Software Engineering Copilot

Domain Modelling

Author's: Andreas Rausch, Vasiliy Seibert,
Darshan Shah, Meet Chavda, Mitbhai Chauhan, Pratham Rathod

***Abstract*—This paper presents Domain Modelling Copilot, a web-based tool that uses AI to convert natural language scenarios into domain models and UML diagrams. Powered by OpenAI's language models, it automates requirement classification, scenario summarization, and model generation. The tool streamlines early software design, reduces manual effort, and supports agile development workflows. [1]**

I. INTRODUCTION

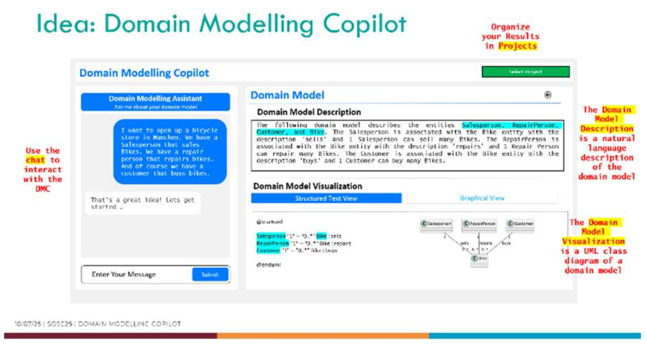
The Domain Modelling Copilot is an AI-powered tool designed to automate the transformation of natural language software requirements into formalized domain models and Unified Modelling Language (UML) diagrams. By leveraging large language models (LLMs), the system processes user-provided scenarios and generates structured representations of domain concepts, relationships, and behaviours. This approach enhances the efficiency and accuracy of early-stage software design by minimizing manual modelling tasks. The tool aligns with agile and model-driven development methodologies, facilitating better communication between technical and non-technical stakeholders and supporting a faster transition from requirements to design artifacts. [1,2]

II. MOTIVATION

The process of converting natural language requirements into formal domain models is traditionally labour-intensive and error-prone. Manual interpretation often leads to ambiguity, miscommunication, and iterative refinement cycles that delay development. These challenges are particularly critical in agile environments, where rapid feedback and adaptability are essential. Recent advancements in LLMs offer new opportunities to automate semantic interpretation tasks. Motivated by the need to reduce design bottlenecks and improve modelling consistency, we developed a system that bridges the gap between informal language and formal software models. This automation aims to reduce cognitive overhead and support more reliable and scalable system design practices.

III. DOMAIN MODELLING COPILOT

The interface of the Domain Modelling Copilot (DMC), shown in Fig. 6, embodies the key concepts of natural language interaction, automated domain modeling, and real-time visualization, aiming to bridge the gap between domain experts and software developers.


Idea: Domain Modelling Copilot

I want to open up a bicycle store in München. We have a Salesperson that sales Bikes. We have a repair person that repairs bikes. And of course we have a customer that buys bikes.

That's a great idea! Lets get started ..

The following domain model describes the entities Salesperson, RepairPerson, Customer, and Bike. The Salesperson is associated with the Bike entity with the description 'sells' and 1 Salesperson can sell many Bikes. The RepairPerson is associated with the Bike entity with the description 'repairs' and 1 Repair Person can repair many Bikes. The Customer is associated with the Bike entity with the description 'buys' and 1 Customer can buy many Bikes.

```
@startuml

Salesperson "1" -- "0..*" Bike : sells
RepairPerson "1" -- "0..*" Bike : repairs
Customer "1" -- "0..*" Bike : buys

@enduml
```

10/07/25 | SOSE25 | DOMAIN MODELLING COPILOT

Organize your Results in Projects
Domain Modelling Copilot
Select Project
Domain Modelling Assistant
Ask me about your domain model
Domain Model
Domain Model Description
Domain Model Visualization
Structured Text View
Graphical View
Enter Your Message
Submit
Use the chat to interact with the DMC
The Domain Model Description is a natural language description of the domain model
The Domain Model Visualization is a UML class diagram of a domain model
Salesperson
RepairPerson
Customer
Bike
sells
repairs
buys

Fig. 1 Domain Modelling Copilot

1) Chat-based Interaction Panel

Located on the left side of the interface, this component features a conversational input field where users can describe their domain in natural language. For example, a user might state: "I want to open up a bicycle store in München. We have a Salesperson that sells Bikes. We have a repair person that repairs bikes. And of course, we have a customer that buys bikes."

This text is parsed by the system to identify domain-relevant entities and relationships. The chat interface lowers the entry barrier for non-technical stakeholders, allowing them to contribute meaningfully to domain modeling without requiring knowledge of formal modeling languages like UML.[2]

2) Project Organization and Navigation

The top-right section includes a **"Select Project"** button, which allows users to organize their modeling efforts within discrete projects. This structure supports better version control, collaboration, and task segmentation—critical features in iterative and agile development workflows.

3) Domain Model Description (Textual Output)

The center-right section of the interface displays the **Domain Model Description**, a structured natural language explanation generated from the user's chat input. In this example, the system identifies four key entities:

Salesperson, *RepairPerson*, *Customer*, and *Bike*. It articulates their relationships using clear verbs such as "sells", "repairs", and "buys", and includes cardinality (e.g., “1 Salesperson can sell many Bikes”).

This component helps validate whether the system correctly understood the user’s intent. It also aids in maintaining a human-readable specification of the model that can be reviewed by stakeholders without UML expertise.

4) Domain Model Visualization (Graphical Output)

Beneath the textual description lies the **Domain Model Visualization**, an automatically generated UML class diagram. This diagram mirrors the entities and relationships described above, depicting them graphically with proper UML syntax. Each entity is represented as a class node, and associations (e.g., “sells”, “repairs”, “buys”) are annotated with multiplicities (“1”, “0..*”) for clarity.

The dual representation (textual + visual) ensures a **bidirectional cognitive channel**: users can understand the model visually and validate it textually, or vice versa. It also supports iterative feedback and refinement, as users can revise their original input to modify the resulting model.

5) Structured and Graphical View Tabs

The interface provides two toggle options: “Structured Text View” and “Graphical View”. These enable users to switch between different representations of the domain model, depending on their preference or task. The **Structured Text View** aligns closely with source-code-like representations (e.g., PlantUML) [3], whereas the **Graphical View** shows the standard UML class diagram.

A. DOMAIN MODEL ARCHITECTURE

The Domain Model architecture of the Domain Modelling Copilot (DMC) system is represented using a Unified Modeling Language (UML) class diagram, as shown in Fig. 2 [2].This diagram illustrates the key entities and relationships that define the functional structure of the system and its interaction logic.

The domain model of the **Domain Modelling Copilot (DMC)** system is structured using a Unified Modeling Language (UML) class diagram, which serves as a formal representation of the system's core components and their interrelationships.

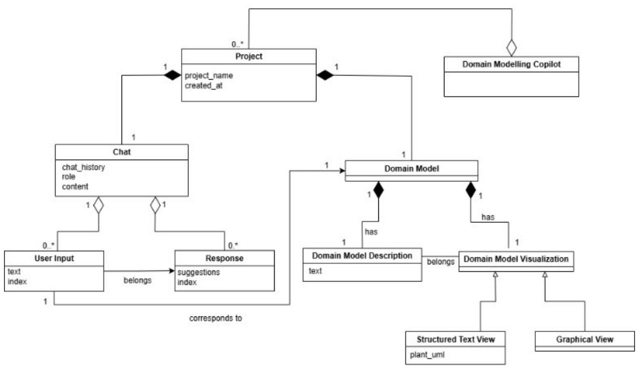


Fig. 2 Domain Model

At the center of the architecture is the **Project** class, which acts as a container for modeling activities initiated by users. Each project encapsulates a modeling session that includes the domain description and its corresponding visualization. The **User** class represents individuals who create and manage these projects, supplying input via a natural language interface and iteratively refining the model. Users interact with the system through a conversational interface, enabling intuitive domain specification without requiring expertise in modeling languages.

Within each project, a **DomainModelDescription** is generated based on the user’s input. This entity captures a structured natural language interpretation of the problem domain, identifying key **Entities** (e.g., Salesperson, Bike, RepairPerson, Customer) and their respective **Relationships** (e.g., sells, repairs, buys). These elements are automatically extracted through natural language processing techniques. The **DomainModelVisualization** class provides a synchronized graphical UML class diagram, ensuring that changes to the text description are reflected visually and vice versa. The model supports bidirectional editing and includes cardinality specifications to enforce UML compliance. This integrated design ensures consistency, traceability, and usability for both technical and non-technical stakeholders, thereby supporting a collaborative and agile domain modeling process.

B. ACTIVITY DIAGRAM

The activity diagram of the proposed Domain Modeling Copilot (DMC) system, which enables interactive and iterative construction of domain models through a natural language interface. The system architecture comprises three primary components: the User, the Chatbot, and the LLM Wrapper, each responsible for distinct roles in the modeling workflow.

The process is initiated when the User provides input in natural language, which may contain a general inquiry or a description of a domain model. This input is received and displayed by the Chatbot component, which subsequently delegates the classification task to the LLM Wrapper. The LLM Wrapper leverages a prompt-based method to determine the nature of the user input via the `determine_user_input_type` function.

If the input corresponds to a general query, the LLM generates an appropriate response using a general-purpose prompt. Conversely, if the input is classified as a domain model description, the system further evaluates whether the description is sufficiently detailed. In cases where additional detail is required, the user is prompted to elaborate. Otherwise, a structured domain model description is synthesized using a specialized prompt.

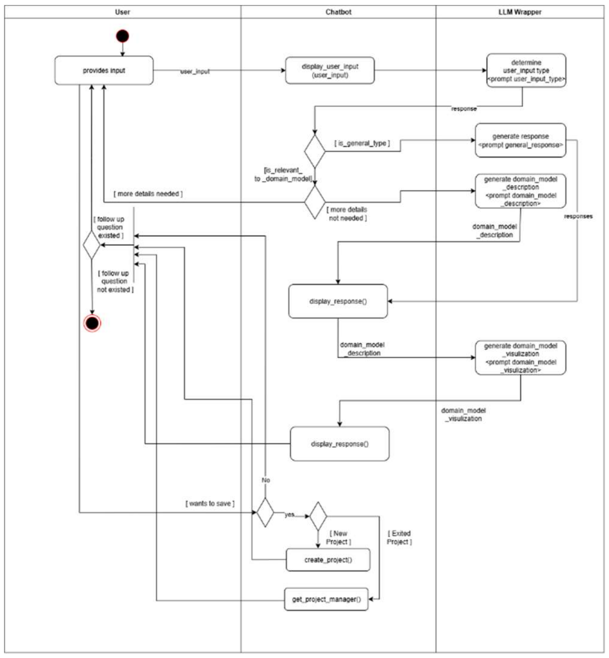

User
Chatbot
LLM Wrapper
provides input
user_input
display_user_input (user_input)
determine user_input type <prompt user_input_type>
response
[is_general_type]
generate response <prompt general_response>
[is_relevant_ to _domain_model]
generate domain_model _description <prompt domain_model _description>
[more details needed]
[more details not needed]
responses
domain_model _description
[follow up question existed]
[follow up question not existed]
display_response()
domain_model _description
generate domain_model _visulization <prompt domain_model _visulization>
domain_model _visulization
display_response()
No
[wants to save]
yes
[New Project]
[Exited Project]
create_project()
get_project_manager()

Fig 3. Activity Diagram

The generated domain model description is passed to a visualization module, which uses a PlantUML-based template to create a graphical representation of the domain model. The resulting visualization is displayed to the user, completing the initial interaction cycle.

The system also accommodates follow-up questions, enabling users to iteratively refine the domain model through subsequent interactions. This loop continues until the user expresses satisfaction with the result.

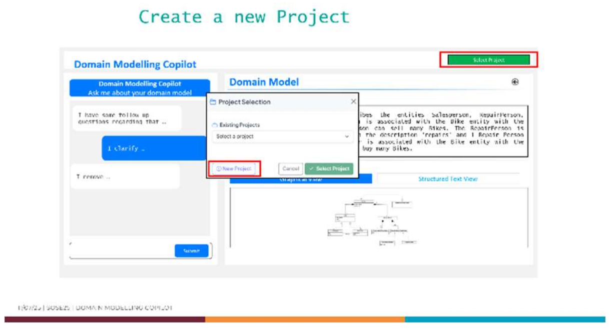
Finally, if the user opts to save the project, the system determines whether to initiate a new project instance or update an existing one. The corresponding functions `create_project()` or `get_project_manager()` are invoked accordingly to persist the model under the appropriate project context.

This workflow demonstrates a tightly integrated system that combines large language models with structured modeling logic to support natural language-based domain engineering. The iterative design ensures flexibility, while the visualization and project persistence capabilities provide practical utility for software modeling tasks.

IV. USER JOURNEY

To demonstrate the practical workflow enabled by the Domain Modelling Copilot (DMC), the system designers developed a persona-driven user journey centered on "Alex," a non-technical entrepreneur with a strong business concept but limited experience in formal software modeling. This scenario reflects a common challenge in software engineering: bridging the communication gap between business stakeholders and technical teams. The user journey illustrates how DMC guides users through model creation, feedback, and refinement using natural language interactions and automatic UML generation.

The journey begins with **project initialization**, where Alex opens the DMC interface and selects the option to **create a new project**. At this stage, Alex is prompted to provide a project name, which is validated for uniqueness. This step marks the beginning of a new modeling session, encapsulating all subsequent user interactions (see *Fig. 3: Creating a New Project*).


Create a new Project

I have some follow up questions regarding that ...

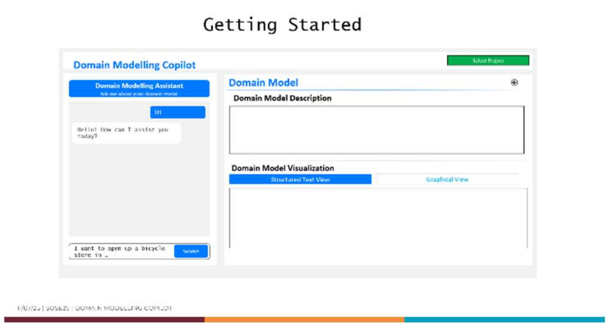
I clarify ...

I remove ...

...ibes the entities Salesperson, RepairPerson, ... is associated with the Bike entity with the ...son can sell many Bikes. The RepairPerson is ... the description 'repairs' and 1 Repair Person ...r is associated with the Bike entity with the ... buy many Bikes.

Fig. 4 Creating a New Project

Once initialized, Alex engages in a **natural language conversation** with the system by describing the core domain concept. For example, Alex might state: *"We have a Salesperson that sells Bikes, a RepairPerson who fixes them, and Customers who buy them."* This input is interpreted by the system and translated into a **Domain Model Description**—a structured textual summary of identified entities and their relationships—automatically displayed in the interface (*refer to Fig. 2*).



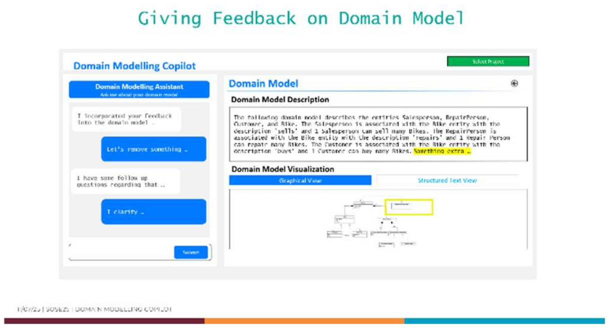
Hi!

Hello! How can I assist you today?

I want to open up a bicycle store in …

Fig. 5 Chat Section

Following this, the system concurrently generates a **Domain Model Visualization**, a UML class diagram that reflects the textual model in graphical form. This dual representation allows Alex to visually validate whether the system has correctly interpreted the domain. If the model appears inaccurate or incomplete, Alex can initiate a **feedback and revision cycle** by modifying the description in natural language or interacting directly with the visual elements. These changes propagate bidirectionally, ensuring consistency between the textual and graphical views (see *Fig. 5: Giving Feedback on the Domain Model*).


Giving Feedback on Domain Model
Domain Modelling Copilot
Select Project
Domain Modelling Assistant
Ask me about your domain model
Domain Model
Domain Model Description
Domain Model Visualization
Graphical View
Structured Text View
Submit
1/07/25 | SOSE25 | DOMAIN MODELLING COPILOT

I incorporated your feedback into the domain model …

Let's remove something …

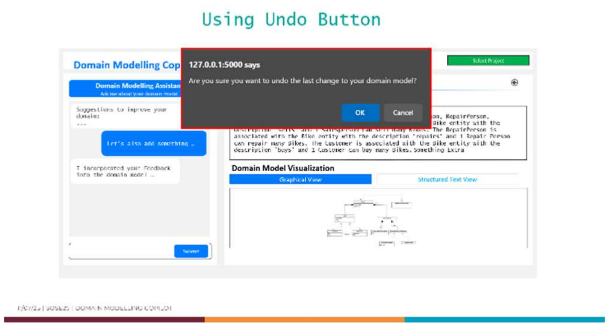
I have some follow up questions regarding that …

I clarify …

The following domain model describes the entities Salesperson, RepairPerson, Customer, and Bike. The Salesperson is associated with the Bike entity with the description 'sells' and 1 Salesperson can sell many Bikes. The RepairPerson is associated with the Bike entity with the description 'repairs' and 1 Repair Person can repair many Bikes. The Customer is associated with the Bike entity with the description 'buys' and 1 Customer can buy many Bikes. Something extra …

Fig. 6 Giving Feedback on the Domain Model

An essential usability feature in the user journey is the **Undo functionality**, which allows Alex to revert changes at any stage of the modeling process. This encourages iterative exploration without risk, aligning with agile and user-centered development practices. The entire journey—from project creation to visualization and refinement—is designed to be intuitive and accessible, enabling even non-technical users to engage in sophisticated modeling tasks effectively (see *Fig. 6: Using Undo Button*).


Using Undo Button
Domain Modelling Cop
Select Project
Domain Modelling Assistan
Ask me about your domain model
OK
Cancel
Domain Model Visualization
Graphical View
Structured Text View
Submit
1/07/25 | SOSE25 | DOMAIN MODELLING COPILOT

127.0.0.1:5000 says

Are you sure you want to undo the last change to your domain model?

Suggestions to improve your domain:
...

Let's also add something …

I incorporated your feedback into the domain model …

…on, RepairPerson, … Bike entity with the … The RepairPerson is associated with the Bike entity with the description 'repairs' and 1 Repair Person can repair many Bikes. The Customer is associated with the Bike entity with the description 'buys' and 1 Customer can buy many Bikes. Something Extra

Fig. 7 Using Undo Button

In summary, the Alex scenario demonstrates how DMC transforms the traditionally complex process of domain modeling into a guided, conversational workflow. It reduces cognitive overhead, lowers technical barriers, and fosters collaborative design by making software modeling more inclusive and responsive to user input.

V. VISION OF AGENT-BASED APPROACH

To enhance the modularity, responsiveness, and scalability of the Domain Modelling Copilot (DMC), a future architectural direction involves transitioning to a **multi-agent system (MAS)[5]**. The envisioned design replaces a monolithic control flow with a set of loosely coupled, autonomous agents—each responsible for a specialized task within the modeling lifecycle.[6] This agent-based approach aims to support parallelism, maintainability, and dynamic behavior while preserving a seamless user experience.

The proposed architecture introduces **three primary agents**, each designed to operate autonomously yet cooperatively within the system:

1. **ChatAgent:**
   The Chat Agent serves as the primary interface between the user and the system. It is responsible for receiving, interpreting, and managing natural language inputs. This agent maintains conversational context, detects modeling intents, and initiates downstream agent workflows. By providing a user-friendly interaction layer, it enables domain experts—regardless of technical background—to describe their systems conversationally.
2. **Domain Model Description Agent:**
   This agent takes the parsed natural language input and generates a structured, formal representation of the described domain. It identifies domain entities, roles, and relationships, constructing a textual Domain Model Description in a consistent and

verifiable format.[5] The agent is designed to handle model validation and semantic enrichment, ensuring that the domain logic adheres to modeling constraints.[6]

3. **Domain Model Visualization Agent:** The Visualization Agent transforms the Domain Model Description into a graphical UML class diagram. It is responsible for layout optimization, graphical rendering, and visual updates. The agent ensures bidirectional consistency—any change made in the textual model is reflected in the diagram and vice versa. Additionally, this agent handles diagram customization and user interactions with visual elements (e.g., adding, editing, or removing entities or relationships).[4]

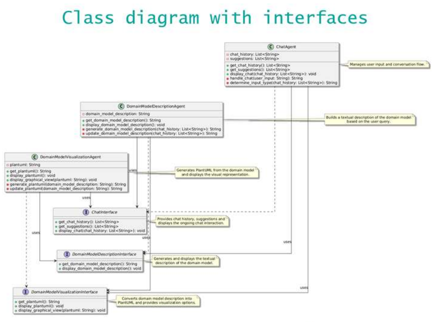

Class diagram with interfaces
ChatAgent
chat_history: List<String>
suggestions: List<String>
get_chat_history(): List<String>
get_suggestions(): List<String>
display_chat(chat_history: List<String>): void
handle_chat(user_input: String): String
determine_input_type(chat_history: List<String>): String
Manages user input and conversation flow.
DomainModelDescriptionAgent
domain_model_description: String
get_domain_model_description(): String
display_domain_model_description(): void
generate_domain_model_description(chat_history: List<String>): String
update_domain_model_description(chat_history: List<String>): String
Builds a textual description of the domain model based on the user query.
DomainModelVisualizationAgent
plantuml: String
get_plantuml(): String
display_plantuml(): void
display_graphical_view(plantuml: String): void
generate_plantuml(domain_model_description: String): String
update_plantuml(domain_model_description: String): String
Generates PlantUML from the domain model and displays the visual representation.
uses
ChatInterface
get_chat_history(): List<String>
get_suggestions(): List<String>
display_chat(chat_history: List<String>): void
Provides chat history, suggestions and displays the ongoing chat interaction.
DomainModelDescriptionInterface
get_domain_model_description(): String
display_domain_model_description(): void
Generates and displays the textual description of the domain model.
DomainModelVisualizationInterface
get_plantuml(): String
display_plantuml(): void
display_graphical_view(plantuml: String): void
Converts domain model description into PlantUML and provides visualization options.

Fig. 8 Class Diagram [AI AGENTS]

The agents operate in a **coordinated workflow**, where the Chat Agent triggers the modeling pipeline, the Description Agent processes and refines the domain logic, and the Visualization Agent renders the updated diagram. This architecture fosters a **modular and reactive system**, where each component can evolve independently while contributing to a unified modeling experience.

Compared to a traditional centralized architecture, the agent-based approach provides several advantages. It enables **parallel processing**, improving performance and responsiveness. It also enhances **fault isolation**—if one agent fails or produces unexpected output, others can continue operating or recover gracefully. Furthermore, this structure supports future extensions, such as integrating new agents (e.g., validation or recommendation agents) without restructuring the core system.

In summary, the agent-based vision for DMC transforms it into a **collaborative multi-agent environment**, where each intelligent component contributes to an interactive, adaptive, and scalable modeling workflow. By decoupling responsibilities across specialized agents, the system achieves better maintainability and lays the foundation for advanced features like continuous feedback, semantic validation, and user personalization.

VI. CONCLUSION

This paper introduced the Domain Modelling Copilot (DMC), a web-based tool designed to automate the transformation of natural language requirements into structured domain models and UML diagrams. By combining AI-driven language understanding with real-time visualization, DMC reduces manual modeling effort and enhances communication between technical and non-technical stakeholders. The tool supports agile and model-driven development by offering an intuitive interface and dual representations of domain logic.

To address challenges in scalability, maintainability, and responsiveness, the paper proposed a modular multi-agent system architecture comprising a Chat Agent, a Domain Model Description Agent, and a Visualization Agent. This approach enables parallel processing, better fault isolation, and future extensibility. The agent-based design lays the groundwork for evolving DMC into a more intelligent, adaptive, and collaborative modeling environment. Future work will explore the integration of validation, recommendation, and personalization agents to further enrich the user experience and modeling capabilities.

VII. REFERENCES

[1] V. Vernon, Domain-Driven Design Reference: Definitions and Pattern Summaries. https://domainlanguage.com/ddd/reference/

[2] Object Management Group (OMG), OMG Unified Modeling Language (UML), Version 2.5, 2015.https://www.omg.org/spec/UML/2.5/

[3] PlantUML, PlantUML Language Reference Guide, 2024. https://plantuml.com/]

[4] A. Rausch, M. Trapp, and M. Naab, "Towards a Componentware Approach for Agent-Based Systems," in Proc. Int. Conf. Software Engineering Research and Practice (SERP'06), Las Vegas, USA, 2006.

[5] N. R. Jennings, "On agent-based software engineering," Artificial Intelligence, vol. 117, no. 2, pp. 277–296, Mar. 2000, doi: 10.1016/S0004-3702(99)00107-1.

[6] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd ed. Chichester, UK: Wiley, 2009.