition must be evaluated against the current state of the process data using the algorithm described in 2. Next to this condition evaluation, Algorithm 1 also incorporates convergence logic to ensure proper synchronization of parallel execution paths. This logic is described using pseudo-code in Algorithm 3.

Algorithm 2 Evaluate a Condition Against Process Data

```
Require: active_process, condition
Ensure: True if the condition is satisfied, else False
 1: Get target objects from active_process using condition.target_class_name
 2: if condition uses an aggregator then
 3:     Compute aggregate value (sum, avg, max, min, or count) on condition.target_attribute
 4: else
 5:     Retrieve target attribute from the first object in the target set
 6: end if
 7: Convert threshold to correct type based on condition.target_attribute_type
 8: Compare the value to the threshold using condition.operator
 9: return  result of the comparison
```

Algorithm 3 Check Convergence Point Completion

```
Require: active_process, convergence_point_id
Ensure: True if convergence point is completed, else False
 1: Retrieve active convergence point for the given process and ID
 2: if no convergence point exists then
 3:     Create a new convergence point for the process
 4:     return  False
 5: else
 6:     Increment the convergence point's count, denoting a new action reached it.
 7:     Save the updated convergence point
 8:     if convergence point is not yet completed then
 9:         return  False
10:     else
11:         Delete the completed convergence point
12:         return  True
13:     end if
14: end if
```

5.6 Runtime adaptability

5.6.1 Rule configuration and extendibility

As mentioned in Section 2.3, the workflow engine is data-driven. At runtime, this allows user to alter the flow between activities by modifying the Flow Rule data in the database. A example use case for this is adjusting the threshold that determines when an order requires additional check, by adjusting the threshold value in the condition JSON data. Next to altering the activity flow, users will also be able to extend the flow rule language itself with relative ease by modifying the code. Specifically, new operators as well as new aggregators can be added by updating the relevant logic in the `models.py` file in the `workflow_engine` application. The following Python snippet from the `FlowRule` class demonstrates how to extend the language by adding a new operator (between) and aggregator (exists).

```
# In workflow_engine/models.py

class FlowRule(models.Model):
    type_converters = {
        # Existing type converters...
        "tuple_int": lambda x: tuple(map(int, x.split("-"))), # New converter
          that converts the threshold value, which is stored as a string, into
           the desired format.
    }

    operators = {
        # existing operators...
        "between" lambda value, threshold: threshold[0] <= value <= threshold
           [1] # New custom operator
    }

    aggregators = {
        # Existing aggregators...
        "exists" lambda qs, _: qs.exists(), # New custom aggregator
    }

    # Further implementation of FlowRule class...
```

Listing 1: Extending the Flow Rule with a Custom Operator

By adding new entries to the operators and aggregators dictionaries, users can now use these by altering the condition in a flow rule entry. The type converter is applied before calling the operator method to ensure the value is in the correct format, since these are stored as string. In this example, the type converter transforms the threshold string “1-100” into a tuple (1, 100) to enable correct comparison. This threshold convert can be used by changing the `target_attribute_type` of the Flow Rule condition in the database.

5.6.2 User Assignment

In addition to being able to control the flow of activities during runtime using configurable rules, the workflow engine is also flexible in determining the user who is assigned to an activity. The current implementation follows a basic strategy: it assigns the next activity to the same user who completed the previous one, if that user holds the required role. If not, the engine selects the first user it encounters who holds the required role. This can be changed by adjusting the `_get_next_user()` method in the `ActionNode` class, which looks like the following:

```python
def _get_next_user(self, current_user: User | None) -> User | None:
    """Determine the next user for the given node."""
    # No user assignment if there is nothing to do for the user
    if not self.url:
        return None

    # Try to keep the current user for the next node
    if current_user and self.actor in current_user.roles:
        return current_user

    # Adjust this to use a more comprehensive user assignemnt strategy,
    # Such as load balancing, availibility, etc.
    users = User.objects.filter(**{f"is_{self.actor}": True})
    next_user = users.first() if users.exists() else None
    return next_user
```

Listing 2: Adjusting user assignment strategy of Workflow Engine

6 Testing and results

6.1 Case studies

As mentioned in Section 3 case studies were used to evaluate the performance of the model during development.

6.1.1 Case study 1

The initial case study focused on the basic functionality of a UML activity diagram. Despite its focus on fundamental features for the workflow engine, this case study required the most effort to prepare, as it involved significant amount of work on features that support the workflow engine. This includes the refactoring of the Django views as well as creating activity pages.

The first key learning form this case study was the importance of more descriptive naming. During development, the labels had become too technical, which can confused users. To address this, the naming of the activity lists on the home page, as well as the button used to complete an activity were revised. Additionally, the feedback indicated that the activity page felt disjointed and lacked a clear structure. To improve this, explanatory text was added to inform users about the required steps for the selected activity.

Another key insight from this case study was the identification of missing requirements. One of these involved the authentication becoming mandatory after the introduction of a workflow. It was considered to alter the workflow engine to support making authentication optional, but this change was not implemented. Additionally the ability to delete existing processes was lacking. To resolve this an option to do this was added to the home page activity list.

Finally a critical issue was identified during this case study. The system would crash when an activity was completed and the next activity in the workflow had to be assigned to a user with that specific role, but no user with that role was available. To fix this, the action is not assigned to any user when this occurs. Instead a warning is added for the manager to assign this task manually. Screenshots taken during this case study can be found in Appendix A.3.1.

6.1.2 Case study 2

The second case study introduced some more advanced features. This also caused the diagrams to be more elaborate, which exposed a problem within the activity diagram modeler. The lines between the nodes could only be straight lines, which caused them to cross through a lot of other elements. To address this *positionHandlers* were added, which are explained in section 5.1. This case study also introduced automatic tasks. Initially, the task types were labeled as “manual” and “automatic”. The term “manual” was found to be confusing, so it was renamed to “UI task” for improved user understanding. The code editor for this automatic task was also found to lack context, required for the user to implement it. The editor was expanded to include comments and additional lines of code to create a better user understanding. Screenshots taken during this case study can be found in Appendix A.3.2.

6.2 User testing

This section presents the findings from different user interviews conducted to evaluate the extended prototype generator. To analyze this qualitative feedback, the thematic analysis framework proposed by Braun and Clarke [BC06] was used. This method was chosen because it offers a flexible yet rigorous approach for identifying patterns and themes within qualitative data. An inductive approach was used, allowing themes to emerge directly from the data, rather than being shaped by pre-existing theories or assumptions. This was done by making the interviews open-ended in nature. Participants were asked to create an application of their choice using the design studio. During this process, they were encouraged to provide their thoughts on the design process. After this an application was generated and briefly evaluated by the participant in terms of alignment with their original model and overall usability. Finally, participants were open-ended questions about the practicality of using the AI4MDE in real world scenarios. A full record of these interviews can be found in Appendix A.4.

6.2.1 Themes

Based on the thematic analysis of the interview data, the following key themes emerged:

Theme 1: Accurate workflow generation Participants collectively agreed the workflow logic generated from the UML activity diagrams captured the intended workflow. This demonstrates the generators strength to translate activity diagrams into executable code, signifying that a major goal of this thesis has been successfully achieved. When asked, *"Does the generated prototype match your expectations based on the model?*, participants responded with

> *"Yes it did, it followed the activity diagram well."*
>
> *"Yes absolutely."*

Theme 2: Workflow modeling limitations
Participants consistently noted the need for more advanced behavior modeling features within the activity diagram editor. For instance, a participant expressed the desire to schedule activities based on time intervals to enable autonomous execution of tasks. Another raised concerns about the inability to carry over context between processes, which limited their ability to connect workflows effectively. Additionally, the current implementation restricts the use of complex control logic by not being able to chain conditions using logical operators or synchronizing multiple incoming control flows into a single action node. These points emerged from the following participant feedback:

> *"Something that is missing, which is quite vital, is the ability to start an activity based on a schedule (cron-job)..."*
>
> *"The workflow engine can't know the context of other processes."*
>
> *"I was missing the ability to have multiple control flows pointing to a single action node... should also be able to chain conditions using logical operators like AND, OR."*

Addressing these limitations will be crucial for improving the models robustness and its ability to be used in real-world scenarios.

Theme 3: User interface and design process frictions
This theme encapsulates the frustration participants experienced when interacting with the design studio interface. These frustrations consisted of both visual inconsistencies, such as misaligned edges, as well as the design process being inefficient and unintuitive. Common frustrations were, poor defaults when creating nodes, excessive navigation especially when designing the prototype interface as well as difficulty creating custom code due to a lack of contextual guidance or reference examples. These issues caused the design process to take significantly longer than expected, as reflected in the participant feedback:

> *"Currently it takes the user a lot of time to work out the core elements."*
>
> *"It took me way longer then expected, about 4 hours, to finish the design after our first session."*

To address these issues a set of recommended improvements, based on participant feedback, has been compiled, which can be found in Appendix A.2.

Theme 4: Inadequate Prototype UI
This theme focuses on how well the generated prototypes aligned with user expectations. There was a general consensus among participants, that the backend logic aligned with the behavior modeled in the activity diagrams. The visual aspect of the generated user interface, however, were found to be lacking. These concerns were illustrated by the following feedback:

> *"The backend functionality of the prototype did [match], however the looks did not."*
>
> *"For personal projects, I would require the front- and backend to be split"*

As mentioned in the second quote, participants emphasized the need for a dedicated frontend framework, such as React or Vue, instead of solely relying on Django for everything. This is essential, as these frameworks offer features that the Django frontend alone cannot provide, resulting in a richer and more responsive user interface.

Theme 5: AI-assisted design through LLM integration
Several participants proposed the use of Large Language Models to enhance and streamline the design process. Suggestion included using LLMs to generate custom code. A key motivation for this was the difficulty participants faced when writing custom code manually, primarily due to a lack of contextual information provided during the design process. As one participant noted, using LLMs to create the code

> *"Would reduce the need for context when creating this code, since only the LLM would require this"*

The second most prominent use of LLMs is to generate the diagrams. Next to the UI improvements mentioned in Theme 2, this would significantly speed up the design process by reducing the number of manual steps required. As one participant put it:

> *"Diagrams should also be generated by an LLM, enabling the user to focus on the details."*

The motivation to further integrate LLMs into the design process reflects the desire to shift from low-level manual modeling to high-level conceptual thinking.

Theme 6: Limited support for real world scenarios
Participants pointed out that, while the tool showed promise, it was not yet suitable in real-world scenarios. It was emphasized it was currently primarily viable in greenfield projects, and lacks the flexibility to adapt to evolving business. One participant phrased it as follows:

> *"I would consider using this if it was able to operate outside of greenfield projects"*

They further added:

> *"The tool should also be able to change existing projects, as requirements often change after the first deployment"*

7 Discussion

This research set out to address a specific limitation in the AI4MDE project: its inability to generate workflow-aware prototypes from UML models. To address this, a data-driven workflow engine was developed and integrated into the AI4MDE platform. This allowed the automatic generation of prototypes that reflect both the structure and behavior of the UML models. Testing and user interviews confirmed the feasibility of the approach, as all participants reported the generated workflow aligned with expectations after modeling the behavior in the UML activity diagram. However the results also revealed that the modeling process proved difficult for the users, highlighting the need to improve the modeling interface and provide better guidance. This research not only extended the AI4MDE platform, but also addressed a core limitation in many existing projects focused on generating applications from UML diagrams. These projects showed that many solutions rely on a limited number of UML diagrams and therefore were unable to generate complete code, as mentioned in Generating Java code from UML class and sequence diagrams [PSDB11], *An approach to code generation from UML diagrams* [GM14] and summarized by M. Mukhtar and B. Galadanci in *Automatic code generation from UML diagrams: the state-of-the-art* [MG18].

Limitations

Research limitations

Firstly, testing of the generator has been limited. The tool would benefit from broader testing across a more diverse set of users to assess its robustness. Additionally, a larger number of participants responses would likely result in more representative themes during analysis. Secondly, this case study research relied on a single source of evidence, meaning data was collected from only one type of information (interviews, documents, etc.). Yin notes that case study research can benefit from multiple sources to enhance the depth and credibility of the findings [**?**].

Conceptual limitations

The primary conceptual limitation the workflow engine has is its reliance on certain properties within the UML activity diagram. Specifically, the first step in the activity diagram, after the start node, must be an action node. This is because it will be seen as the first node to be set active when starting an activity. Additionally, parallel flows must not contain an end node as this will stop the paths from converging. Furthermore, when evaluating a flow rule, the engine assumes an instance of the relevant class is already linked to the workflow. A second limitation concerns the condition defintions within the worflow engine. Conditions can not be chained together using logical operators, such as AND/OR. Next to this conditions can only be compared to static values and not to other dynamic expression, limiting the flexibility of the workflow engine. Finally the current implementation is limited to working on greenfield projects and thus can not be integrated with existing projects.

Technical limitations

Despite the successful extension of the AI4MDE studio, there are some limitations to the current implementation. Firstly the fronted used to model UML diagrams requires additional tools as well as refinement to optimize the user experience and understanding. This particular important for non-experts users, especially when creating custom code. Lastly testing showed the generated frontend of the application is lacking in comparison to the backend functionality.

8 Conclusion

This research aimed to address a key limitation of the AI4MDE project: its inability to generate workflow-aware prototypes from UML models. The questions guiding this study were as follows:

1. How can a mapping be defined from UML activity models to a low code execution platform?
2. How can such a mapping be implemented by extending the prototype generator in the LIACS AI4MDE project?

To achieve this, a new data-driven workflow engine format was developed that translates activity diagrams into an executable format. This data-driven engine allows dynamic workflow adjustment at runtime, separating itself from existing code-depended engines. This new format was used to extend the prototype generator in the AI4MDE projects. The results show that this approach was successful, as the generator became capable of producing interactive, workflow-aware prototypes using different UML models. These generated applications reflect both the applications structure and behavior, inherently providing more complete code than many existing UML-to-code generation projects.

Future work

Despite the progress achieved during this research, there still remain a significant number of issues. The main areas for improvement have been identified as follows:

Improved frontend/LLM usage
Interviews revealed the need for further enhancements of the design studio's frontend. This includes improvement of navigation and the visual quality of diagrams (e.g. incorrect line connections). A Large Language Model should be added as an aid in the design process. This would not only decrease the design time due to being able to outsource the design of the main outlines, which currently requires significant time, but also allow the users to dedicate more time to the details. This LLM can also be used to generate the custom code, which is currently difficult due to a lack of context.

Data validator
Currently the generation process can fail very easily due to an incorrect input from the user. When this happens it is unclear to the user why it failed unless the Python error is analyzed. A validator should be added in between the design studio and the generator, which informs the user if anything is incorrect.

Workflow engine robustness
As mentioned in the Section 7, the workflow engine engine assumes certain properties in the UML activity diagram. These should either be incorporated into the validator mentioned above, or preferably, be addressed by improving the engine's current design

Decoupled prototype frontend
The generated prototype frontend is currently limited by relying on Django's templating system, which is less flexible than modern frontend frameworks. Future work should focus on generating a decoupled user interface using a framework like React to improve user experience.

References

[BC06] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3:77–101, 01 2006.

[dH24] P. de Hoogd. An effective user experience for prototype generation in model driven engineering. *Bachelor thesis at LIACS*, 2024.

[Fou24a] Django Software Foundation. Django, class-based views, 08 2024. Accessed on April 23th, 2025, Available at: https://docs.djangoproject.com/en/5.2/topics/class-based-views/.

[Fou24b] Django Software Foundation. Django, the contenttypes framework, 08 2024. Accessed on April 11th, 2025, Available at: https://docs.djangoproject.com/en/5.1/ref/contrib/contenttypes/.

[GM14] Harshal D Gurad and VS Mahalle. An approach to code generation from UML diagrams. *International Journal of Engineering Sciences & Research Technology*, 3(1), 2014.

[Gro14] Object Management Group. Mda guide rev. 2.0, 06 2014. Accessed on April 7th, 2025. Available at: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[Gro17] Object Management Group. Omg unified modeling language, 12 2017. Accessed on April 7th, 2025. Available at: https://www.omg.org/spec/UML/2.5.1/PDF1.

[Hau11] Jørgen Hauberg. Research by design : a research strategy. *Revista Lusófona de Arquitectura e Educação*, (5):46–56, 01 2011.

[HH95] David Hollingsworth and U Hampshire. Workflow management coalition: The workflow reference model. *Document Number TC00-1003*, 19(16):224, 1995.

[MCF03] S.J. Mellor, Tony Clark, and Takao Futagami. Model-driven development - guest editor's introduction. *Software, IEEE*, 20:14– 18, 10 2003.

[MG18] Maryam I Mukhtar and Bashir S Galadanci. Automatic code generation from UML diagrams: the state-of-the-art. *Science World Journal*, 13(4):47–60, 2018.

[PSDB11] Abilio G Parada, Eliane Siegert, and Lisane B De Brisolara. Generating java code from UML class and sequence diagrams. In *2011 Brazilian Symposium on Computing System Engineering*, pages 99–101. IEEE, 2011.

[TPS97] David J. Teece, Gary Pisano, and Amy Shuen. Dynamic capabilities and strategic management. *Strategic Management Journal*, 18:509–533, 1997.

[vA22] B. van Aggelen. Enabling data-driven wireframe prototyping using model driven development. *Bachelor thesis at LIACS*, 2022.

[vD24] S. van Dam. A flexible, template-driven generation framework for model-driven engineering. *Bachelor thesis at LIACS*, 2024.

[Wil23] L. Willemsens. Utilising use case models for multi-actor prototype generation. *Bachelor thesis at LIACS*, 2023.

A Appendix

A.1 Examples

A.1.1 Flow Rule Evaluation

For this example we will be evaluating the rule associated with Action Node 1 in Figure 12 under different process contexts.

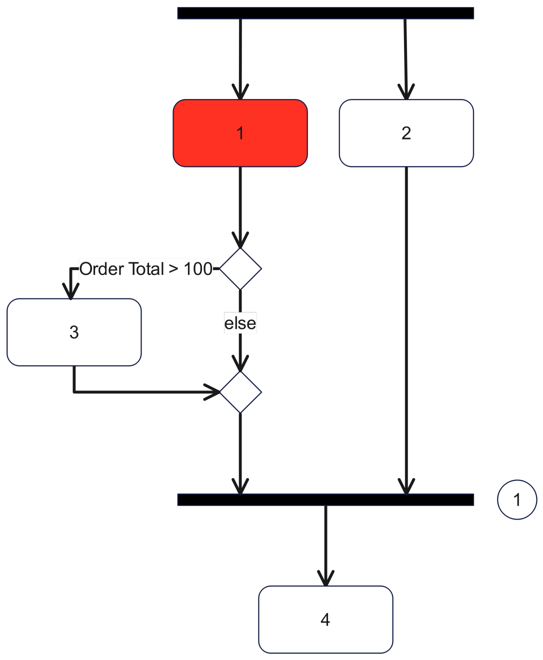


Figure 12: Rule Evaluation Example

The flow rule consists of two possible transitions:

$$T_1 = \left\{ \begin{array}{l} \texttt{next} : 3, \\ \texttt{condition} : \left\{ \begin{array}{l} \texttt{operator} : \texttt{gt}, \\ \texttt{threshold} : 100, \\ \texttt{target\_attribute} : \texttt{order\_total}, \\ \texttt{target\_class\_name} : \texttt{Order}, \\ \texttt{target\_attribute\_type} : \texttt{numeric} \end{array} \right\} \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{l} \texttt{next} : 4, \\ \texttt{check} : 1 \end{array} \right\}$$

First scenario This considers a case where the order total is 500. First the transitions with a condition associated are evaluated. In this case this is transition T_1. Since the order total is 500, which is greater that the threshold of 100, the condition predicate $Cond(d_1)$ evaluates to $\top$. Once a transitions with a satisfied condition is found, the next step is to evaluate the check predicate. Because the check predicate is optional and omitted in this case, it defaults to True, Therefore, $Check(c_1) = \top$ and transition T_1 is selected. As a result Node 3 becomes the new active node.

Second scenario In this case the order total is 50 and action node 2 has already been completed. As in the first scenario, transition T_1 is evaluated first, however is not selected, because the condition predicate $Cond(d_1)$ evaluates to $\perp$, given that 50 is not greater than the threshold of 100. As a result, the next transition, T_2, is considered. Since T_2 does not have a condition defined, it defaults to true. With a valid transition found, the check predicate is evaluated. $Check(c_2)$ will evaluate to $\top$, because Action Node 2 has already been completed, meaning it has reached convergence point 1. Together with the current flow from Action Node 1, this makes a total of two incoming control flows to the convergence point, matching the expected number of incoming paths. Therefore the check predicate passes and action node 4 is set as the new active node.

Third scenario This scenario matches the second, with the exception that Action Node 2 has not yet been completed. As before, T_1 is skipped since the condition predicate evaluates to false. T_2 is once again selected since the condition evaluates to true by default since it is omitted. However, in this case, the check predicate $Check(c_1)$ evaluates to $\perp$. This is because only one control flow (our current control flow), has reached convergence point 1 and thus the required amount of incoming paths has not yet been reached and the transition will not be taken. No further transitions will be evaluated, since the correct transition has already been found, but cannot be used since it has to wait for another action to be completed. Whenever Action Node 2 will be completed, Action Node 4 will be set as an active node.

A.2 Design studio UI improvements

Based on the participant feedback, the following improvements are suggested:

1. Node creation and positioning

1.1 A new node should be inserted at the location of the cursor, instead of the middle of the canvas.

1.2 The add node popup should appear at the location of the user's cursor.

1.3 Improve the default settings when creating a node. For example, when adding an Action node, the user is first required to select Role: action and then choose a type, even though the only available option is an action. This step is redundant and should be improved by setting it as the default.

1.4 When creating a node, you should be able to confirm the action using Enter.

1.5 The position of a node changes when connecting them to a swimlane.

1.6 The user should be able to add a node directly to a swimlane by moving the cursor inside and pressing add node.

2. Swimlanes

2.1 Swimlanes should be correctly rendered as a background layer. Currently, nodes sometimes disappear behind the swimlanes, and the user should be unable to connect edges to the swimlanes.

2.2 Swimlanes should be re-sizable by dragging their edges.

2.3 A multi-select feature should be implemented when creating swimlanes, enabling users to create multiple swimlanes simultaneously.

2.4 You should be able to remove an action from a swimlane.

3. Edges and connections

3.1 The clickable area of an edge should be enlarged to improve usability.

3.2 The edge rendering algorithm should be improved to make sure they align properly with the node. This includes being able to move the edge originating from a fork node, allowing for a better distribution.

3.3 The text on edges representing conditions should be displayed in a distinct color to differentiate them from 'else' edges.

4. Interface design

4.1 Section components as well as categories should be creatable directly from the Pages tab, reducing the amount of navigation required.

4.2 When creating a section component, users should be able to select a CRUD operator for each individual class attribute rather than for the entire class.

4.3 Add a new attribute to the section component creator to specify whether all entries of a foreign class field should be displayed, or only those associated with the active process.

4.4 You should be able to decide whether a foreign key should be editable or not in a section component.

4.5 The user should be able to select or upload a custom SVG icon for their custom method button.

5. Conditions

5.1 When creating a condition, the sequence of selections should be optimized to make it more intuitive.

5.2 The guard input field can be removed when creating a control flow originating from a decision node, since this got replaced by a condition.

6. Additional improvements

6.1 It is easy to lose track of progress during the design process. This could be improved by implementing a progress bar that outlines the steps the user needs to complete.

6.2 The system box should be added to the use case diagram.

6.3 The difference between a UI and manual activity should be explained further in the UI.

6.4 The user is currently unable to change the type of a class attribute without changing the name.

A.3 Case studies

A.3.1 Case study 1

Case study 1 commit hash: 66aa7ff65d5b2614bdf62916f1e4dcfac6a4774a

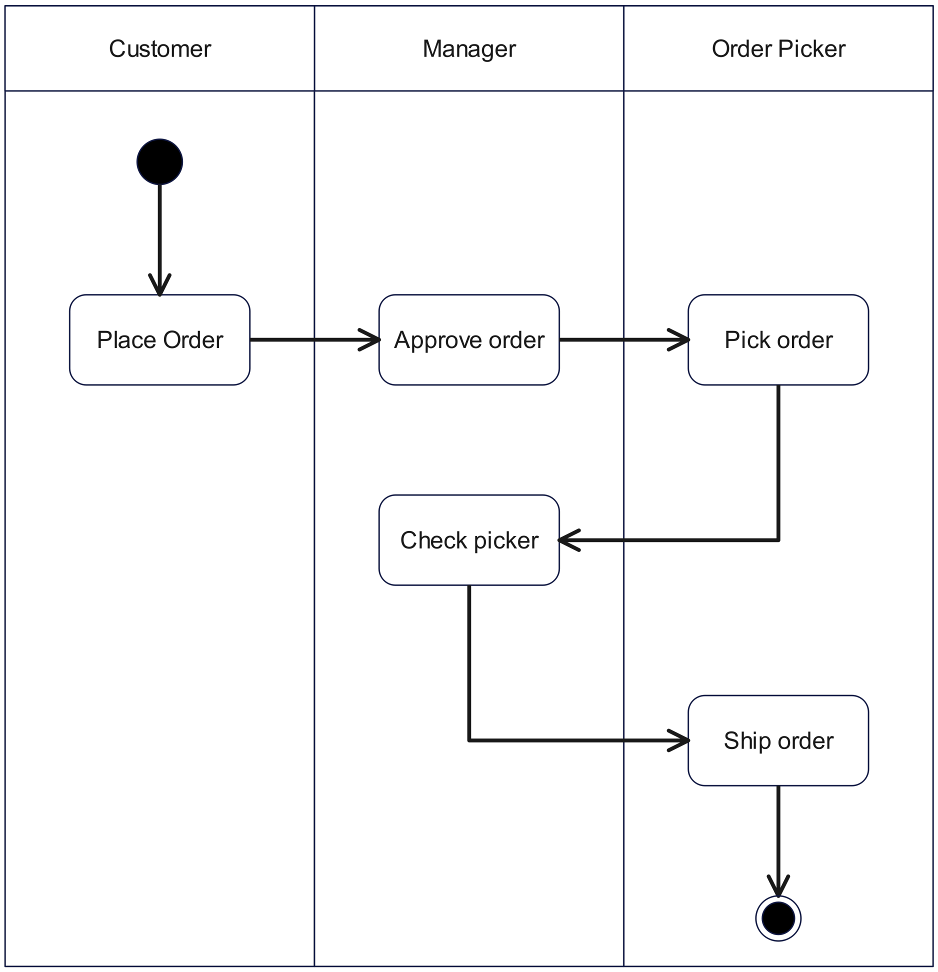
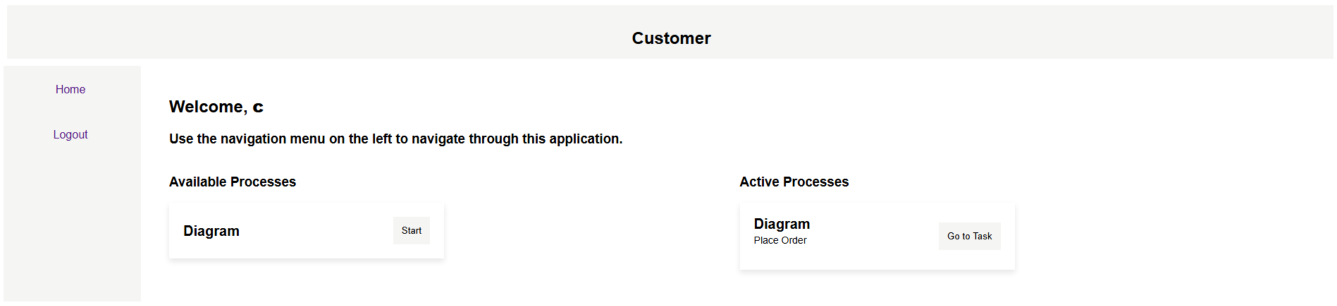


Figure 13: Process order case study



Customer

Welcome, c

Use the navigation menu on the left to navigate through this application.

Available Processes

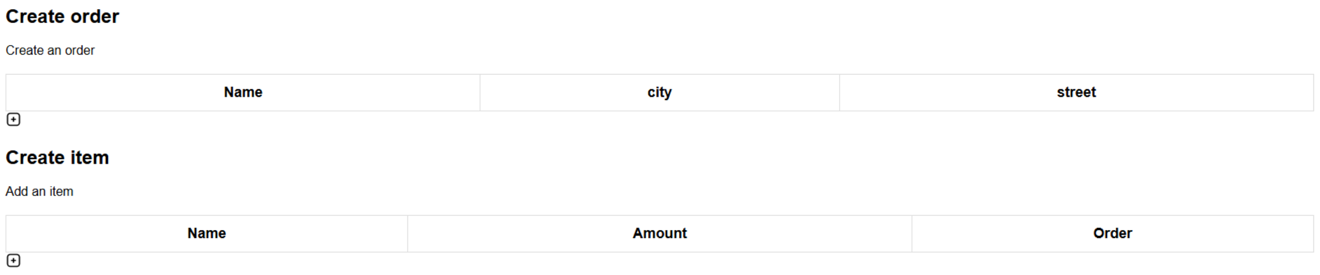
Diagram

Active Processes

Diagram
Place Order

Figure 14: Case study 1 Home page

Figure 15: Case study 1 Complete activity step button



Create order

Create an order

Name	city	street

Create item

Add an item

Name	Amount	Order

Figure 16: Case study 1 Activity page

A.3.2 Case study 2

Case study 2 commit hash: 8cfe9beceeca81ac0d73836cf89c184d6191164e

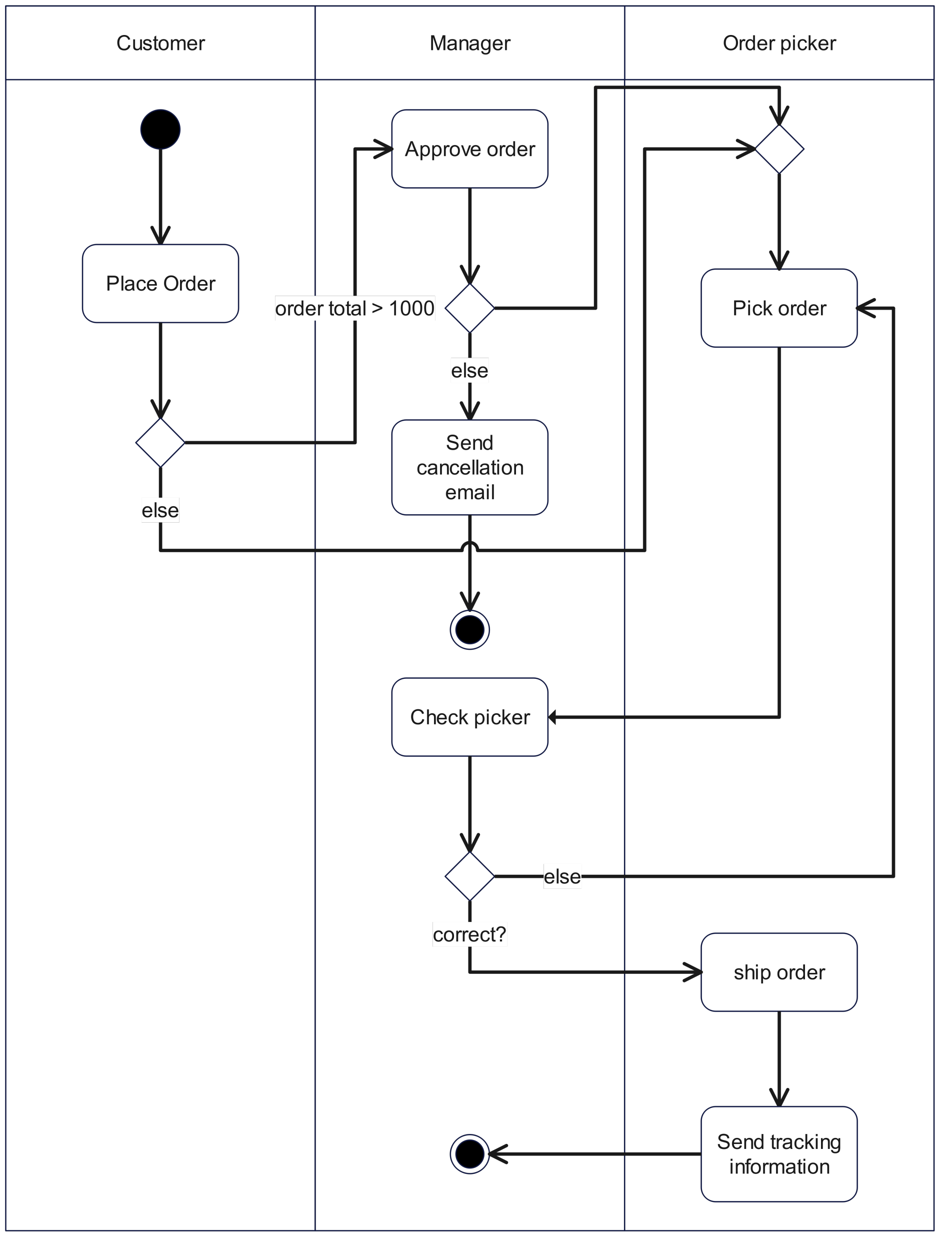


Figure 17: Process order case study

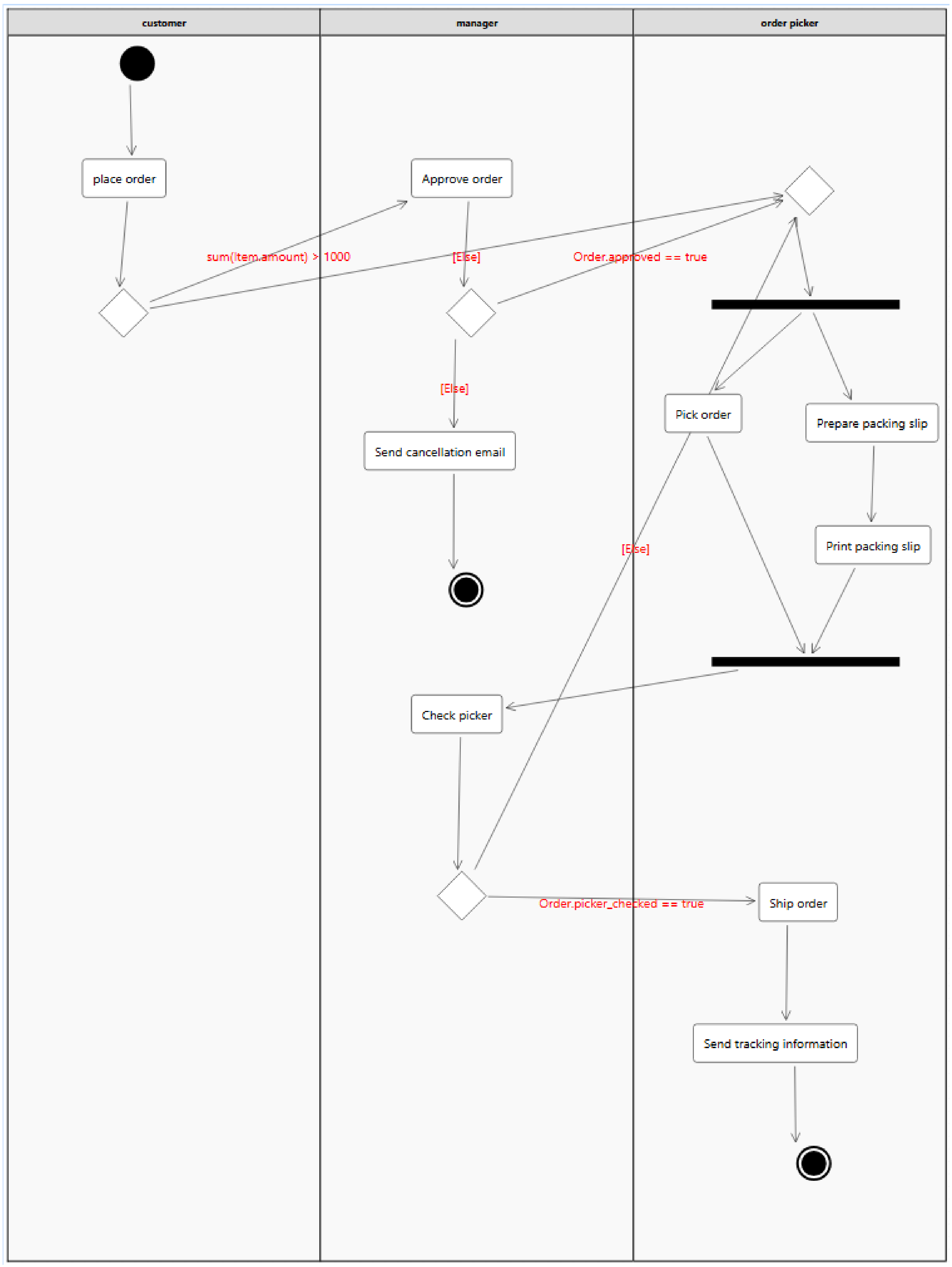


Figure 18: Diagram lacking edges and corners in control flow

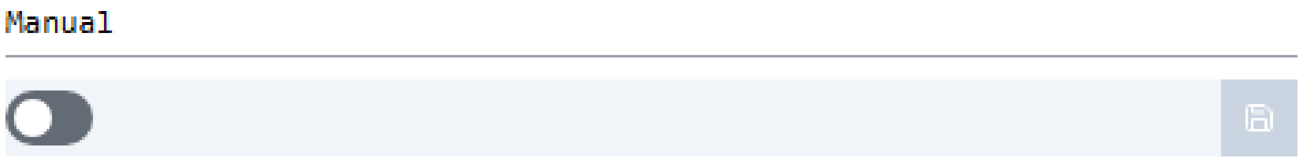
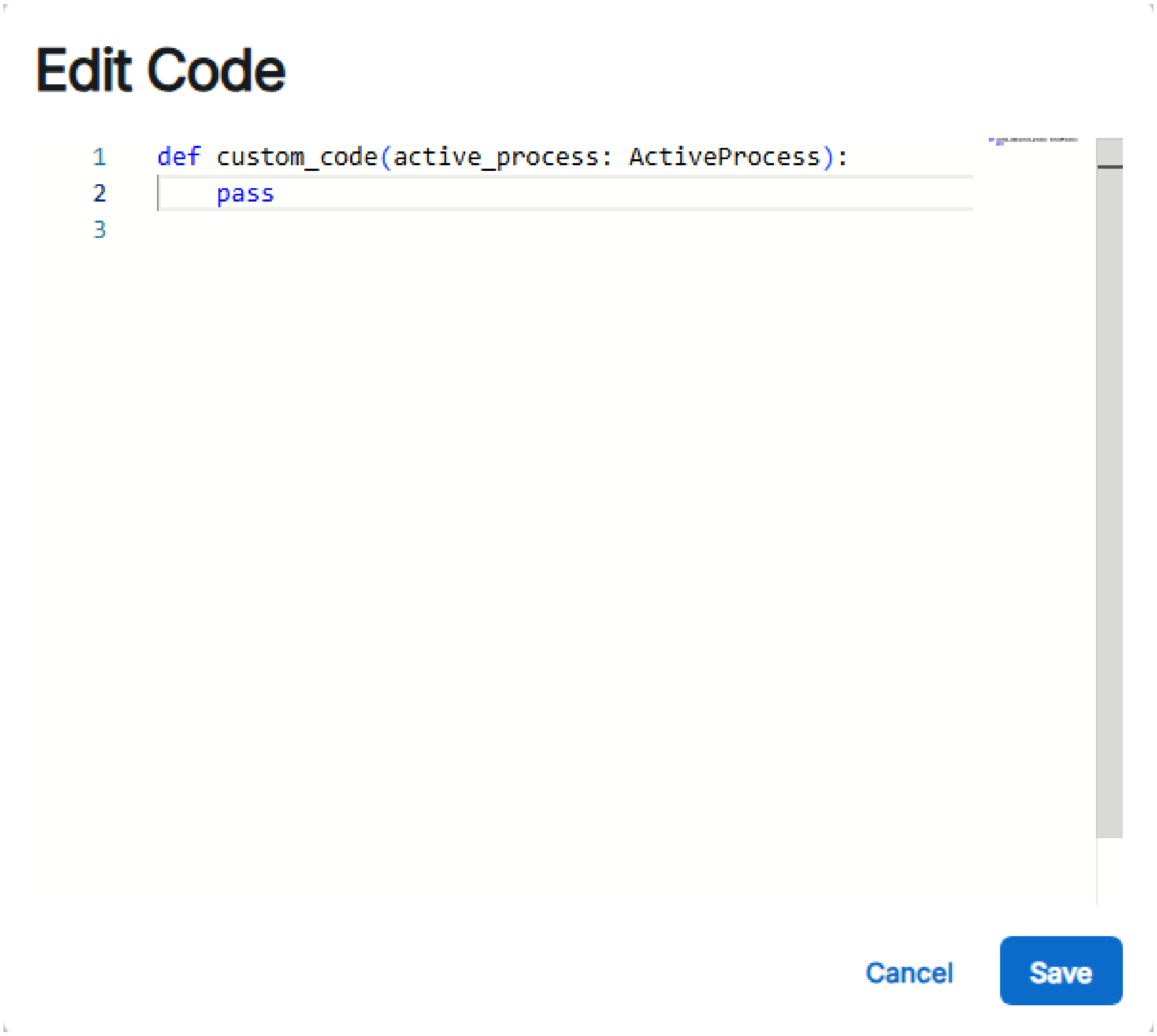


Figure 19: Page type label



```
def custom_code(active_process: ActiveProcess):
    pass
```

Figure 20: Case study 1 Studio code editor

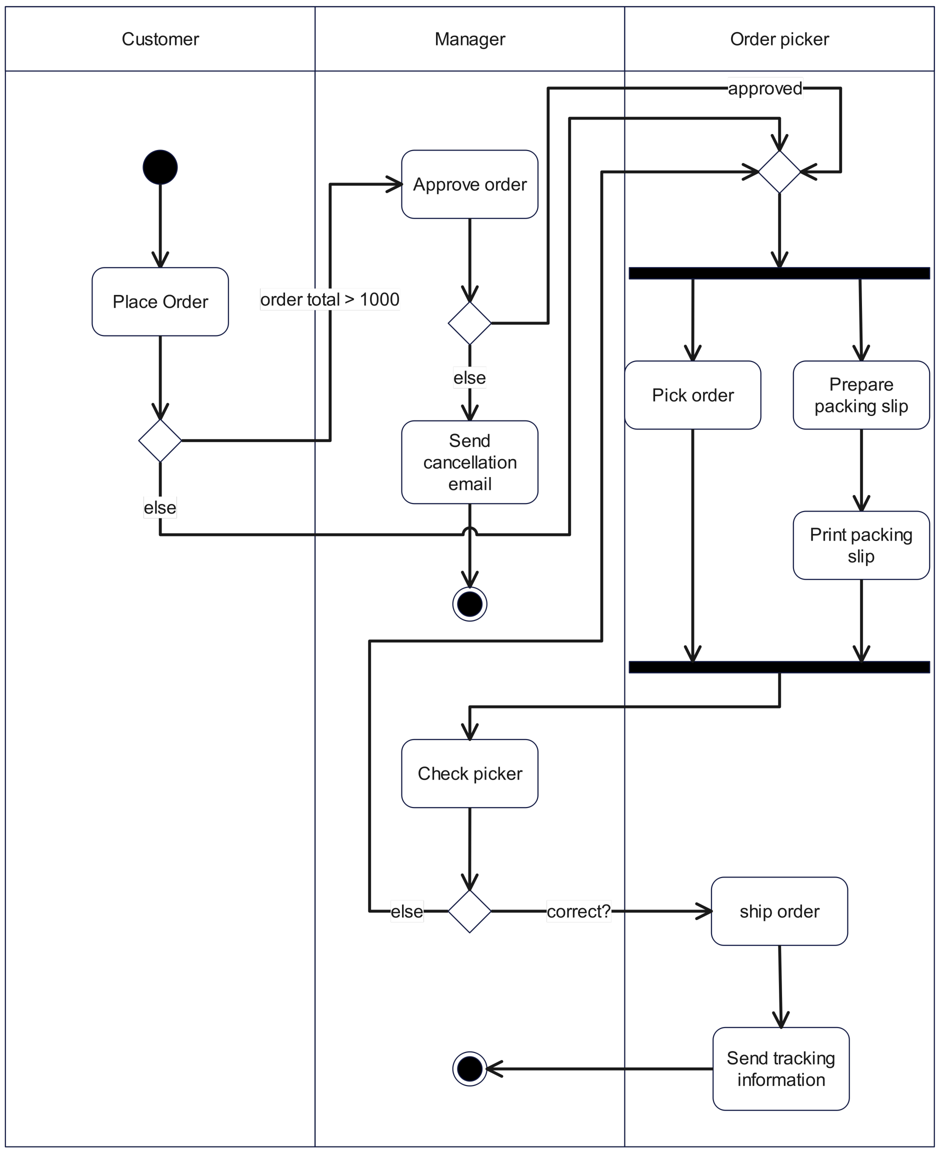


Figure 21: Process order case study

A.4 Interviews

A.4.1 Interview 1

Prototype diagrams

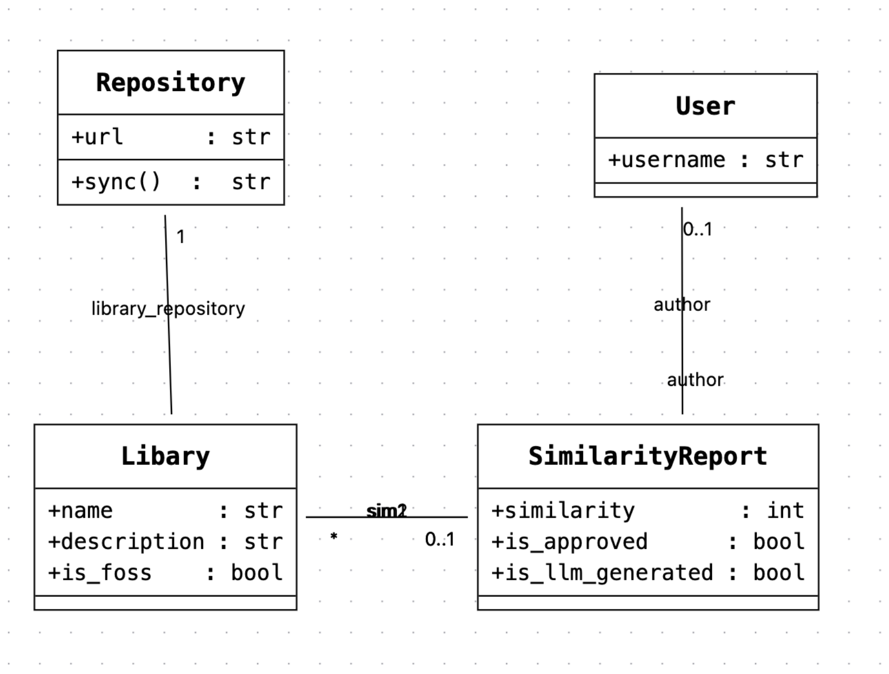


Figure 22: Interview 1 class diagram

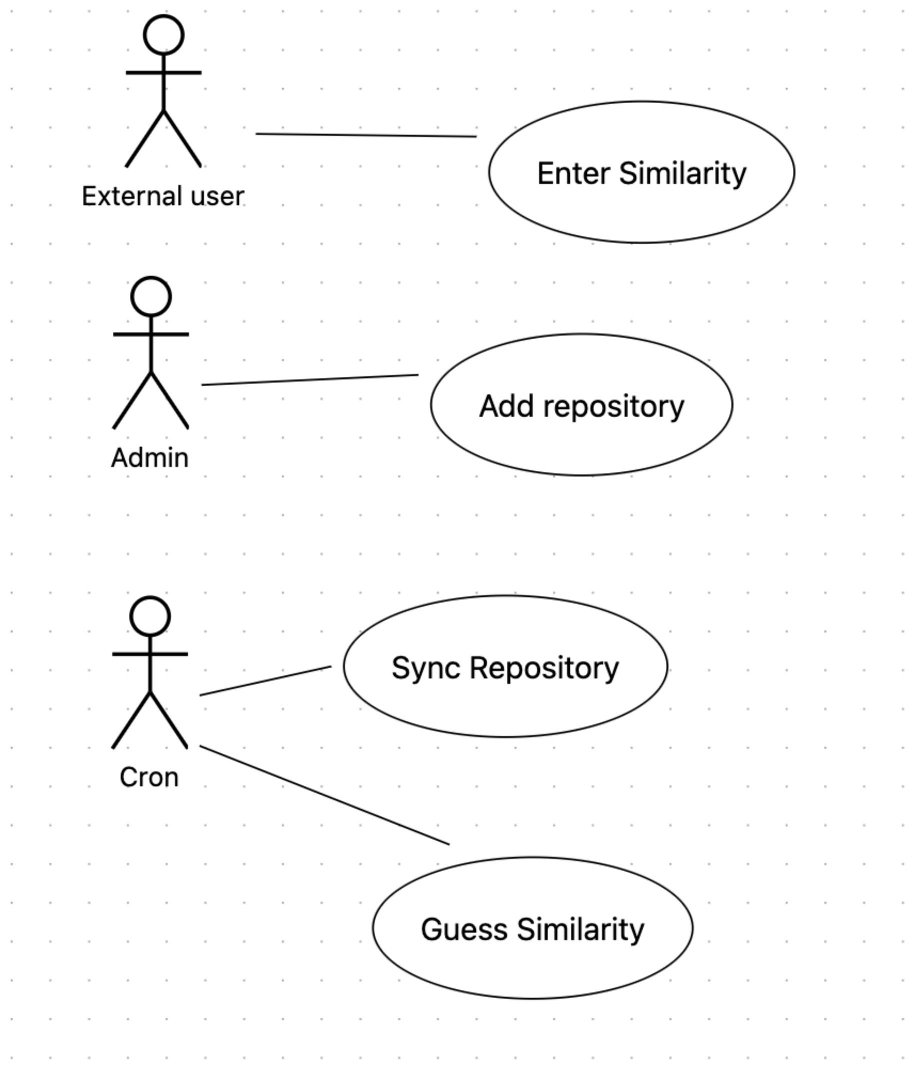


Figure 23: Interview 1 use case diagram

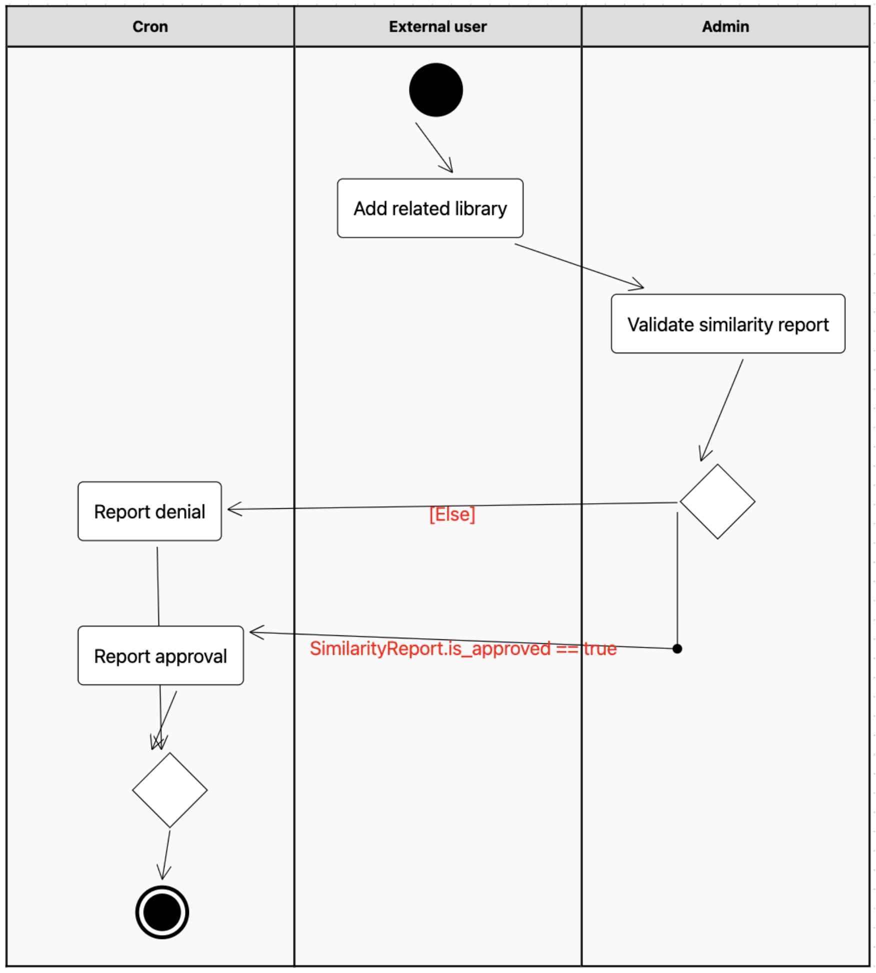


Figure 24: Interview 1 activity diagram

Question & Answers

Were all the tools available to model the behavior and workflow as you envisioned? (In particular, the activity diagram)
Something that is missing, which is quite vital, is the ability to start an activity based on a schedule (cron-job). This would also allow for background jobs to be added which is vital for every system. You should also be able to link and make additions to the Account class which is created by default.

Did you find that the interface design tools offered enough options to create an activity interface? If not, what would you like to see added?
The following additions should be made:

1. Foreign keys should be made editable or not in the section components.

2. The order of the section components can not be changed after adding them, without having to remove them all.
3. It should be able to create section components from the page creation tab. This would reduce the amount of navigation required.
4. Generating the interfaces fees disconnected. Giving a preview of how it is going to look gives a clear feedback to the user of the effect of their change on the application.

Which parts of the design process did you find clear, and which were confusing?
The design studio was sufficient to get the job done, however required significant fine-tuning. These include, but are not limited to, things like:

1. Edges not lining up correctly.
2. When creating a node they appear on a different location.
3. When connecting a node to a swimlane they move to a different location.
4. Nodes have bad default selections. For instance when creating an action node you also have to select the type which can only be action.

The navigation was confusing due to the presence of too many navigation tabs, which made it difficult to find the desired options quickly. A navigation feature that made it more clear was the order of the diagrams, interface, prototypes, releases tabs which could be seen as a progress bar. Creating custom code was also very difficult due to the lack of context. A suggestion was made to create an example project or to generate this code using LLMs to provide this context.

Does the generated prototype match your expectations based on the model? (Which points do or do not?)
Yes the backend functionality of the prototype did, however the looks did not.

Was the workflow (as represented in the activity diagram) clear and usable during interaction with the prototype?
Yes it was, however I expected more descriptive text at each activity.

Which parts of the prototype did you find worked well, and which worked less well? Why?
You were able to link new data instances to unrelated data instances within the project. For example, adding new items to an order that belongs to someone else.

What type of users do you envision using this tool? (Who would benefit most from using it?)

Mostly on bureaucratic management level, where process is more important then comfort. An example of this is software development by the government. With 25 year of experience (smaller companies) I have not encountered a project which was completely reliant on UML diagrams and thus do not see it being used there.

What would you like to see added or changed in this tool?
All code editors should be replaced by LLM descriptions to have them create this. This would reduce the need for context when creating this code, since only the LLM would need this. Diagrams should also be generated by an LLM, enabling the user to focus on the details. Currently it takes the user a lot of time to work out the core elements. The tool should also be able to change existing projects, as requirements often change after the first deployment.

Would you use this project to create prototypes at the start of a real project?
No not in the current state, since it is still to young. I would consider using this if it was able to operate outside of greenfield projects.

A.4.2 Interview 2

Prototype diagrams

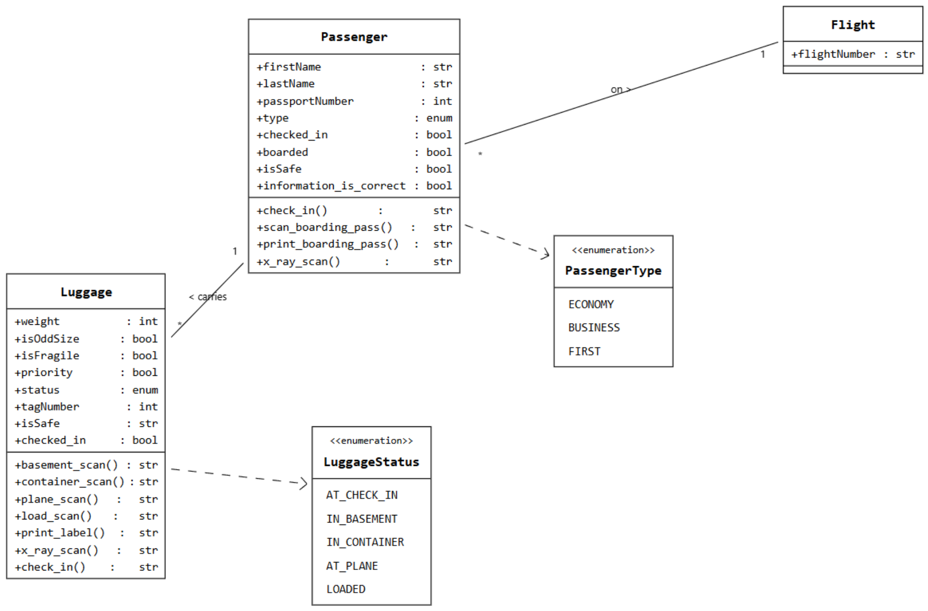


Figure 25: Interview 2 class diagram

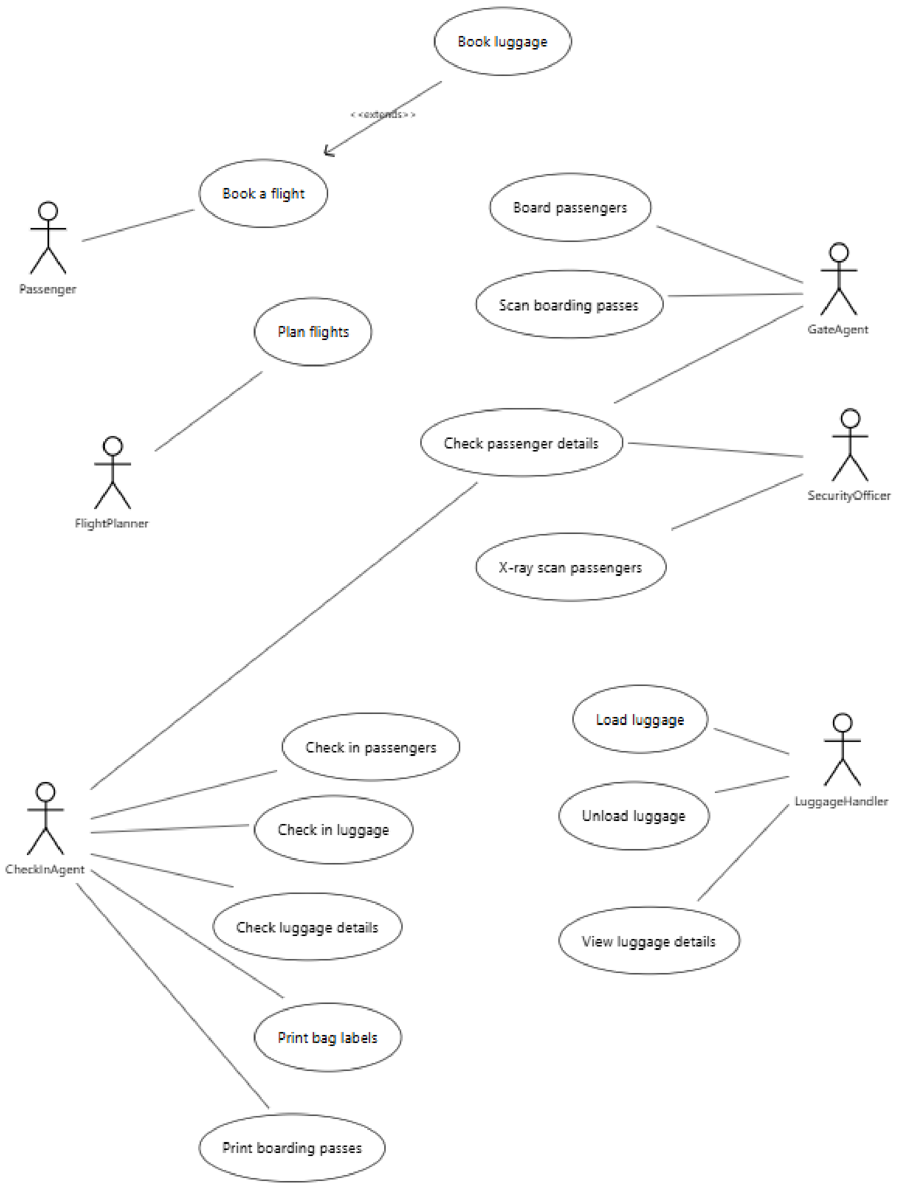


Figure 26: Interview 2 use case diagram

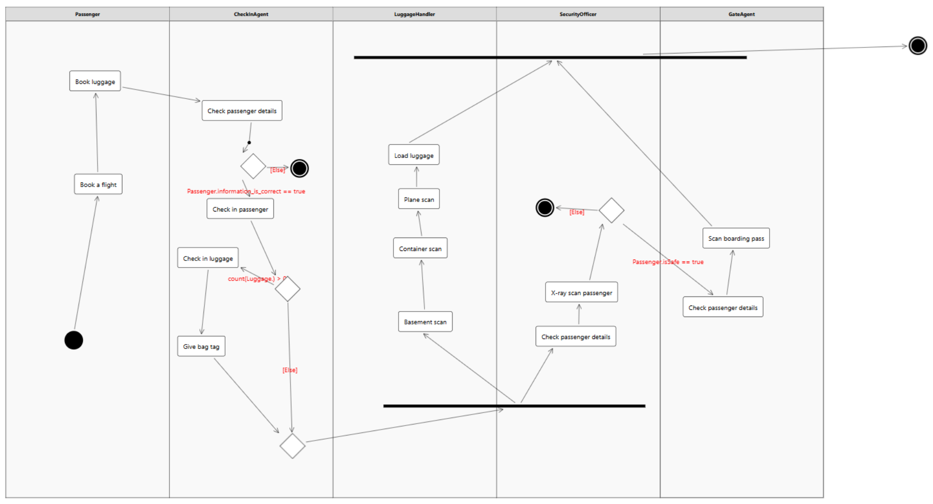


Figure 27: Interview 2 activity diagram

Question & Answers

Were all the tools available to model the behavior and workflow as you envisioned? (In particular, the activity diagram)
No not all, the workflow engine currently only works with data created during the process. First I wanted to split different activities between diagrams, such as the booking of the flight and checking in/departing. This however was not possible since the workflow engine can't know the context of other processes. Object flow should be added to enable processes to inherit context from other processes. Next to this I would say it mainly requires refinement of the current design process. One of the most important change would be to add nodes to swimlanes automatically when using the right mouse button add node action on an existing swimlane.

Did you find that the interface design tools offered enough options to create an activity interface? If not, what would you like to see added?
I was missing an input that filters the foreign key queryset when creating/updating an instance. Currently it shows all foreign keys, but sometimes this should only display the foreign keys that are linked to the active process. The button to execute custom code currently displays the name of the method. I would like for the user to be able to select an svg from a standard selection or be able to upload one myself.

Which parts of the design process did you find clear, and which were confusing?
I found the difference between a UI activity and manual activity when creating an action node confusing. This requires an additional explanation in the UI.

Does the generated prototype match your expectations based on the model? (Which points do or do not?)

Yes it did, it followed the activity diagram well.

Was the workflow (as represented in the activity diagram) clear and usable during interaction with the prototype?
yes it was clear.

What type of users do you envision using this tool? (Who would benefit most from using it?)
Mainly more technical users, who know the process of code generation. In the current state is it not possible for a "layperson" to use it.

What would you like to see added or changed in this tool?
Creating an activity diagrams involves a lot of steps, this should be improved. This can be done by using defaults or a mapping a human sketch to the current expected model.

Would you use this project to create prototypes at the start of a real project?
Within the context of AI4MDE I would. For my personal project i wouldn't. For this to happen I would require the front- and backend to be split. This should result in an individual Django API and a frontend using a javascript framework.

A.4.3 Interview 3

Prototype diagrams

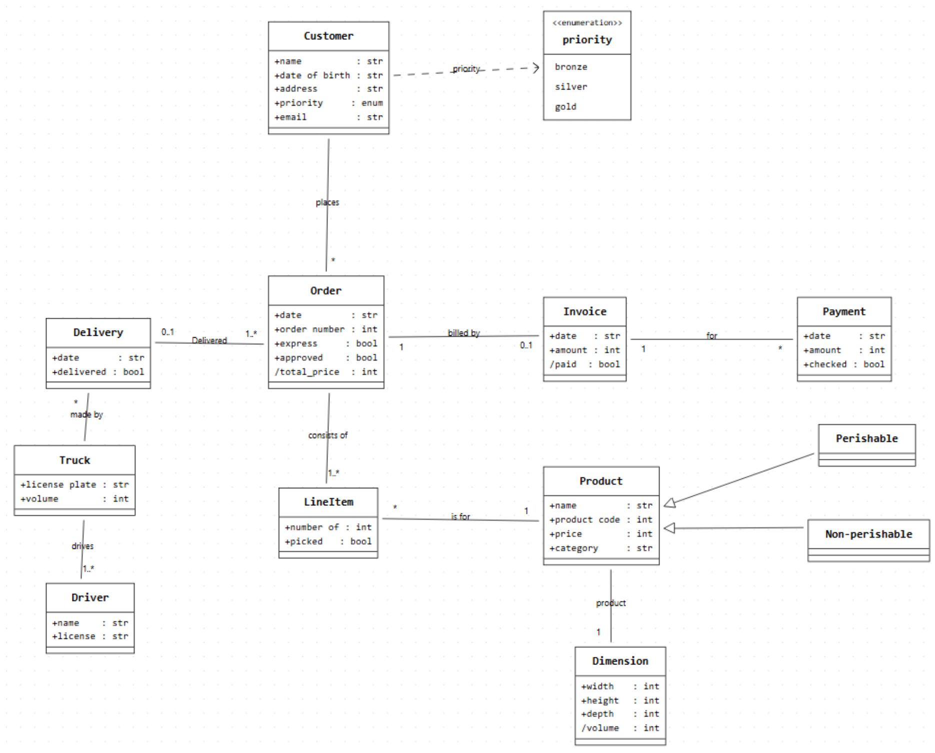


Figure 28: Interview 3 class diagram

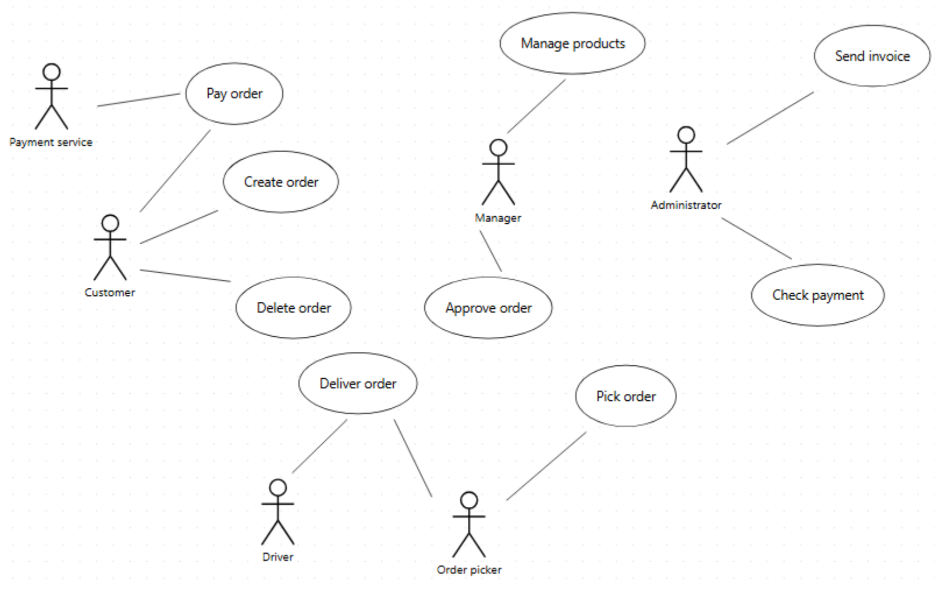


Figure 29: Interview 3 use case diagram

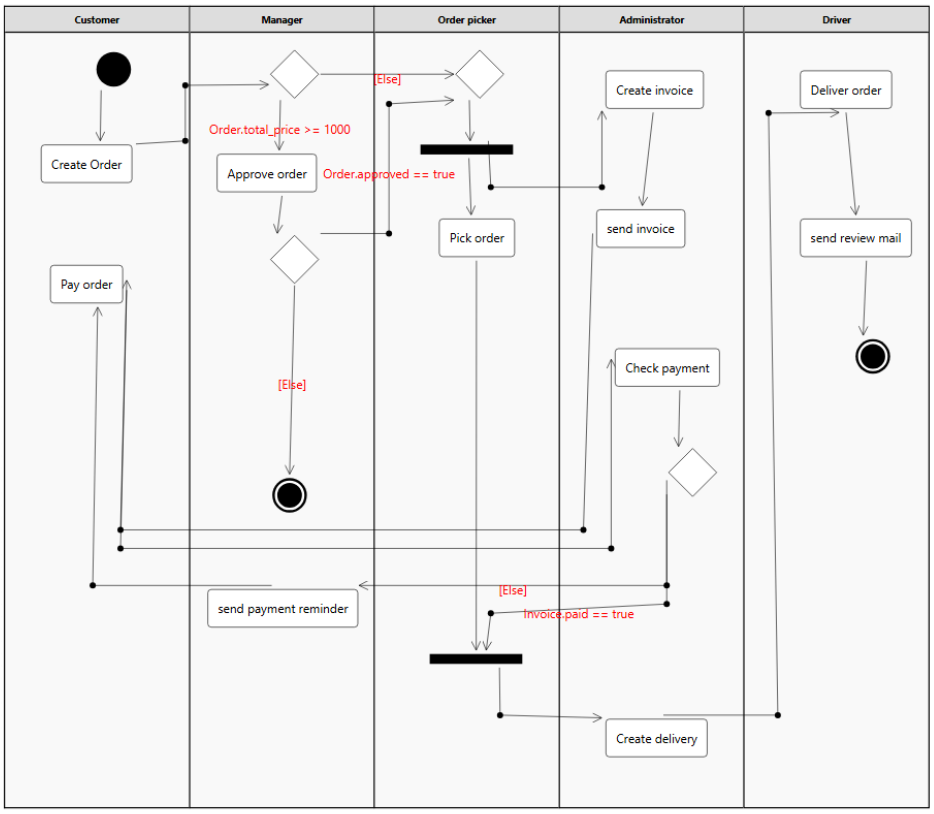


Figure 30: Interview 3 activity diagram

Question & Answers

Were all the tools available to model the behavior and workflow as you envisioned? (In particular, the activity diagram)

No, I was missing the ability to have multiple control flows pointing to a single action node. In this case the action can only be done when all control flows have reached it. The conditions from a decision node should also be extended. The user should be able to compare conditions to other entries, not just to a single threshold. You should also be able to chain conditions using logical operators such as AND, OR, currently you are only able to create one.

Did you find that the interface design tools offered enough options to create an activity interface? If not, what would you like to see added?

Designing the interface required a lot of back and forth, between making categories and section components. The user should be able to create section components and categories from within the page creation tab. To decrease the amount of navigation further you should add the ability to create pages for an activity from within the activity diagram modeller.

Which parts of the design process did you find clear, and which were confusing?

It was generally good, it does however require UI improvements to make the process more fluent. Creating custom code is a bit confusing. A solution to this is the usage of 2 pass generation. The user would define the custom code in natural language. When the user generates the prototype, first the Jinja2 templating is used to create everything except the custom code. After this the custom code is filled in using Large Language Models using the previously generated code as context.

Does the generated prototype match your expectations based on the model? (Which points do or do not?)
Yes absolutely. The only difficulty I encountered was identifying the cause of generation failures due to mistakes in designing the activity model. These errors should be communicated to the user more clearly. This could possibly be done using a data validation layer between the design studio and the generator.

Was the workflow (as represented in the activity diagram) clear and usable during interaction with the prototype?
Yes it was.

What type of users do you envision using this tool? (Who would benefit most from using it?)
Academic researches to experiment with AI functionality within an open source MDE environment. I would also like to use it in an educational setting with students.

What would you like to see added or changed in this tool?
Designing costs a lot of time. It took me way longer then expected, about 4 hours, to finish the design after our first session. The user experience should also be improved a lot to make it more usable.

Would you use this project to create prototypes at the start of a real project?
Yes in context of a university course I would.