Coding by Design: GPT-4 Empowers Agile Model Driven Development

Ahmed R. Sadik[a], Sebastian Brulin[b] and Markus Olhofer[c]
Honda Research Institute Europe, Carl-Legien-Strasse 30, Offenbach am Main, Germany

Keywords: GPT-4, Auto-Generated Code, AI-Empowered, Model Driven Development, Ontology-Constrained Class Diagram, Object Constraint Language, Cyclomatic Complexity.

Abstract: Generating code from a natural language using Large Language Models (LLMs) such as ChatGPT, seems groundbreaking. Yet, with more extensive use, it's evident that this approach has its own limitations. The inherent ambiguity of natural language proposes challenges to auto-generate synergistically structured artifacts that can be deployed. Model Driven Development (MDD) is therefore being highlighted in this research as a proper approach to overcome these challenges. Accordingly, we introduced an Agile Model-Driven Development (AMDD) approach that enhances code auto-generation using OpenAI's GPT-4. Our work emphasizes "Agility" as a significant contribution to the current MDD approach, particularly when the model undergoes changes or needs deployment in a different programming language. Thus, we presented a case-study showcasing a multi-agent simulation system of an Unmanned Vehicle Fleet (UVF). In the first and second layer of our proposed approach, we modelled the structural and behavioural aspects of the case-study using Unified Modeling Language (UML). In the next layer, we introduced two sets of meta-modelling constraints that minimize the model ambiguity. Object Constraints Language (OCL) is applied to fine-tune the code constructions details, while FIPA ontology is used to shape the communication semantics. Ultimately, GPT-4 is used to auto-generate code from the model in both Java and Python. The Java code is deployed within the JADE framework, while the Python code is deployed in PADE framework. Concluding our research, we engaged in a comprehensive evaluation of the generated code. From a behavioural standpoint, the auto-generated code not only aligned with the expected UML sequence diagram, but also added new behaviours that improved the interaction among the classes. Structurally, we compared the complexity of code derived from UML diagrams constrained solely by OCL to that influenced by both OCL and FIPA-ontology. Results showed that ontology-constrained model produced inherently more intricate code, however it remains manageable. Thus, other constraints can still be added to the model without passing the complexity high risk threshold.

1 INTRODUCTION

In the era of Large Language Models (LLMs), Model-Driven Development (MDD) is gaining momentum as a promising way to enhance software engineering (Sadik, Ceravola, et al., 2023). Therefore, our study presents an Agile Model Driven Development (AMDD) approach that utilizes LLMs to auto-generate complete, deployment-ready software artifacts. This method streamlines MDD, avoiding the need to constantly update code generators with each model change. The resulting software is not just intricate but also designed to fulfill specific requirements and functionalities. MDD uses formal language models such as Unified Modeling Language (UML) to generate code. This automation aligns design with implementation, reducing errors and accelerating market deployment. UML primarily focus on structural aspects, often overlooking domain-specific rules and constraints (Sarkisian et al., 2022). Yet, the Object Constraint Language (OCL) and domain-specific ontology languages like FIPA-ontology address this gap. They ensure model

[a] https://orcid.org/0000-0001-8291-2211
[b] https://orcid.org/0000-0002-9710-6877
[c] https://orcid.org/0000-0002-3062-3829

Sadik, A., Brulin, S. and Olhofer, M.
Coding by Design: GPT-4 Empowers Agile Model Driven Development.
DOI: 10.5220/0012356100003645
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 149-156
ISBN: 978-989-758-682-8; ISSN: 2184-4348
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

integrity and consistency by defining detailed constraints and shared knowledge understanding. Integrating class diagrams with OCL and ontologies could revolutionize code generation, producing software that reflects both foundational design and domain expertise (Kapferer & Zimmermann, 2020).

Evaluating auto-generated code is crucial for assessing its quality. Traditional qualitative criteria like testability and reliability are often subjective. Our research adopts more objective, quantifiable metrics, focusing on structural integrity. We use cyclomatic complexity to evaluate structural soundness and conduct comparative analyses to ensure the code behaves as expected across different languages and aligns with the model's intended behaviour (Ahmad et al., 2023). This paper outlines our study, starting with a detailed problem statement in Section 2 about challenges in the current MDD approach. Section 3 details our four-layered AMDD approach, and Section 4 applies it to model a UVF and Mission Control Center (MCC). Using UML diagrams, OCL, and FIPA ontology, leading to auto-generated Java and Python code for simulations. Section 5 evaluates the model and generated code through assessing its structural complexity and testing its deployment behaviour. Finally, Section 6 concludes with findings, implications, and future research directions.

2 PROBLEM STATEMENT

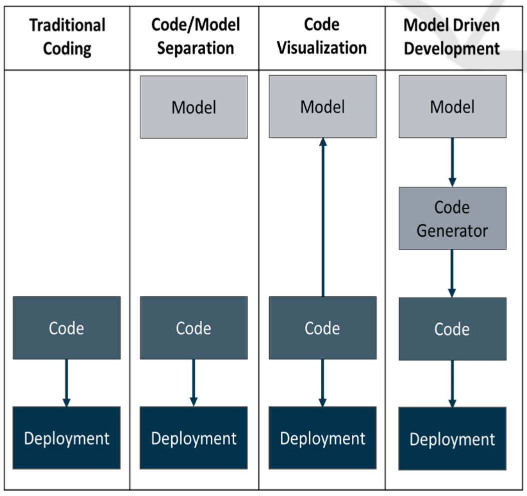

Figure 1: Traditional coding vs MDD.

Natural language, with its inherent ambiguity, presents significant challenges in both human and machine comprehension. This is particularly evident when using LLMs such as ChatGPT to auto-generate software artifacts. The uncertain and open-ended nature of natural language prompts often leads to the production of flawed code, a problem that becomes more critical with the increased complexity and multidimensionality of the software being developed. Yet, MDD aims to address these challenges by offering a higher level of software abstraction. In MDD, high-level models are the core artifacts, serving as the blueprint from which final applications are generated (Sadik & Goerick, 2021).

The difference between traditional coding and MDD code generation can be understood through Figure 1. Traditional coding directly translates software functionalities into code, suitable for smaller and straightforward features. Debugging, testing, and maintenance are conducted at the code level. In contrast, MDD uses models to abstractly understand the system, separate from the coding process. In traditional model-and-code separation approaches, models are often discarded post-coding due to the high maintenance cost. In code visualization, models are created after software development to understand program functionalities or integrating elements into models. However, these models are typically not used for implementation, debugging, or testing. MDD differs significantly in that models are the primary development artifacts, replacing source code (Kelly & Tolvanen, 2008). Tools like Eclipse Papyrus, MagicDraw, Enterprise Architect, and IBM Rational Rhapsody facilitate this process by generating target code from these models, thus hiding underlying complexities (David et al., 2023).

However, while MDD promises efficiency and precision, it also introduces challenges. For instance, changes in deployment language or significant model updates necessitate extensive modifications to the code generators, affecting the agility of the development process. The task of creating and maintaining the code generators is resource-intensive and complicated by the need to tailor them to each programming language. This is where the potential of LLMs like ChatGPT as universal code generators becomes intriguing (Chen et al., 2021). Our study points to a significant issue that MDD, despite its structured approach, hasn't fully harnessed LLM capabilities in code auto-generation. This gap results in a misalignment, making the current MDD approach less suited for agile software development environments, where quick adaptability and flexibility are key. Thus, while MDD overcomes some limitations of natural language, it still faces challenges in fully integrating with advanced AI capabilities for efficient and agile software development.

3 PROPOSED APPROACH

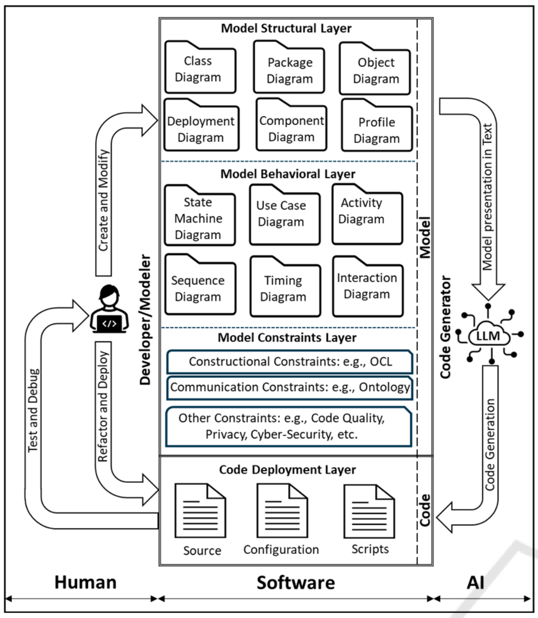

Figure 2: Proposed AMDD approach.

Addressing the challenge outlined in our problem statement, our AMDD approach requires ChatGPT to thoroughly understand the model and its associated views. Recognizing ChatGPT's text-based processing capabilities, we utilized PlantUML to convert visual UML diagrams into a text format that can be easily integrated into ChatGPT prompts. In this methodology, illustrated in Figure 2, the modeler first establishes different model layers, including structural, behavioural, and constraints. The structural layer encompasses diagrams that display the static aspects of the model, like class diagrams detailing object relationships and hierarchies, package diagrams grouping objects to highlight dependencies, and component diagrams breaking down system functionality. Deployment diagrams depict the physical layout, object diagrams provide runtime snapshots, and profile diagrams adapt UML models to specific platforms. The behavioural layer, through sequence, activity, interaction, timing, use-case, and state diagrams, models system operations, interactions, and the lifecycle of entities, offering a comprehensive view of how the system functions and interacts with users and other systems.

Although, the structural and behavioural diagrams provide a holistic architectural view, they lack the rules that regulate the model semantics. Accordingly, in this research we propose the constraints layer to fine-tune the model architecture, by explicitly including its meta-values that cannot be expressed by UML notations. OCL is used to restrict the code construction details of the structural and behavioural layers, by specifying invariants on classes and stereotypes, describing pre- and post-conditions on method and states, and limiting the parameters' values. Furthermore, communication constraints can be defined using the proper formal method such as an ontology language to express the communication semantics, that is necessary to share knowledge among the software artifacts.

Ultimately, in the deployment layer, ChatGPT, based on the GPT-4 is employed to generate code. The choice of GPT-4 over GPT-3.5 is due to its superior reasoning capabilities, a vital feature for our approach. The LLM must comprehend the model semantics encapsulated in the constraints layer to integrate them into the generated code. Furthermore, post auto-generation of code using ChatGPT, it is crucial for the modeler to deploy the generated code and verify its operationality. However, it's essential to acknowledge that ChatGPT's code generation capabilities are continually evolving and may not be flawless (Dong et al., 2023). Thus, the possibility of encountering bugs during the code deployment is anticipated. It becomes imperative for the modeler to address these bugs, possibly with assistance from ChatGPT, and iteratively run the code until it successfully accomplishes its intended purpose.

4 CASE-STUDY MODEL

The chosen use-case involves a UVF, comprising various types of UVs that undertake specific missions and are coordinated by an MCC involving a human operator (Sadik, Bolder, et al., 2023). This case-study is intentionally distributed, enabling it to be modelled and simulated as a MAS (Brulin & Olhofer, 2023). The MAS often encompasses a high complexity level, as each entity is represented as an agent and must communicate and share information with other entities (i.e., agents) to achieve a common goal (i.e., the fleet mission). To avoid overwhelming the reader with the intricacies of the MAS, in the following sections, only the essential model that facilitates an understanding of the MAS operation concept views (Sadik & Urban, 2018).

4.1 Model Structural Layer

In the model structure layer, the class diagram is pivotal, representing every entity in the case-study as an agent class, as illustrated in Figure 3. The operator agent models the human operator with actions like sending mission briefs and receiving performance data. The MCC agent, with attributes like MCC-ID, coordinates the mission and monitors the fleet. The UVF-Manager agent handles fleet management, including task delegation and performance tracking. The Unmanned Vehicle (UV) agent serves as a generic class for UVs, with specific subclasses like UAV, UGV, and USV for different vehicle types. This class diagram elucidates the complex internal dynamics of each agent, their attributes, operations, and interrelations, including various types of relationships such as composition, aggregation, and inheritance, and establishes the cardinality between agents.

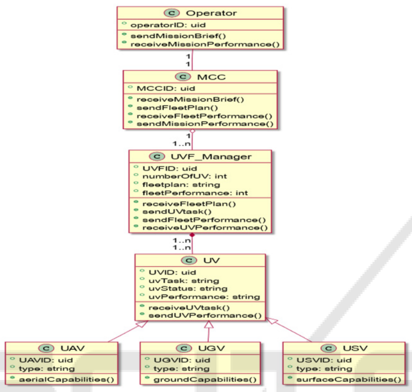

Figure 3: Case-study class diagram.

4.2 Model Behavioural Layer

To maintain brevity and keep the article focused, the article will only explain two views which are the activity and state diagrams, as they are the most important behavioral views to understand the case-study model. The activity diagram refines and complements the class diagram by meticulously detailing aspects such as synchronization, parallel execution, and conditional flows, which are indispensable for effectively achieving the mission goals. In contrast, the state diagram offers a microscopic perspective, unveiling the life cycle of the agents' class within the model and illuminating how they coordinate and respond to realize the overarching mission objectives.

The activity diagram in Figure elaborates the interplay of information and task flows within the agents. It illustrates the orchestration of processes and the sequence in which tasks are allocated, carried out, and assessed, providing an understanding of the MAS temporal and logical dynamics. Thus, the interaction begins when the operator agent sends the mission-brief to the MCC agent. The latter transforms the brief into a plan and conveys it to the UVF-manager, which, in turn, assigns the tasks to the available UVs. Subsequently, the UVF-manager agent awaits the completion of tasks by each UV and collates their performance, which is instrumental in assessing the overall UVF performance. This consolidated performance is relayed to the MCC, translated into mission-performance, and communicated back to the operator agent.

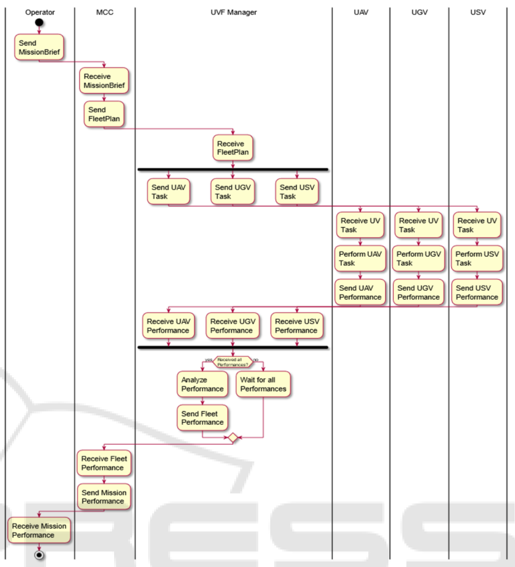

Figure 4: Case-study activity diagram.

Furthermore, the state diagram offers an in-depth view of the internal behaviour of each agent, revealing the transitions and actions triggered by various events. The operator, MCC, and UVF-manager are depicted with a simple two-state diagram indicating 'busy' or 'free' states. However, the UV requires a more complex state machine to evaluate task performance. Figure outlines the UV states: 'Available' (registered or unregistered), 'Unavailable' (out of service), 'Unregistered' (available but not yet registered), 'Registered' (controlled or uncontrolled), 'Uncontrolled' (registered without a mission), and 'Controlled' (registered with an assigned mission).

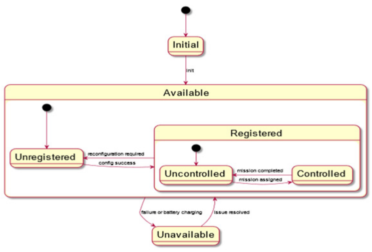

Figure 5: UV agent state diagram.

4.3 Model Constraints Layer

The constraints layer in the proposed AMDD approach acts as a meta-model that encapsulates all aspects of the technical requirements that cannot be formalized in the structural and behavioral layers. In the following sections we will discuss in detail the different types of meta-model constraints that have been considered within the case-study modeling.

4.3.1 Construction Constraints

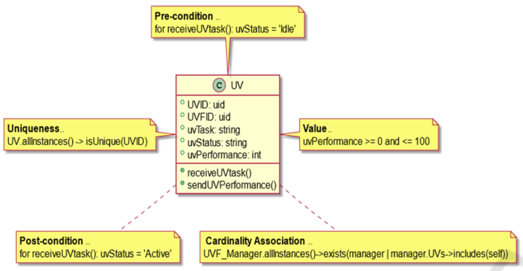
Pre-condition ..
for receiveUVtask(): uvStatus = 'Idle'
UV
UVID: uid
UVFID: uid
uvTask: string
uvStatus: string
uvPerformance: int
receiveUVtask()
sendUVPerformance()
Uniqueness..
UV.allInstances() -> isUnique(UVID)
Value..
uvPerformance >= 0 and <= 100
Post-condition ..
for receiveUVtask(): uvStatus = 'Active'
Cardinality Association ..
UVF_Manager.allInstances()->exists(manager | manager.UVs->includes(self))

Figure 6: UV agent construction constraints in OCL.

The OCL is a declarative language that complements UML by defining rules applicable to classes within a model. Its integration as construction constraints in UML diagrams is crucial for enhancing model refinement and clarity. OCL effectively addresses ambiguities in model construction, ensuring precision, particularly beneficial when generating deployed code directly from model views. This precision guarantees a more accurate and seamless transition from model to code. In our study, we demonstrate the application of OCL by imposing five types of constraints on all agent classes, as shown in Figure for the UV agent class. These constraints include 'Uniqueness' for distinct agent identifiers, 'Cardinality' for managing agent associations (e.g., a UVF-manager linked to multiple UVs), 'Value' for restricting class values within certain ranges (like UV performance between 0 and 100), 'Pre-condition' to ensure agents are in the right state for transitions (such as a UV accepting tasks only when idle), and 'Post-condition' for mandatory state changes after transitions, like changing a UV's status to 'Active' after receiving a task.

4.3.2 Communication Constraints

OCL excels in setting constraints for UML model classes but falls short in managing inter-class communication, a critical aspect in Multi-Agent System (MAS). Addressing this, Java Agent DEvelopment (JADE) and Python Agent DEvelopment (PADE) utilize the FIPA-ontology communication language. FIPA-ontology enriches MAS with comprehensive interaction protocols for complex agent communication, an area where OCL is limited, as highlighted in our case study.

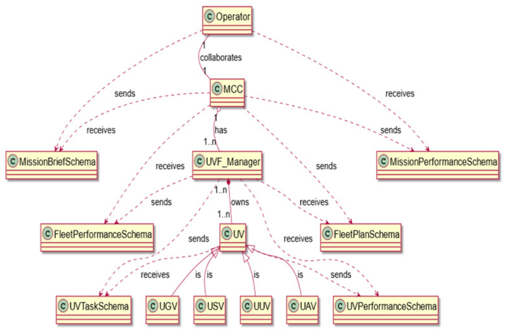

Figure 7: Case-study FIPA-ontology model.

In our MAS communication model, as depicted in Figure , FIPA-ontology defines a series of fixed schemas. The first set, the message communication schemas, encompasses various aspects of mission and agent management. This includes the Mission-Brief schema with details like mission-ID and status, Fleet-Plan outlining fleet specifics, UV-Task for individual task assignments, and UV-Performance for performance metrics. Additionally, it covers Fleet-Performance and Mission-Performance, each capturing specific performance-related metrics. The second set of schemas in FIPA-ontology are the predicates. These define the relationships between agent classes, encompassing aspects like inheritance (e.g., UAV as a type of UV), composition (e.g., MCC having a UVF-manager), aggregation (e.g., UVF-manager owning multiple UVs), and collaboration (e.g., operator working with MCC). Lastly, the third set in our model involves actions, which are operations that agents can perform, particularly on message schemas. This includes actions like sending and receiving data, exemplified by an operator agent sending a mission brief to the MCC or the MCC receiving it from the operator.

5 CODE EVALUATION

In the final step of our AMDD approach, we used GPT-4 to transform models into code, resulting in an average of four bugs per agent class, mostly due to missing library imports. Despite these issues, the corrected code was deployable. Our study prioritized evaluating the completeness of the auto-generated code over its correctness. We conducted two experiments: the first analyzed the behavior of the auto-generated code, and the second assessed its structure and complexity. This approach allowed us to thoroughly understand the code generation capabilities and limitations of GPT-4 within the AMDD framework.

5.1 Experiment 1: Behavioural Dynamic Analysis

The first experiment orchestrated the generation of two distinct deployments. The first is written in Java to run on JADE platform, while the second is written in Python to run in PADE. The goal of the experiment is to compare the code behaviour that run on JADE against the code that is running on PADE, to ensure the consistency of the system dynamic regardless of the execution language. Accordingly, the agent interaction behavior on JADE and PADE framework is observed. We found that the agents' behavior as captured by JADE Sniffer tool aligns with the plotted sequence diagram from PADE agent interaction, as shown in Figure.

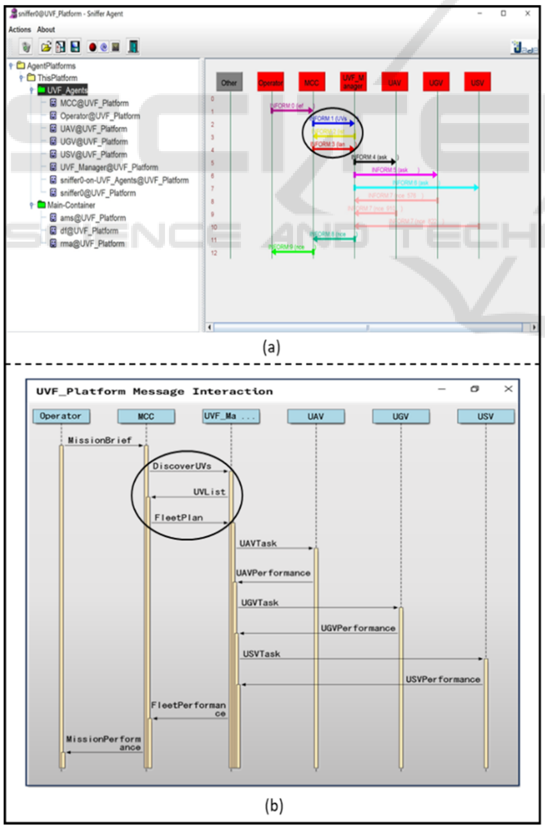

Figure 8: JADE vs PADE Sequence diagram.

In both sequence diagrams in Figure, the interaction starts when the operator transmitting a mission-brief to the MCC. By receiving this message, the MCC asks the UVF-manager to identify available UVs. Upon obtaining a list of accessible UVs, the MCC devises a fleet-plan and conveys it to the UVF-manager. After this, the UVF-manager dispatches specific tasks to the available UVs. Each UV, upon task completion, relays performance data to the UVF-manager. Thus, the UVF-manager formulates a comprehensive fleet-performance metric, which is relayed back to the MCC. The MCC, in turn, evaluates this metric in congruence with the mission objectives, compiling a definitive mission-performance report. This report, the culmination of the entire operation, is ultimately returned to the operator.

Two important remarks have been noticed from comparing these two sequence diagrams in Figure with the original case-study activity diagram in Figure. First, we noticed that ChatGPT has enhanced the interaction by adding new behaviours to MCC agent and UVF-manager agent. This new behaviour can be seen when the MCC is sending DiscoverUVs message to the UVF-Manger agent and waiting the UVList before forming a FleetPlan, as logically the MCC needed to know what the available UVs resources are before planning them based on the mission-brief. This new interaction behaviour was not explicitly mentioned in the case-study activity diagram. The second remark is that the timing of interaction between the MCC and the UVs differ in JADE and PADE, most probably due to the difference in the state machine of each UV instance. This is a good indication that these UV state machine can emulate the operation of the agents.

5.2 Experiment 2: Structural Complexity Assessment

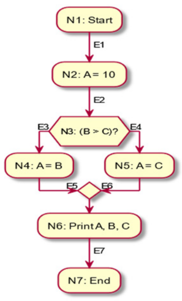

Figure 9: Code control-flow graph example.

In our second experiment, we focused on examining the structure and complexity of auto-generated code by employing the cyclomatic complexity metric. This metric is pivotal in quantifying code complexity, as it counts the number of linearly independent paths through the source code. The calculation is based on the code's control-flow graph, similar to the one illustrated in Figure. The cyclomatic complexity (M) can be calculated from the formula:

$$M = E - N + 2P \tag{1}$$

Where E represents the number of edges in the flow graph, N is the number of nodes, and P indicates the number of separate branches in the graph. For example, in the graph shown in Figure 9, the calculated M equals 3. This value of M is crucial for assessing various aspects of the software, including the difficulty of testing, maintenance, understanding, refactoring, performance, reliability, and documentation. Based on M's value, risks are categorized as follows: M between 1 and 10 indicates low risk, M between 11 and 20 suggests moderate risk, M between 21 and 50 points to high risk and may necessitate reviewing or subdividing the code into smaller modules, and M exceeding 50 signals severe risk, requiring significant refactoring.

As in our AMDD approach, we emphasized the effect of adding the formal constraints on generating a deployed code, our interest in this experiment is to understand the influence of the constraints layer on the auto-generated code. Therefore, in the experiment we auto-generated two distinct deployments, that differ in the level of constraints involved in their models. The first model implements only the OCL constraints, while the second model add the FIPA-ontology to the model.

Table 1: Cyclomatic Complexity of the auto-generated code.

Class	Operator		MCC		UVF-Manager		UV	
Constraints	OCL	OCL + Ontology	OCL	OCL + Ontology	OCL	OCL + Ontology	OCL	OCL + Ontology
Edges (E)	8	12	15	22	16	23	8	12
Nodes (N)	8	11	13	19	14	19	8	11
Branches (P)	1	1	1	1	1	1	1	1
Complexity (M)	2	3	4	5	4	6	2	3

After generating the two distinct deployments, we transformed the agent classes into control flow diagrams to calculate their M, as shown in Table. Comparing the M values of the auto-generated code is evidence that the complexity is slightly increasing by adding FIPA-ontology constraints. However, the complexity of all the classes in both deployments is still locating under the low-risk category. This means that the auto-generated structure is adequate and does not need any further refactoring. Furthermore, the highest M value equals to 6 belongs to the UVF-manger in the second deployment, where both OCL and FIPA-ontology constraints are considered. This means that there is still a large risk margin that allows to add further constraints in our model, without passing the low-risk threshold.

6 DISCUSSION, CONCLUSION, AND FUTURE WORK

Our research addresses the challenges of auto-generating deployable code from natural language using LLMs like ChatGPT, mainly due to language ambiguity. We used formal modelling languages such as UML to improve ChatGPT's interpretation, revealing a gap in agility within the MDD process. By integrating "constraints" into UML models, we added semantic depth for more accurate code generation, enhancing software structure and communication. In a case study, we applied our AMDD approach to model a multi-agent UVF system. We used class, activity, and state diagrams for agents' layout and behaviour, employing OCL for structure and FIPA-ontology for communication. This model was the basis for auto-generating Java and Python code using GPT-4, chosen for its improved reasoning capabilities. The success of our approach depended on GPT-4 understanding of the model's constraints.

In the first evaluation experiment, we examined the behaviour of auto-generated code within simulation environments: Java's JADE and Python's PADE frameworks. Both deployments effectively captured the intended agent interactions, though there were minor sequence variations between them. Remarkably, GPT-4 not only adhered to the specified agent logic but also enriched it by introducing two new behaviours in the MCC agent's communication sequence. This addition highlighted the power of communication constraints in guiding GPT-4 and its enhanced comprehension of agent interactions. While these improvements were impressive, they underscored a need for meticulous code review. Despite GPT-4's advancements, ensuring that the generated code remains consistent with design intentions is crucial to prevent unexpected behaviours.

In the second experiment, we examined the structure of the auto-generated code, specifically by assessing its cyclomatic complexity. In this experiment, we created two separate deployments. The first deployment code resulted from a model that involves only OCL constraints, while the second deployment code resulted from a model that involves both OCL and FIPA-Ontology constraints. Our analysis revealed the intriguing finding that integrating FIPA-ontology constraints didn't dramatically augment the complexity of the auto-generated code. This suggests that these constraints provide meaningful semantics without unduly complicating the resultant codebase. Furthermore, the analysis also hinted at a notable latitude in our approach. There appears to be a reasonable buffer allowing for the inclusion of additional constraints to the model in future iterations without triggering an immediate need for a code refactor. This is indicative of the robustness and scalability inherent in our AMDD approach.

Our findings indicate that formal modelling languages can mitigate natural language ambiguities in code generation. Meta-modelling constraints refine this process and provide structural complexity insights, signalling a transformative approach to agile MDD practices. Future research will focus on assessing code correctness, introducing privacy and cybersecurity constraints, and comparing our methodology with existing MDD frameworks to enhance industry adoption.

REFERENCES

Ahmad, A., Waseem, M., Liang, P., Fehmideh, M., Aktar, M. S., & Mikkonen, T. (2023). Towards Human-Bot Collaborative Software Architecting with ChatGPT (arXiv:2302.14600).

Brulin, S., & Olhofer, M. (2023). Bi-level Network Design for UAM Vertiport Allocation Using Activity- Based Transport Simulations.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating Large Language Models Trained on Code (arXiv:2107.03374).

David, I., Latifaj, M., Pietron, J., Zhang, W., Ciccozzi, F., Malavolta, I., Raschke, A., Steghöfer, J.-P., & Hebig, R. (2023). Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. Software and Systems Modeling, 22(1), 415–447. https://doi.org/10.1007/s10270-022-01010-3

Dong, Y., Jiang, X., Jin, Z., & Li, G. (2023). Self-collaboration Code Generation via ChatGPT

Kapferer, S., & Zimmermann, O. (2020). Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, 299–306.

Kelly, S., & Tolvanen, J.-P. (2008). Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons.

Sadik, A. R., Bolder, B., & Subasic, P. (2023). A self-adaptive system of systems architecture to enable its ad-hoc scalability: Unmanned Vehicle Fleet - Mission Control Center Case study. Proceedings of the 2023 7th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence, 111–118.

Sadik, A. R., Ceravola, A., Joublin, F., & Patra, J. (2023). Analysis of ChatGPT on Source Code (arXiv:2306.00597).

Sadik, A. R., & Goerick, C. (2021). Multi-Robot System Architecture Design in SysML and BPMN. Advances in Science, Technology and Engineering Systems Journal, 6(4). https://doi.org/10.25046/aj060421

Sadik, A. R., & Urban, B. (2018). CPROSA-Holarchy: An Enhanced PROSA Model to Enable Worker–Cobot Agile Manufacturing. International Journal of Mechanical Engineering and Robotics Research, 7(3).

Sarkisian, A., Vasylkiv, Y., & Gomez, R. (2022). System Architecture Supporting Crowdsourcing of Contents for Robot Storytelling Application.