Journal Pre-proof

Debugging in the Domain-Specific Modeling Languages for multi-agent systems

Baris Tekin Tezel, Geylani Kardas

PII: S2590-1184(25)00011-5
DOI: https://doi.org/10.1016/j.cola.2025.101325
Reference: COLA 101325

To appear in: *Journal of Computer Languages*

Received date : 17 May 2024
Revised date : 6 February 2025
Accepted date : 6 February 2025

Please cite this article as: B.T. Tezel and G. Kardas, Debugging in the Domain-Specific Modeling Languages for multi-agent systems, *Journal of Computer Languages* (2025), doi: https://doi.org/10.1016/j.cola.2025.101325.

Debugging in the Domain-Specific Modeling Languages for Multi-Agent Systems

Baris Tekin Tezel[a], Geylani Kardas[b]

[a]*Department of Computer Science, Dokuz Eylul University, Buca, Izmir, 35390, Türkiye, baris.tezel@deu.edu.tr*
[b]*International Computer Institute, Ege University, Bornova, Izmir, 35100, Türkiye, geylani.kardas@ege.edu.tr*

Abstract

In many cases, developers face challenges while implementing Multi-Agent Systems (MAS) due to the complexity of expanding software systems, despite the presence of numerous agent programming environments and platforms. To tackle this complexity, Model-driven Engineering (MDE) can be employed at a higher level of abstraction and component modeling before diving into MAS development, which helps alleviate the intricacies. Probably, the most effective method of incorporating MDE into Multi-Agent Systems (MAS) is to adapt Domain-Specific Modeling Languages (DSMLs) along with integrated development environments (IDEs). These tools make it easier to model the system and generate the necessary code for the development process. Although existing MAS DSML IDEs offer some control over systems modeled based on the language's syntax and semantics, they lack built-in debugging support. This deficiency leads to uncertainty among agent developers about the accuracy of models prepared during the design phase. To address this issue, this study proposes a comprehensive debugging framework (MASDebugFW) that facilitates the design of agent components within modeling environments. The framework's utilization commences with modeling MASs using a design language, and then converting these design model instances into a runtime model. Following that, the runtime model undergoes simulation using an integrated simulator specifically designed for debugging purposes. Additionally, the framework includes a simulation environment model and a control mechanism to manage the simulation process effectively. These features further enhance the debugging capabilities

and overall functionality of MASDebugFW. Furthermore, we have qualitatively and quantitatively evaluated MASDebugFW, subjecting all obtained results to statistical analysis. The evaluation results show that, on average, the implemented framework reduces debugging time by around 45%, leading to more efficient debugging processes. Moreover, it significantly enhances bug detection and repair capabilities, as it increases the number of bugs fixed in the models by approximately 50%.

Keywords: Empirical software engineering, Software Debugging, Domain-specific Modeling Languages, Multi-agent systems, simulation

1. Introduction

As the demands of modern society continue to grow, the reliance on software systems is increasing day by day. This increasing need and dependence leads to the complexity of software systems. While mechanical systems and software systems had limited interaction in the past, current requirements necessitate tight integration. Multi-agent systems (MAS) are one of the technologies used in the realization of software systems belonging to environments with such interactions [1].

MASs consist of computer systems called agents, each of which can interact with others. Agents can perform autonomous actions within an environment to achieve their design goals [2]. Agents can exhibit complex behavior patterns, ranging from simple reflexive responses to environmental changes to more proactive decision-making. Although many different agent programming environments and platforms such as JACK [3], JADE [4], JADEX [5], and Jason [6] have been developed until today, software developers still face challenges in dealing with the complexity of implementing MASs caused by the complexity of growing software systems [7, 8].

With the growing complexity of MASs, Agent-oriented Software Engineering (AOSE) [9] researchers are working to enable development processes, methods, and techniques that allow system developers to successfully address system concerns such as security, interoperability, and performance. Among these techniques, it is seen that software modeling and Model-driven Engineering (MDE) [10, 11, 12, 13, 14] are widely used [15, 16, 8, 17, 18].

By working at a higher level of abstraction and utilizing component modeling with MDE before going deeper into the MAS development can help reduce the complexity that may arise in the implementation of the MAS [19, 20, 21]. In the development processes based on MDE, models are considered to be first-class concepts. When developing systems according to the MDE process, engineers create models that reflect discrete parts of the system in various modeling languages and provide high-level abstraction [22]. In this way, development processes provide abstractions that allow engineers to focus on what the system should do instead of how the system should be implemented. In other words, the MDE allows engineers to focus on defining the system's functionality rather than getting caught up in the implementation details.

AOSE researchers define various agent metamodels [23, 24, 25] to facilitate the modeling of every relevant aspects of MAS at the most suitable level of abstraction. These metamodels are designed to encompass agent characteristics like plans and beliefs, goals, and interactions within MAS organizations. To implement MDE effectively for MASs, a convenient approach involves tailoring Domain-Specific Modeling Languages (DSMLs) using integrated development environments (IDEs) that support both modeling and code generation for the targeted system [16]. The proposed MAS DSMLs (e.g., [26, 27, 28, 19, 29, 30]) are based on the mentioned agent metamodels and offer various abstract syntaxes. These DSMLs facilitate the modeling of both static and dynamic aspects of agent software from different perspectives in the MAS domain, including agent internal behavior, interactions with other agents, and the utilization of environmental entities. This allows developers in the MAS field to model systems at the appropriate level of abstraction using the concepts they are most familiar with. However, although DSMLs provide intuitive modeling capabilities and help manage the development process for highly complex systems, they do not guarantee error-free models [31, 32, 33].

The complexity of MAS modeling cannot be entirely eliminated by using a different development method; however, DSMLs provide an intuitive and domain-specific approach to system design due to their proximity to field-specific concepts. Early verification techniques that enable the identification of unmet system require-

ments can discover and observe the failures of a system. However, when a failure is observed, it is necessary to determine why the failure occurred and how to modify the model to eliminate the cause of the failure. With a good understanding of the models, the faults that are the reasons of the failures can be found and rectified manually. Alternatively, a wide variety of debugging techniques (e.g. [34, 35, 36, 37]) can be used to help developers to find and fix the faults [38]. Indeed, the IDEs associated with these MAS DSMLs do offer checks for systems modeled based on the syntax and semantic descriptions of the respective DSML. However, a notable limitation is the absence of built-in support for debugging MAS models [39]. As a result, developers are left uncertain about the accuracy and correctness of the MAS models prepared during the design stage.

Understanding the runtime behavior of software systems can be a challenging activity. Debuggers play a crucial role in helping developers comprehend the runtime behavior by providing direct access to running systems [40, 41, 42]. In the context of software development, debugging support is typically provided with a language and IDE that allows monitoring and modifying the status of a running program. However, in Domain-Specific Modeling (DSM), debugging activities have a broader meaning [33]. Model developers must debug the models at the model level, not at the code level. This new requirement prompts the researchers on developing new debugging approaches for model-driven development and DSMLs.

Finding the cause of a system's failure and correcting it can be done manually with a good understanding of the models. Alternatively, a wide variety of debugging techniques can be used to help developers find the cause of the failure [40]. Since software bugs have a large economic impact [43], it is imperative to provide such debugging tools and techniques to assist developers as much as possible. With the boosted importance of MDE techniques for developing complex systems, researchers are increasing the reliability of modeled systems by integrating, inter alia, verification and validation techniques. However, few debugging tools and techniques are available, especially when it comes to models developed using DSML (e.g. [34, 35, 37, 44]). To locate the source of a failure observed in a system implemented using models, developers often have to resort to temporary methods. An

approach that involves analyzing or debugging code generated from models can be proposed by reusing well-known debugging techniques built into a DSML IDE. However, this approach seems extremely problematic since the developer needs to change the contexts and understand the semantics of the underlying application language. Moreover, it must perform a conceptual mapping between the code and the model properly and map the faults on the code to the model entities with this conceptual mapping. Instead, it is clear that specific debug support is required for modeling languages [45, 46]. However, although structured approaches to creating model debugging environments based on language engineering techniques are extremely limited, debugging support in the modeling of MASs is not yet available in any study. On the other hand, based on reports and observations[17, 47, 48, 49], it seems that a significant number of developers of agents rely on basic debugging techniques, such as using straightforward logging statements, in order to understand the situation at hand. The current agent platforms mainly offer assistance in examining the cognitive state of an agent [48]. Hence, we investigated some possible debugging approaches to support MAS DSMLs [39] and also presented the conceptual definition of the MAS metamodels to provide debugging capabilities to MAS DSMLs [50]. Lastly, in this paper, we introduce a complete general debugging framework that supports the design of agent components within modeling environments [1]. Furthermore, MASDebugFW has been evaluated both qualitatively and quantitatively. In this context, the contributions of this study can be listed as follows:

- It contributes to the AOSE community and the state of the art in software engineering by discussing how a MAS debugging framework can be constructed by enhancing the present design structures of existing MAS DSMLs with supplementary runtime, simulation, and visualization languages.
- It introduces a framework for creating a completely new MAS DSML with debugging support and its implementation method. The proposed framework is

[1] To provide additional clarity and demonstrate the usability of MASDebugFW, a video walkthrough is available and can be accessed at https://youtu.be/3MKDgob9ij4.

highly versatile, as it caters to the design of both agent internals and communications. Its flexibility opens avenues for developing MAS DSMLs equipped with built-in debuggers. Moreover, a new debug-supported MAS DSML developed by MASDebugFW is also presented in this study to demonstrate the applicability of this framework.

- It presents an evaluation method that has been developed and implemented to verify the effectiveness ofMASDebugFW. With the quantitative assessment performed, the effectiveness of the new debugging framework in time consumption, the true localization of faults, accurate diagnosis of faults, and acceptable corrections at the modeling level were confirmed. In addition, an expanded qualitative assessment model from the Technology Acceptance Model (TAM) [51] was developed to demonstrate the acceptance of this framework, and the framework was evaluated according to this model.

The rest of the paper is organized as follows. Section 2 provides a theoretical foundation, including background information on debugging in MAS and an overview of MASDebugFW. Section 3 provides a comprehensive explanation of MASDebugFW, detailing its design and implementation for enriching DSMLs with advanced debugging capabilities tailored for MAS models. Section 4 presents the case study, demonstrating the practical applicability of MASDebugFW by showcasing its effectiveness in identifying and resolving faults within a real-world MAS scenario. Practical researchers and developers may prefer to start with Section 4. For readers primarily interested in the technical evaluation of the framework, Section 5 offers quantitative and qualitative assessments of its performance. Section 6 discusses the threats to the validity of this work. Lastly, Section 7 finalizes the paper with discussion and conclusions. Depending on their focus, readers can choose to dive deeper into theoretical discussions (Sections 2 and 3) or practical applications and evaluations (Sections 4 and 5).

2. Related Work

Software debugging is a crucial yet challenging and time-consuming activity, particularly as software systems grow in size and complexity. This section summarizes the state-of-the-art debugging approaches for General Purpose Programming Languages (GPLs) and DSMLs, highlighting advancements in automation, visualization, and abstraction. Finally, studies on controlled experiments for DSLs and DSMLs are briefly discussed.

2.1. Debugging Approaches in General Purpose Programming Languages

Zeller [40] provided a foundational overview of state-of-the-art debugging techniques, introducing "scientific debugging" as an iterative process involving hypothesis formation, prediction, and experimentation to locate and resolve software bugs. Central to this approach is reproducing the failure conditions and iteratively refining hypotheses until the bug is identified and eliminated. Traditional interactive debuggers allow developers to navigate through code, inspect variables, and observe call stacks. While suitable for procedural languages, these debuggers may fall short for paradigms such as object-oriented programming, where objects—not functions—are the focal elements of development.

Object-centric debugging extends traditional approaches by enabling direct interactions with objects [52]. Similarly, event-based debugging allows users to define and monitor high-level events. For example, Marceau et al. [53] introduced a codable debugging approach employing data streams to track runtime behavior and generate high-level events.

Visualization enhances understanding by intuitively representing runtime states. Tools like the Data Display Debugger (DDD) [54] and systems developed by Cross et al. [55] enable visualizations of complex data structures like linked lists and hash tables. Risberg et al. [56] further introduced property probes, a mechanism that links source code spans with analysis nodes to assist developers in exploring program analysis results, even after code modifications.

Automation also plays a key role in modern debugging. Fault localization techniques, such as slicing, help identify code sections affecting specific variables [57],

while algorithmic debugging seeks to eliminate failing code paths through systematic isolation [58]. Silva's comparative study of algorithmic debugging techniques provided a framework for integrating their strengths. Iterative Delta Debugging [59] leverages historical program versions to isolate minimal code changes responsible for failures. Despite these advances, many automated techniques remain underutilized in industry, where traditional debugging approaches dominate.

For MAS, debugging approaches extend traditional paradigms by focusing on agent-level states, behaviors, and interactions. Tools developed for MAS, such as VisDebug [60] and agent communication monitors [61], provide tailored solutions that integrate visualization and partial automation to address the unique challenges posed by decentralized architectures [62].

2.2. Debugging Approaches in Domain-Specific Modeling Languages

Debugging is a critical yet challenging aspect of software development, especially for DSLs and DSMLs, as traditional debugging tools often lack support for domain-specific concepts. Initial studies, such as Kelly & Tolvanen [63] and Safa [64], observed that debugging in DSMLs frequently occurs at the code level without tool support. Mannadiar and Vangheluwe [33] proposed mapping programming language debugging concepts to DSMLs, emphasizing model-level debugging to improve efficiency and accuracy.

Subsequent approaches, such as Sequencer by Kosar et al. [34], introduced DSML-specific debugging tools. Similarly, Moldable Debugger by Chiş et al. [35] combined domain-specific debugging views and operations. Wu et al. [31] developed a sophisticated debugger enabling automated mapping between DSL models and synthesized code, facilitating breakpoints and execution control without requiring prior knowledge of the underlying code.

Innovative frameworks, like TIDE [65], SEL-based instrumentation [66], and omniscient debugging for xDSMLs [67], extended these capabilities, addressing issues such as non-executable model parts and providing platform-independent solutions. Recent advancements, such as the test coverage framework by Khorram et al. [68]

and temporal breakpoints by Paquier et al. [69], further enhanced debugging efficiency in complex systems.

Despite these advancements, no approaches specifically address MAS DSMLs, which require tailored debugging due to their unique properties, such as autonomy and distributed operations. The framework proposed in this study aims to fill this gap by providing structured debugging support for MAS DSMLs, integrating conventional debugging concepts with simulation environments to handle their inherent complexity.

2.3. Controlled Experiments on DSLs and DSMLs

Controlled experiments have been instrumental in evaluating the efficiency and effectiveness of DSLs compared to GPLs. Foundational studies, such as Prechelt et al. [70], highlighted the benefits of DSLs in enhancing programmer productivity and code accuracy. Building on this, Kosar et al. [71] demonstrated that DSLs improve comprehension accuracy and efficiency, aligning closely with domain-specific abstractions.

Further research by Kosar et al. [72] confirmed that DSLs reduce errors and cognitive load, making development more intuitive. Replication studies, including those by Erdweg et al. [73], validated these findings across varied conditions, reinforcing the generalizability of DSL benefits.

Recent work [74] examined the impact of code bloat on comprehension in genetic programming solutions, confirming its negative effect on understanding attribute grammars. These findings emphasize the significance of DSLs in enhancing efficiency and comprehension across domains.

Overall, the body of researches on controlled experiments with DSLs and DSMLs consistently demonstrates their benefits in terms of comprehension accuracy, efficiency, and reduced cognitive load. This extensive researches provide solid foundation for the ongoing development and adoption of DSLs and DSMLs in various specialized domains.

3. A debugging framework for DSMLs for MAS

In the realm of software development, debugging support is primarily facilitated through the synergy between a programming language and an IDE. The combination of these two components empowers developers to monitor and modify the execution of programs in real-time [40]. As stated in [45], various debugging techniques (e.g. breakpoints, stepping operators, symbolic execution) are used for GPLs.

However, it is essential for model developers to perform debugging activities at the model level rather than at the code level and this new requirement has led researchers to develop new debugging approaches for model-based development and DSMLs. Debugging methods and tools for DSMLs are generally less abundant and mature compared to those available for GPLs. For example, the Moldable Debugger [35] also enables domain-specific debuggers to be created by building and combining debugging with domain-specific debug views. An omniscient debugging method that enables unrestricted access to states during system execution has been applied in the development of debuggers for xDSMLs [44]. Additionally, this approach has been employed to enhance model transformations [75]. Inspired by these endeavors, we explored various debugging methods suitable for MAS DSMLs [39]. Motivated by the lack of debugging support or a method for existing MAS DSML studies, we presented two different debugging approaches in the model-driven development of software agents [39]. The first approach focuses on creating a mapping between the entities within the MAS model and the generated code. On the other hand, the second approach encompasses the meta-model-based explanation of the operational meanings of the agents. Although none of these two approaches was directly used in this study, our work described in this paper was shaped in light of the results obtained during the investigation of the applicability of these prior approaches.

In this paper, we present a framework (MASDebugFW) that extends existing MAS DSMLs with debugging capabilities and adds model debugging features during the design and implementation of new MAS DSMLs, based on the conceptual definition of MAS metamodels from our previous study [50]. Additionally, the framework

serves as a valuable reference for DSML developers, guiding them on how to integrate model debugging capabilities into the design and implementation of new MAS DSMLs. By following this framework, developers can enhance the debugging experience and overall reliability of their MAS DSMLs, resulting in more robust and effective modeling environments. It is designed to be generic, reusable, and adaptable to various aspects and perspectives of MAS DSMLs.MASDebugFW allows for modeling and validation of different aspects of MAS, such as the execution of agent plans, belief set consistency, or agent communications following well-defined agent protocols like Contract Net [76]. Figure 1 provides an overview of the framework, which comprises four distinct metamodels and a simulator that facilitates MAS operational semantics. The formalization of this framework is influenced from Van Mierlo's work [38], which proposes a structured and model-based approach to transform modeling and simulation environments into interactive debugging environments. Additionally, the framework incorporates sub-language descriptions from the ProMoBox [77] system and dynamic modeling language composition defined in [78]. Figure 1 represents the overview of MASDebugFW. This overview can also be seen as a metamodel of MASDebugFW. The metamodels and simulator work in unison to enable robust debugging functionalities for MAS DSMLs.

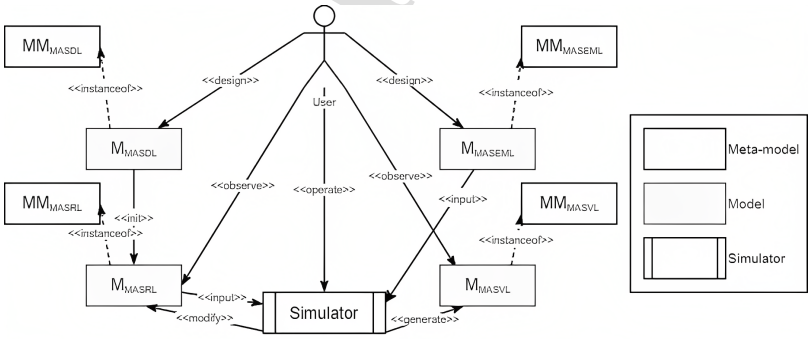


Figure 1: Overview of the generic debugging framework for MAS DSMLs.

In this study, MASDebugFW is designed to transform conventional modeling and simulation environments into interactive debugging environments. In this con-

text, the following aspects are presented and explored:

- an approach that enables simulators to be instrumented with debugging support based on a clear representation of control flows.
- an architecture that enriches the (graphical) modeling environment with existing components for debugging purposes, including enhanced simulator and model-specific visualizations for debugging.
- a reference framework that may constitute a base for language engineers and toolmakers to develop their languages and tools with debugging support.

3.1. Overview of The Framework

In the framework, a MAS design meta-model named MM_{MASDL} defines the structure of the design language used to model the static structure of MAS. Existing MAS DSMLs have such perspectives often combined with behavioral representations. Therefore, if debugging capability for any existing MAS DSML is desired, refinement of the MAS DSML meta-model is required to obtain the relevant static structure. For example, MAS DSMLs such as DSML4BDI [19], DSML4MAS [79] and SEA_ML [26] can be classified as design languages, as they lack any behavioral diagrams. However, if a MAS DSML incorporates both structural and behavioral viewpoints, with the structural viewpoints utilized in the design language, then the behavior viewpoints, or some of them, can be integrated into the runtime language, as described below. An instance of this meta-model is called the MAS design model (M_{MASDL}). Users can model the entire structure of the system using these instance models. These models are specifically designed and constructed by the users. In the design model, various concepts such as agents, roles, capabilities, plans, and events, along with the relationships between these concepts, are represented and modeled.

It is worth indicating that by saying user, here, we refer to the MAS language developers who may benefit from MASDebugFW for design and implementation of their own languages.

Next, a metamodel called MAS runtime language metamodel MM_{MASRL} is introduced to support modeling within the framework, representing the runtime states

of the previously discussed design models. This metamodel enables the description of the dynamic behavior and changing states of the MAS during its execution.For example; meta-entities such as $currentPlan$, $nextPlan$, $currentAction$, $nextAction$, and $currentBelief$ are possibly included in this meta-model. The runtime model (M_{MASRL}) of the system corresponds to an instance of MM_{MASRL}. This model serves to depict the real-time states of the system during execution. It is derived from the design model, signifying that it captures the ongoing states of the MAS as it operates. Essentially, these models act as snapshots, expressing the instantaneous representations of the MAS states during runtime.

Thirdly, there exists a metamodel for the MAS Simulation Environment Modeling Language (MM_{MASEML}), which enables the modeling of the simulation environment where the MAS model is executed. Instances (M_{MASEML}) of MM_{MASEML} are used to represent MAS simulation scenarios. Essentially, this model describes the behavior and characteristics of the simulation environment. Various aspects of the simulation environment are captured by this model, such as the outcomes of agents' actions within the environment, events that occur, communication conditions between agents, resource access, and resource utilization scenarios. In essence, this simulation environment model expresses the conditions under which the MAS will be tested during the simulation process. It provides a comprehensive representation of the environment's dynamics and serves as a crucial element in conducting MAS simulations and evaluating the system's performance.

Finally, a MAS visualization language metamodel, MM_{MASVL} allows the creation of customized models that graphically show the relevant parts of the MAS to improve understanding of MAS models by developers. These models (M_{MASVL}) are requested by the developer, so they are not compulsory to debug MAS models. Obviously, this part of the framework can be used optionally depending on the request of the developer. For example, if a developer is only concerned with communication between specified agents, a simulator can provide a visual representation of a complete simulation trace to present the communication step by step. Also, M_{MASVL} instance can be generated not only for visualizing the simulation trace but also for internal views of an agent such as plan execution.

The conceptual framework proposed in this study offers debugging support for MAS designed using MAS DSML models. These models are created by agent developers during the system design phase, before the actual implementation takes place. The primary goal of this framework is to minimize faults in the resulting MAS during development. According to MASDebugFW, the runtime model is generated by initializing it from the design model through model-to-model transformation. This transformed runtime model represents the initial state of the MAS. Simultaneously, the simulation environment needs to be modeled separately using the simulation environment modeling language. The simulator takes the simulation environment model and the runtime model as inputs. It then utilizes these models to generate subsequent states of the MAS during its execution. The simulator modifies the runtime model based on the specified conditions and interactions within the simulation environment, allowing for the representation of the evolving behavior and states of the MAS over time. In fact, the Runtime model and the Simulator together provide the operational semantics of this system. Of course, the operational semantics will also be determined by either how both of them will be implemented, and the limits of a system to be implemented according to this framework. For example, a synchronous simulator will only allow us to analyse a synchronous way to debug a MAS , e.g. executing each agent's actions sequentially. On the other hand, the detail level of the runtime model will express how detailed the system can be executed. The responsibility of initiating and using the simulator for debugging operations lies with the user. In MASDebugFW, the user takes on the role of controlling the simulation and debugging processes using the simulator. The subsequent subsections of the paper likely discuss in detail how the user can interact with the simulator to perform debugging tasks effectively. This user-centric approach empowers developers to actively engage with the debugging process, facilitating a more thorough understanding of the MAS's behavior and aiding in the identification and resolution of potential issues.

If a developer of a MAS DSML intends to adapt an existing MAS DSML to MASDebugFW, (s)he needs to derive the abstract syntaxes of the sub-languages required by the framework. Metamodels for these sub-languages can be created based on

the existing DSML metamodels. To achieve this, the DSML metamodel should be divided into appropriate parts, such as static/dynamic and intra-agent/MAS organization, and the developer should decide which parts can be equivalent to abstract metamodels of the framework. In this way, the developer transforms these parts into the metamodels required for the framework and reshapes the existing DSML in accordance with the framework. However, it should be noted that after this process, some changes may be required to make them consistent. These steps lead to the creation of initial versions of the meta-models for the sub-languages. Once the abstract syntax of the sub-languages is established, the developer must define transformation rules between these sub-languages. These transformation rules enable the conversion of design-time models to runtime models and facilitate the simulation and debugging process. Next, the simulator should be implemented, ideally using a model-to-model transformation approach, theoretically speaking. This means that the simulator can be developed by transforming models from one representation to another, allowing the simulation of the MAS based on the established transformation rules.

Once MASDebugFW is concretely implemented for a specific MAS DSML, it can be extended and reused for another MAS DSML through re-engineering the model transformations. By establishing transformations between the new DSML and the runtime language of the DSML that already incorporates the framework, debugging capabilities can be inherited for the target DSML. The process of adapting the framework to a new DSML can also include horizontal transformations, such as transformations between the new DSML and the design language. Readers may refer to [80] for an example implementation of horizontal transformations between different MAS DSMLs. A similar approach can be followed here to support the interoperability of MAS DSMLs and hence extend their debugging capabilities. Consequently, debugging functionalities can be enabled for the target DSML when the framework has already been applied to the source DSML during the transformation process. This approach enhances the efficiency and adaptability of the framework, allowing it to be effectively deployed for different MAS DSMLs without the need to develop a completely new implementation for each one.

3.2. Debugging Operations

In this section, we will explore various debugging operations that can be implemented within the simulation environment, as per MASDebugFW. In this framework, debugging operations are categorized into two fundamental classes: steps and breakpoints. By leveraging these two basic classes of debugging operations, developers can effectively monitor and analyze the behavior of the MAS during simulation, helping them identify and resolve potential issues, and ensuring the system functions as intended.

In code debugging, users frequently employ code stepping to comprehend how system states change during execution. This approach provides a detailed representation of the system's behavior, offering insights into its workings. When adapting such an approach to debugging models, stepping on the model can be seen as transitioning the current model state to the next state based on the provided operational semantics. Developers can specify the granularity at which they want to observe the MAS's behavior during simulation. For instance, they can step through the execution state by state, allowing them to analyze the MAS's behavior at each discrete step. This level of control aids in identifying issues and understanding the system's dynamics during runtime. The runtime states of the program can be likened to a call hierarchy since the coding structure in existing agent programming languages/environments often shares similarities with the object-oriented paradigm. In its simplest form, software agents can invoke plans triggered by events in the environment to achieve their goals, and these plans may call sub-plans when needed. Similar to the object-oriented paradigm, stepping over code can generally be accomplished using three possible operators: step into, step over, and step out. Debugging operations related to steps involve controlling the execution flow of the simulation environment. The debugging framework's definition of these operators is as follows:

> **Step into:** This operator allows the user to delve into the details of the model execution, navigating into the sub-plans and exploring their internal behavior step by step. As commonly accepted in the object-oriented paradigm, an object serves as an encapsulation unit that hides its current state and includes

both data and methods. Similarly, in the context of agents within the MAS, which comprise beliefs, goals, and plans, they can be viewed as encapsulation units. These encapsulated elements represent composite elements that contribute to an agent's presence within the model. When referring to "Step into" in the debugging framework, it means traversing through the model element containing any composite element defined within it. For instance, when the user attempts to step into an internal state of an agent, this action involves exploring all possible plans, actions, or sub-plans within the selected plans from the pool of all possible plans associated with that agent. By stepping into the internal states of an agent and comprehending its beliefs, goals, and plans, developers gain valuable insights into the agent's behavior and decision-making processes, thus enabling effective debugging and analysis of the MAS model during simulation.

Step over: The step-over operator enables the user to execute the current plan without diving into its sub-plans. This way, the focus remains on the broader picture of the model's behavior."Step over" and Step into" are complementary concepts in debugging. "Step over" refers to moving through the model at the composite level, acting as a filtering process during debugging. However, this doesn't mean that underlying elements beneath the composite level are not executed; they are still executed but hidden from the user. For example, when stepping over an agent's plan, the user observes the resulting state at the end of plan execution without seeing the execution of sub-elements (e.g., actions or sub-plans). This approach allows users to control the level of detail they observe during debugging, ensuring a more efficient debugging process.

Step out: The step-out operator lets the user return to the parent plan, exiting from the current plan's execution. It allows for a seamless transition back to higher-level behaviors in the model. It refers to transitioning from the inner composition level of a model element back to the element's composition level. For instance, when the user steps into an agent's plan and navigates inside the plan, they have the option to step out and return to the plan's composition

level. When this occurs, the simulation continues at the composite level, and the finer details of the plan are concealed from the user. This capability allows users to control the level of granularity during debugging, focusing on higher-level behaviors while hiding the complexities of individual plans or actions.

These stepping operators in the debugging framework offer users a powerful set of tools to analyze and comprehend the behavior of the MAS models during simulation, facilitating effective debugging and troubleshooting.

Breakpoints may be compared to basic assertions in programming. They serve as checkpoints set by developers at specific locations in the code or model, causing the program's execution to pause during debugging. When the program reaches a breakpoint, it temporarily halts, allowing developers to inspect variables, data, and the system's state at that particular point. If a condition or assertion at the breakpoint fails to hold true, the debugger interrupts the program's execution. When the execution of a program is interrupted, it means that an assertation in this program fails. Generally, in the breakpoints concept, this situation occurs when the execution of the program reaches a specific code line. When it comes to debugging models within MASDebugFW, "interruption" refers to pausing the simulation. In this context, users can set specific breakpoints in the model, which cause the simulation to pause when certain conditions are met. The framework introduces three possible types of breakpoints:

State based: It is the situation where the simulation pauses when the model reaches a predefined state or a specific pattern within the state. This can be thought of as a kind of temporal breakpoint formalism that improves on classical debugging expressions to reason not only about the current state of the system, but also about its past and future. For example, it can be thought that a certain event occurs, and an agent triggers a plan accordingly, a certain agent reaches a belief pattern, or a predefined communication pattern is captured among the agents.

Condition based: In the model, when certain properties of an agent are defined, they introduce specific logical conditions that are checked against these

properties. For instance, if the agent's beliefs are no longer valid due to changes in the environment facts, the simulation can be paused based on this condition. This allows developers to identify critical points during the simulation where certain properties or beliefs of agents are affected by changes in the environment, leading to a pause in the simulation for further examination and debugging.

Time based: In certain scenarios, the simulation halts when a specific pre-arranged time is attained within the simulation environment. However, this behavior is closely tied to the predefined time settings (e.g., real-time, scaled real-time, as fast as possible, timeless - discrete event-based) specified for the particular simulation environment being utilized.

By incorporating these three types of breakpoints, MASDebugFW provides users with flexible and granular control over the debugging process, enabling them to analyze and troubleshoot the model's behavior effectively during simulation.

3.3. Execution States

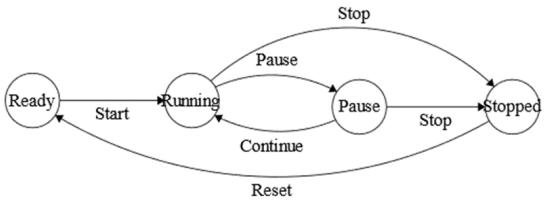


Figure 2: Execution States transition graph of the simulator.

A simulator should offer the capability to switch between different execution states, allowing the user to have control over the simulation. This way, the user can intervene in the simulation process as needed. They can choose to stop, pause, or continue the simulation of the runtime model at any point during the simulation duration. This level of control empowers the user to actively interact with the simulation and provides flexibility for effective debugging and analysis of the MAS model

in real-time. To facilitate user control over the simulation, the framework defines four different execution states, each with distinct characteristics. These execution states allow smooth transitions, providing the user with flexibility during the simulation process. The different execution states and transitions are shown in Figure 2. These execution states are briefly described as follows:

Ready: This state corresponds to the default state of the runtime model and serves as the initial state of the simulation. It mirrors the starting point of the MAS as well. For instance, this mode encompasses an initial snapshot of the agent's beliefs, goals, and plans, capturing the initial state of the agents in the system. When the simulation begins, it starts from this default state, providing a foundation for the subsequent execution and behavior of the MAS.

Running: Once the user initiates the simulation, the simulation environment transitions to Running state. In Running state, the environment can switch to all simulation states except Ready state. However, this switch is only possible when the simulation is interrupted. The interruption can occur due to encountering a breakpoint or when the simulation execution is controlled step-by-step. In the Running state, each MAS agent executes atomic events, which represent actions in an agent's plan. This execution continues until the simulation is interrupted by a breakpoint or a user-initiated step-by-step control. In essence, Running state represents the active execution state of the simulation, where the agents perform their respective actions until a pause point is reached or the user manually interrupts the simulation process.

Pause: During the ongoing simulation, the user has the option to switch the simulation environment to the Pause state. In Pause state, the simulation freezes, allowing the user to examine the system's current state in detail. The user can choose to resume the simulation at any time, switching back to Running state. While in Pause state, the user can access a snapshot of the current MAS state. This snapshot includes execution traces of each agent's plans, communication traces between agents, and other relevant information. This

feature allows the user to analyze the MAS behavior and interactions at a specific moment, providing valuable insights into the system's behavior during the simulation process.

Stopped: When the user intends to terminate the simulation, they should switch it to the Stopped state. In Stopped state, the simulation can only be switched back to the Ready state. At this point, the simulator converts the current MAS model back to its initial state. Once the simulation is stopped and converted to the initial state, it is not possible to resume or continue the simulation from that moment onwards. In summary, stopped state marks the end of the simulation, and any further simulation activities can only start from the Ready state, using the original, unaltered initial state of the MAS model.

By offering these distinct Execution States and seamless transitions, the framework grants the user full control over the simulation process, facilitating effective debugging and exploration of the MAS model at their own pace.

3.4. Model Simulator

In MASDebugFW, the operational semantics of the modeled MAS is specified within the runtime language metamodel and interpreted by a simulator. There are various options available for defining the simulator, with one of the most preferred approaches being directly modifying the runtime model based on the model transformation rules to achieve the next state of the system. However, the general structure of simulation algorithms often follows a high-level abstraction. Algorithm 1 listed above represents the abstract pseudo code of the simulator proposed in the framework. The simulation algorithm is implemented using the following functions:

initialize: The given function is responsible for generating the initial state of the simulation. This initial state is obtained through a model-to-model transformation process, where the instance of the MAS design language, which needs to be simulated, is transformed into the MAS runtime model.

terminationCondition: The function takes the runtime model, simulation environment model, and time as inputs and returns "true" when the desired simula-

Input: to be simulated Model (M), Simulation Environment Model (E)

Output: Runtime Model (RM), time (t)

t, $RM \leftarrow$ initialize(M,E) ;

while *not terminationCondition(RM,E,t)* **do**

  foreach *agent* $a_i \in RM$ **do**

    $a_i \leftarrow$ updateBeliefs(RM,a_i,E);

    $a_i \leftarrow$ updateGoals(RM,a_i,E);

    $a_i \leftarrow$ updateActivePlan(RM,a_i,E);

    $a_i \leftarrow$ nextAction(RM,a_i,E);

  end

  $t \leftarrow$ increaseTime(RM,t);

  if *checkForBreakpoints(RM)* **then**

    pauseSimulation();

  end

end

Algorithm 1: Generic MAS Simulation Algorithm

tion termination condition is met, such as when the user requests termination or when a predefined time limit is reached during the simulation.

updateBeliefs: This function is responsible for updating an agent's current beliefs based on the runtime model and the current state of the simulation environment model.

updateGoals: After updating the beliefs of an agent, the simulator needs to determine which goals the agent desires to reach. To achieve this, the simulator invokes the function responsible for updating the runtime model. This function updates the runtime model, reflecting the agent's updated beliefs and goals, and prepares the system for further simulation steps.

updateActivePlan: The given function employs a reasoning mechanism to determine a plan for an agent to achieve its goals. If the agent already has an ongoing plan, the function waits for the current plan to finish before proceeding, ensuring the integrity of the agent's actions. Once the active plan is completed, the function then sets a new active plan based on the agent's current goals and the reasoning process. This approach enables the simulator to manage the agent's actions in an orderly manner, ensuring that the agent's plans are executed sequentially and in accordance with its objectives.

nextAction: The given function is responsible for determining and executing the next action of an agent based on the active plan. Using the active plan as a guide, the function identifies the subsequent action that the agent should perform to progress towards achieving its goals. It then carries out the execution of this action, allowing the agent to move forward in its plan and towards fulfilling its objectives within the simulation environment

increaseTime: The provided function advances the simulation time while adhering to the time semantics defined in the simulation setup. The time semantics can be predetermined and set as fixed during the simulator implementation. However, the flexibility exists to adjust the time semantics through the simulation environment modeling language, allowing users to customize and mod-

ify the time progression rules according to their specific simulation requirements. This dynamic time management capability enhances the adaptability of the simulator to various simulation scenarios.

checkForBreakpoints: The mentioned function facilitates the logical checking of breakpoint conditions to determine if the simulation should pause. It evaluates the specified conditions associated with breakpoints at various points during the simulation. When a breakpoint's condition is met, the function triggers a pause in the simulation, allowing developers to inspect the system's state and perform debugging operations at the designated breakpoint locations. This capability enhances the debugging process, enabling users to intervene and examine the simulation at critical moments during the simulation's execution.

pauseSimulation: The given function is responsible for pausing the simulation. Once the simulation is paused, it awaits a new signal from the user to resume the simulation. This allows the user to take control of the simulation, inspect the current state, perform debugging operations, and proceed with the simulation at their own pace. The pausing and resuming feature empowers the user to interact with the simulation environment effectively, providing a flexible and user-friendly debugging experience.

By utilizing these functions, the simulator can efficiently interpret the operational semantics defined in the runtime model and execute the simulation, advancing the MAS through different states and time steps, while providing relevant status updates to the user. The simulator uses the operational semantics defined in the initial runtime model to guide the simulation process. During the simulation, the runtime model evolves dynamically to reflect the updated states of the system, which are then used as output for further analysis.

3.5. An Implementation of MASDebugFW

In order to understand the validity and adequacy of the framework, a MAS DSML was developed from scratch based on the above-discussed framework. According

to the framework, a design language is needed first. In this context, the Simplified Agent Modeling Language (SAML) was developed as the first step of implementation. Defining the abstract syntax of the language is the first step in the development of this modeling language. While developing the abstract syntax, an effort has been made to show the existence of the basic elements of MAS and the interaction of agents with an environment in the simplest way. Here, the abstract syntax was developed based on the MAS modeling concepts presented in Tezel et al. [25]. Then, the concrete syntax of the modeling language was created. While constructing the concrete syntax, the findings of studies [81, 82] examining MAS DSMLs from a usability perspective were utilized. As a second step, the syntax and the operational semantics of the MAS Runtime Modeling Language (MASRL) were created and the rules for model-to-model transformation between SAML and MASRL were derived. Thus, a statically designed MAS is transformed into a dynamic state. Although the model-to-model transformation code can be given in detail, it is not emphasized separately due to the fact that this transformation is not directly related to the contributions of the study. Similarly, the details of the transformations made are not given throughout this paper. As the third step, MAS Environment Modeling Language (MASEML) was developed. Abstract syntaxes of MASEML and MASRL have references to certain elements of each other. The implementation of the MAS visualization language proposed in the general framework has not been implemented at this stage, as it is optional and left to the discretion of the developer. However, it can be thought that this language, in its simplest form, could consist of a series of diagrams showing the exchange of messages between agents, or a summary diagram that instantly shows the internal structure of an agent. Although a MAS visualization language was not created based on the framework, a diagram was created that can summarize the internal structure of any factor while applying MASRL to understand the contribution of this part of the framework to debugging. The language development approaches introduced in Tezel et al. [20] and Kardas et al. [19] were adopted here in the engineering of the above mentioned MAS languages. The general view of the presented application is illustrated in Figure 3.

The concrete implementation of the debugging framework was realized by using

the Sirius tool [2] which enables developing various specific graphic modeling environments according to Eclipse Modeling technologies within the Eclipse ecosystem.

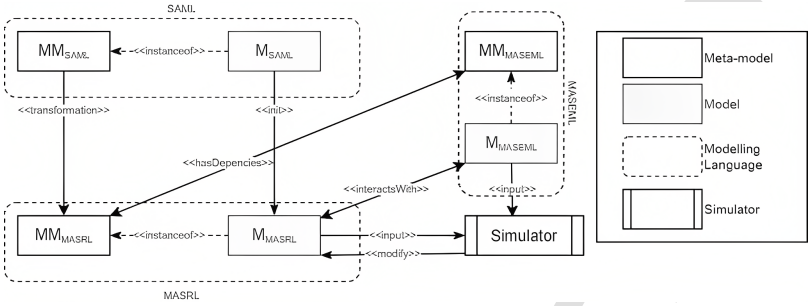


Figure 3: Overview of the implementation of MASDebugFW.

3.5.1. Design Language

The SAML was developed to be used as the design language of the framework. As seen in Figure 4, the SAML meta-model includes some elements that can be used to model the interactions of agents with the environment, as well as the basic elements required for a MAS. In the metamodel, there is a MAS meta entity representing the system which is needed to be modeled. The MAS must have at least an $Agent$ and an $Environment$ meta entity. It can also have $Capability$ meta entities to represent the capabilities of the agents exist in the system. $Capabilities$ within the system can interact with one or more environments. In addition, each $capability$ can have sub-capabilities and $Plan$ and $Belief$ meta entities. Plans are triggered by events, which are represented by the $Event$ meta entities, that occur within the system. Each plan also has a context and, actions represented by the $Action$ meta entity. Actions can use functional parts represented by the $Operation$ meta entities in the environments in which they interact. Thus, new events can be triggered in the environment.

To provide a clearer understanding of the underlying concepts used in SAML, we

[2] Eclipse Foundation, Sirius, https://www.eclipse.org/sirius/

briefly define key meta-entities that serve as a foundation for agent-based systems. Interested readers may refer [23] or [19] for an extended discussion of these MAS domain concepts. A capability refers to an agent's ability to perform specific actions or tasks, enabling it to achieve its goals. A belief represents the information or knowledge an agent holds about itself or its environment, which influences its decision-making processes. The environment is the external context in which agents operate and interact with other agents or entities. A goal defines the desired state or outcome that an agent strives to achieve, guiding its behavior and decision-making. A plan is a structured sequence of actions that an agent executes to accomplish a specific goal. An event represents a change or occurrence in the environment or the system that triggers specific agent behaviors or reactions. An action is an atomic operation performed by an agent as part of its plan execution, affecting its internal state or the environment. Lastly, an operation is a behavior that, when executed, can generate updates to observable properties and trigger specific observable events, allowing agents to interact dynamically with their environment. These meta-entities are core components of SAML and are integral to modeling agent behaviors, interactions, and decision-making processes.

The concrete syntax of the language consists of 4 different diagrams representing 4 different perspectives. These are MAS, Environment, Capability and Plan diagrams. Graphical concrete syntax elements of the language and their notations are given in Figure 5. The MAS diagram, which expresses the general structure of the system, is responsible for the modeling of the agents, capabilities and environments of the system and the interaction between them. The Environment, Capability and Plan diagrams are responsible for modeling the internal structures of the environment, capabilities, and plans, respectively. Examples of a MAS diagram, a Capability diagram, and a Plan diagram are shown in Figure 6, representing different perspectives of the same MAS model.

3.5.2. Runtime Language

The MAS Runtime Modeling Language (MASRL) was developed with a metamodel that contains the operational semantics of a MAS and is shown in Figure 7.

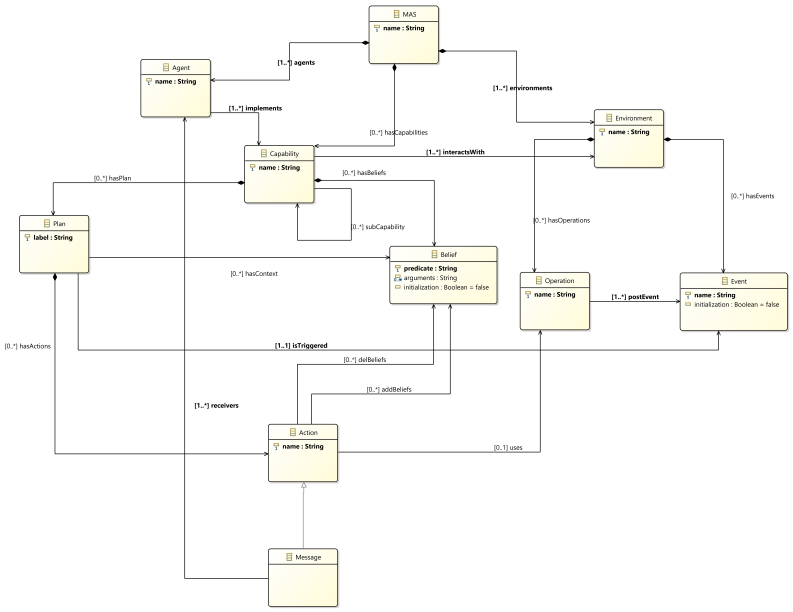


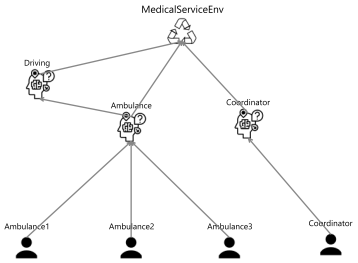
Figure 4: The meta-model of the SAML.

Element	Notation	Element	Notation
Action		Event	
Agent		MAS	

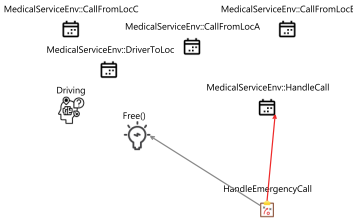
Element	Notation	Element	Notation
Belief		Message	
Capability		Operation	
Environment		Plan	

Figure 5: Graphical concrete syntax elements of the SAML language and their notations.

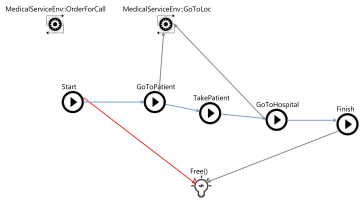
Although the metamodel of MASRL has similar meta entities with the metamodel of SAML, some new properties to the meta entities and different relationships between



(a) MAS Diagram Example



(b) Capability Diagram Example



(c) Plan Diagram Example.

Figure 6: Examples of a MAS diagram, a Capability diagram, and a Plan diagram

meta entities have been added. Thus, MASRL can contain operational semantics. The concrete syntax is defined in such a way that it can accurately represent the runtime, although notations similar to SAML were chosen, especially when considering icon selection. Instances are generated by transformation through the SAML instances. The instances created by the transformation reflect the initial moment of MAS that is intended to be included in the debugging process. Figure 8 shows an example of a model.

3.5.3. Simulation Environment Modeling Language

The simulation environment modeling language (MASEML) enables simulation parameters to be given to the simulator by modeling. The metamodel of MASEML is shown in Figure 9. It has two main meta-elements. These elements are $Scenarios$ and $Breakpoints$. There are three types of possible Breakpoints in the language:

StateBased Breakpoints: Simulation is paused when the model reaches a certain predefined pattern. To be more specific, when we consider the runtime

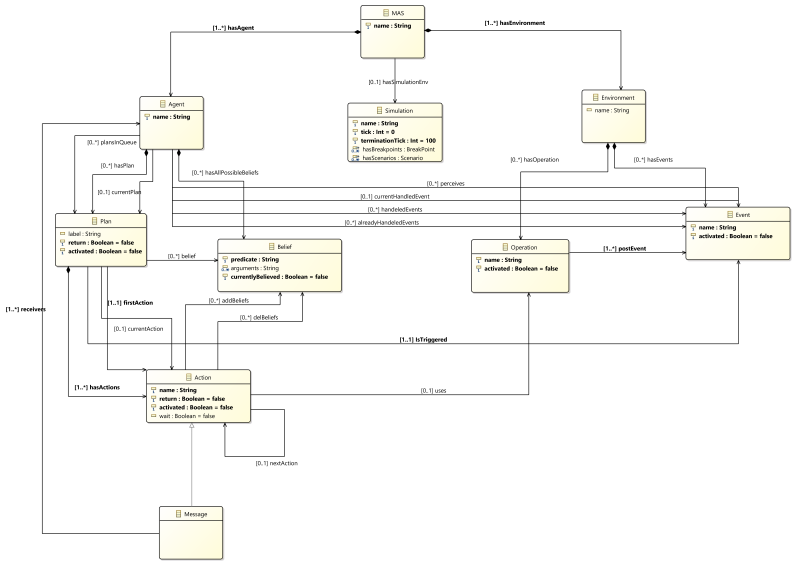


Figure 7: The meta-model of the MASRL.

model as a graph, the simulation is paused if a predefined sub-graph is reached.

ConditionBased Breakpoints: The simulation is paused in cases where the elements defined on the model provide a certain logical condition, such as the occurrence of a certain event, activation of a specific plan or an agent in the model believes in a certain fact about the environment.

TimeBased Breakpoints: It is paused in a simulated environment if a certain time is reached. However, this situation is directly related to the time definition of the simulation environment. In the implementation here, the time of the simulation is operated independently of the real-time as a sequence of events in time what is called discrete-event simulation.

Another important meta-element here is the scenario. Scenarios are parts where the simulation is modeled under which conditions it works. There are two types of elements that can be input into the system under a scenario. The first one tells when any event will occur in the simulation environment according to the time concept

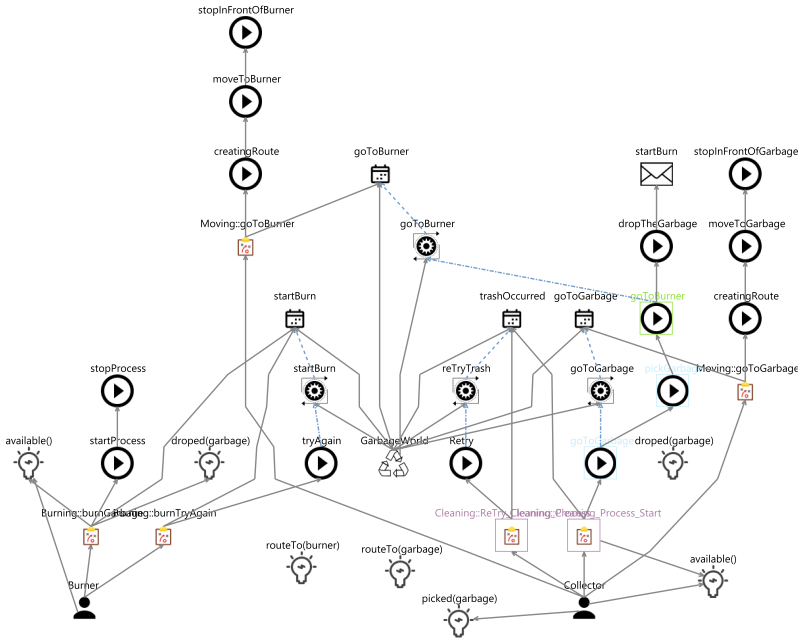


Figure 8: MASRL diagram example.

of the simulation. Second, it states under what conditions the actions in the plans of the agents whether would be taking place.

3.5.4. Simulator and Debugger

The simulator and debugger come embedded in the modeling environment developed according to MASDebugFW. Although the simulator represents a kind of model-to-model transformation approach that performs in-place transformations to create the MASRL instance in the next step, we coded a Java-based simulator due to the existing model-to-model transformation tools being too generic to implement this kind of simulator. A screenshot of the debugger is given in Figure 10. The simulation environment has 4 different Execution States as stated in the framework. These are Ready, Running, Pause, Stopped, respectively. The first state of the system is the ready state. After the simulator starts simulating, the system switches between

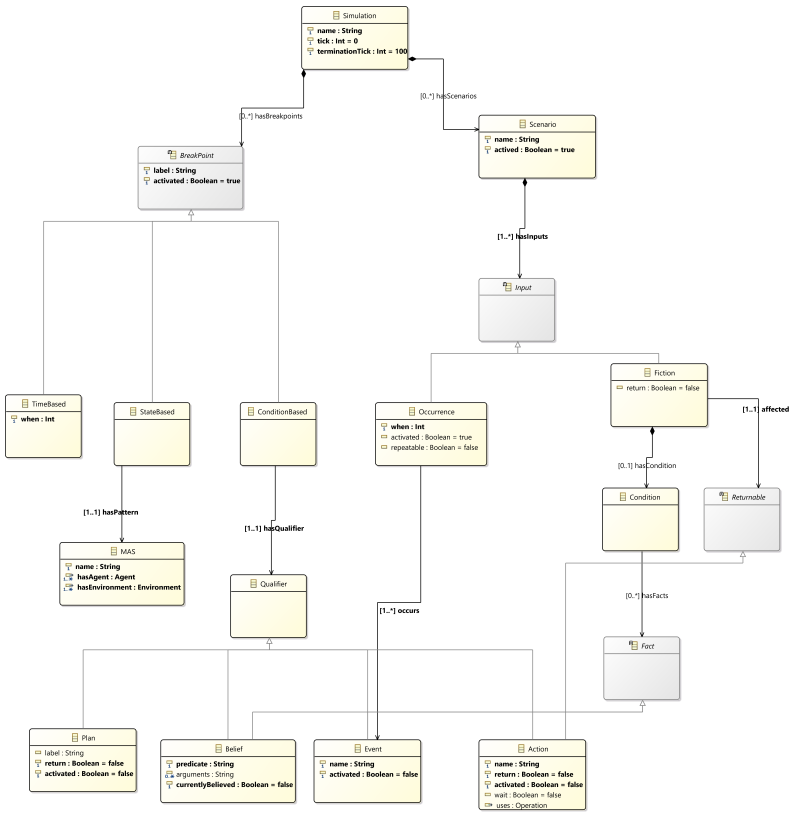


Figure 9: The meta-model of the MASEML.

the two state, these are Running and Pause state. While the user can put the simulator into Running state at any time, (s)he can enter Pause state with breakpoints or steps. The user can switch to the last state, Stop state, at any time. In this case, the simulation terminates, and the system returns to the initial state.

In the next section, a case study of MASDebugFW is presented. We also prepared a video for interested readers to see how to use it (https://youtu.be/3MKDgob9ij4). Additionally, the source code and bundle of the implemented framework can be found in the Mendeley repository [83].

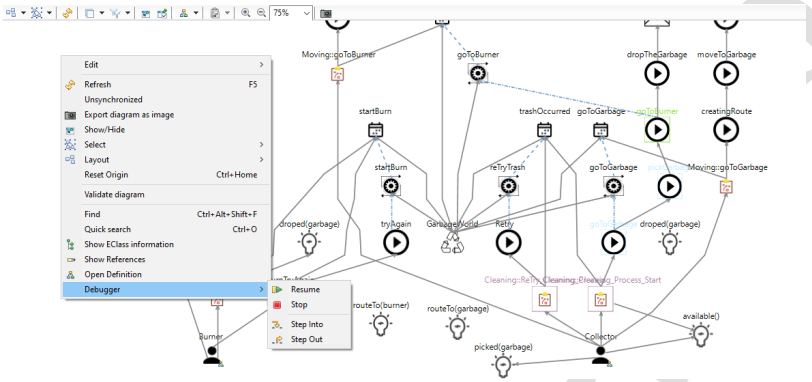


Figure 10: A screenshot from the simulator and debugger

4. Case Study

The following subsections discuss the details of the design and debugging of an agent-based Garbage Collection System both using MASDebugFW and benefiting from the implementation discussed in Section 3.5.

4.1. System Design with SAML

The development of a Garbage Collection MAS is widely used to illustrate MAS design and implementation in the AOSE field (for example [19, 84, 6, 85, 86]). There are two types of agents called Burner and Collector in this system, which allows agents to collect and dispose of garbage in an environment collectively. The first task of the Burner agents is to report the location of garbage to the Collector agents. The other task of the Burner agents is to burn the garbage brought by the Collector agents. Collector agents go to the location of the garbage reported by the Burner agents, pick up the garbage and bring it to the Burner agents. After the garbage is delivered, the Collector agents must wait for the messages from the Burner agents about garbage appearing in the environment.

The process is started by creating a design model based on different perspectives in accordance with MASDebugFW. In Figure 11, a screenshot from the IDE containing the MAS diagram of the garbage collection system is given. Developers can cre-

ate a design model that conforms to the system's SAML specifications by dragging and dropping the necessary items from the palette on the right side of the modeling environment. In this way, the main assets of the system such as agents, capabilities, and environments are determined in the MAS diagram. For example, the Garbage Collection MAS model includes the two agents, the three capabilities, one of which is sub-capability of the other, and the one environment.

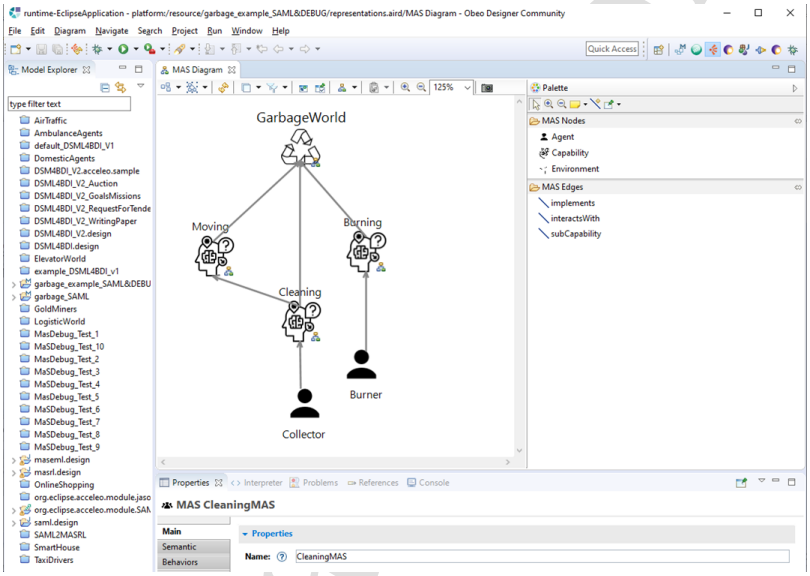


Figure 11: MAS model of the garbage collection system.

As noted in the Section 3.5, SAML has three different diagrams apart from the MAS diagram. One of them is the environment diagram. In this diagram, the events that should occur in the environment and the operations that may trigger these events are modeled. The environment diagram of the garbage collection system is given in Figure 12 . Four different events can occur in the system and four different processes can trigger each. According to the relevant case study scenario, events other than *trashOccured* should appear in the environment only when operations are triggered, and naturally, *trashOccured* should also occur spontaneously in the environment.

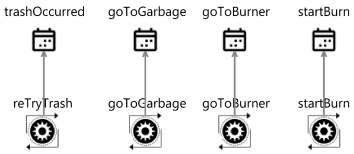


Figure 12: Environment diagram of the garbage collection system.

When each of the environments is defined in the corresponding diagram, the internal structure of each capability should be modeled. Figure 13 shows an example model in the capability diagram for the Cleaning capability. Two different plans have been modeled here to deal with the *trashOccured* event. Also, in one of the plans, there is a prerequisite that the relevant agent believes in the *available()* belief. Additionally, the capability given here has a sub-capability called *Moving*. Beliefs created to be used in modeled plans are also seen in the capability viewpoint. Although plans for a single event are modeled on the corresponding diagram, other events defined in the environment in which the capability interacts appear in the diagram. Therefore, the user can model the plan for different events if (s)he wishes.

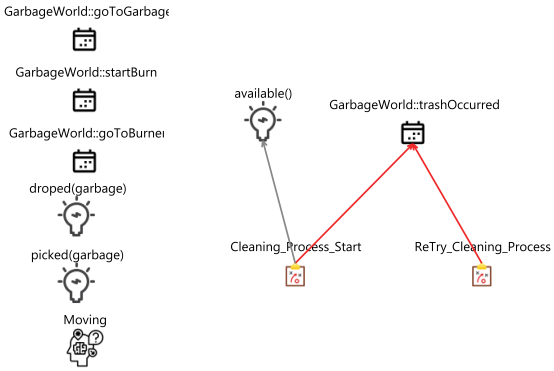


Figure 13: The capability diagram of the *Cleaning* capability.

With the help of the Plan diagram shown in Figure 6, the internal structure of any plan defined in a capability can be modeled. For example, the *Cleaning_Process_Start* plan given in Figure 14 shows a sequence of actions and messages. As can be seen

here, actions have relationships with belief and operation elements. For example, if action is related to an operation, it causes another event to occur in the environment when the relevant action takes place. The actions can also lead to a certain belief being believed or not believed by the agent. While the consequences of actions affect the agent performing the action, in the case of messages, which are a special type of action, the consequences affect the agent receiving the message.

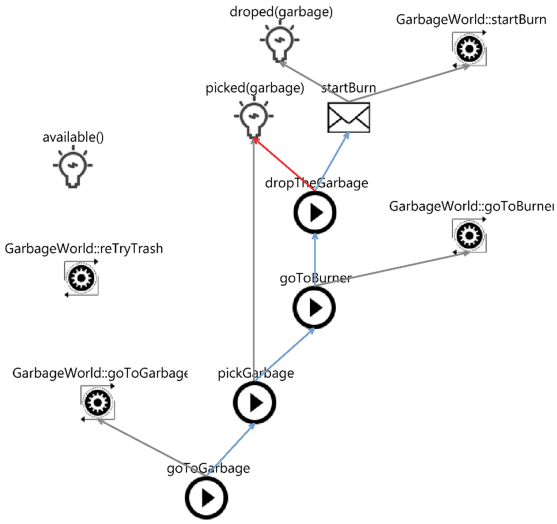


Figure 14: The plan diagram of the *Cleaning_Process_Start* plan.

4.2. Debugging Modeled MAS

Based on MASDebugFW, model-to-model transformations are applied for transforming the model created in the design language into an instance of the runtime language. In the example here, MAS modeled in SAML has been transformed into MASRL instance shown in Figure 15. However, before starting debugging the model, the MASEML model instance should be modeled where breakpoints and scenarios are modeled. As mentioned before, there are 3 types of break points. These are *TimeBased*, *ConditionBased*, and *StateBased*.

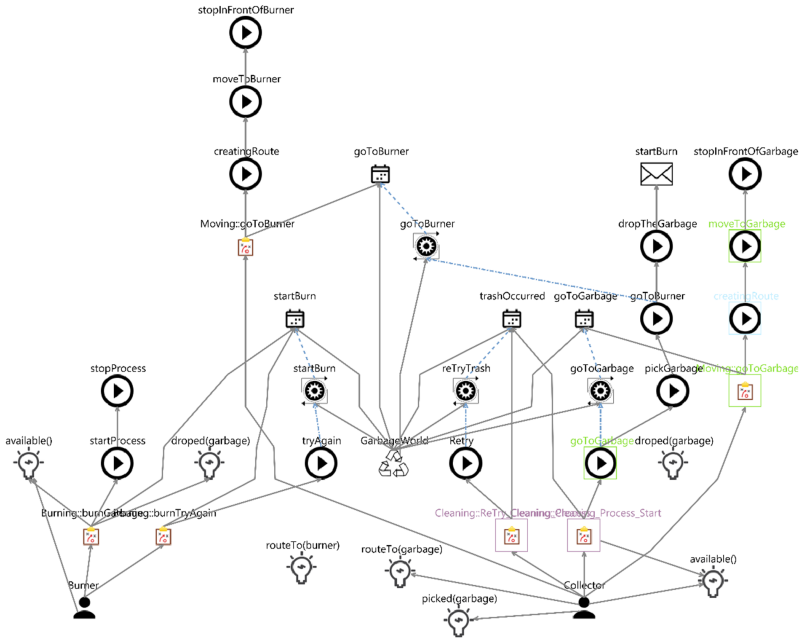


Figure 15: MASRL model of Garbage Collection System.

While it is easy to define Time-Based and Conditional-Based breakpoints in the corresponding diagram with only drag-and-dropping from the palette, for a State-Based breakpoint, a subgraph of a MASRL instance should be created. As an example of the state-based breakpoint, when the plan *goToGarbage* is executed successfully, the status of the plan *Cleaning_Process_Start* waiting in the planning queue of the *Collector* agent is shown in Figure 16.

In Figure 17, the scenario diagram, which is the other important element in MASEML, is shown. Here, it is possible to model the occurrence of a defined event in the environment once or for certain periods, or the conditions under any action which could be executed or not.

For example, consider Figure 17, for the *dropTheGarbage* action to take place, the relevant agent must believe in *picked(garbage)* belief. However, as a result of the *pickGarbage* action that took place in Figure 18, the agent does not believe picked(garbage)

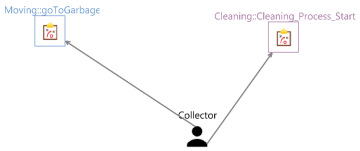


Figure 16: State-based breakpoint example.

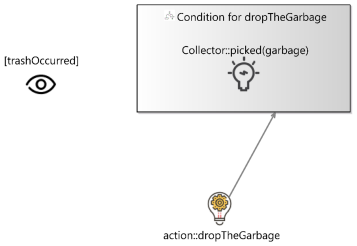


Figure 17: Scenario diagram example.

belief as it should. In this case, the *dropTheGarbage* action that is in the same plan will not be able to take place due to the scenario given in Figure 17 and the related plan will fail. Figure 19 illustrates this situation. When confronted with such a situation, it is necessary to make a design change in the design model so that the *pickGarbage* action adds the *picked(garbage)* belief into the mental state of the agent.

5. Empirical Evaluation

In this section, we evaluate MASDebugFW both qualitatively and quantitatively. Quantitative evaluation was carried out in 10 sessions with participants volunteering to participate, 6 days apart. The participants were asked to perform debugging activities over the given case study models and the data regarding the debugging performances of the participants were recorded. Qualitative evaluation was made with a questionnaire based on the technology acceptance model. All the results obtained were analyzed statistically.

5.1. Quantitative Evaluation

A quantitative evaluation was conducted to empirically determine whether MASDebugFW could significantly improve MAS modeling and debugging in terms of time efficiency, error localization accuracy, diagnostic precision, and the ability to make acceptable fixes. In the quantitative evaluation, a within-subject experimental design was employed to ensure consistency across participants and to minimize variability caused by individual differences. This design allowed each participant to

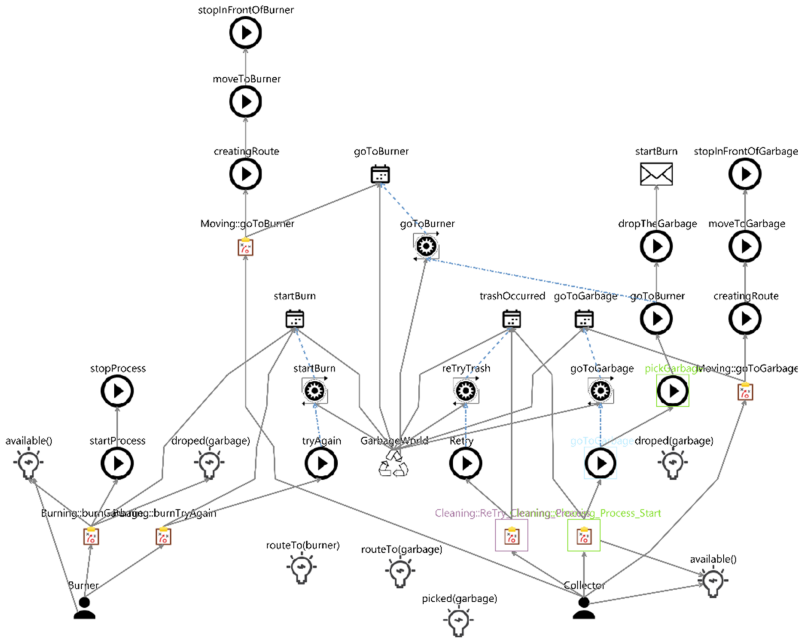


Figure 18: The moment when the pickGarbage action occurs in the MASRL model of the Garbage Collection System.

be exposed to all conditions, ensuring a robust comparison of the experimental factors. This approach was chosen to provide a direct measurement of MASDebugFW's impact on MAS modeling and debugging tasks. The experiment was designed with 10 case studies, created by experts experienced in MAS and DSML development.

The participants were assigned tasks to identify, diagnose, and fix faults in MAS models. In each session, participants alternated between using the debugging tools developed as part of MASDebugFW and debugging without these tools. The order of tool usage was randomized to prevent learning effects from biasing the results. In each session, participants were divided into two groups: one group used the debugging tools developed under MASDebugFW, while the other group performed the debugging tasks without these tools. This allowed for a controlled comparison of performance metrics, including time spent, errors localized, and fixes applied. The

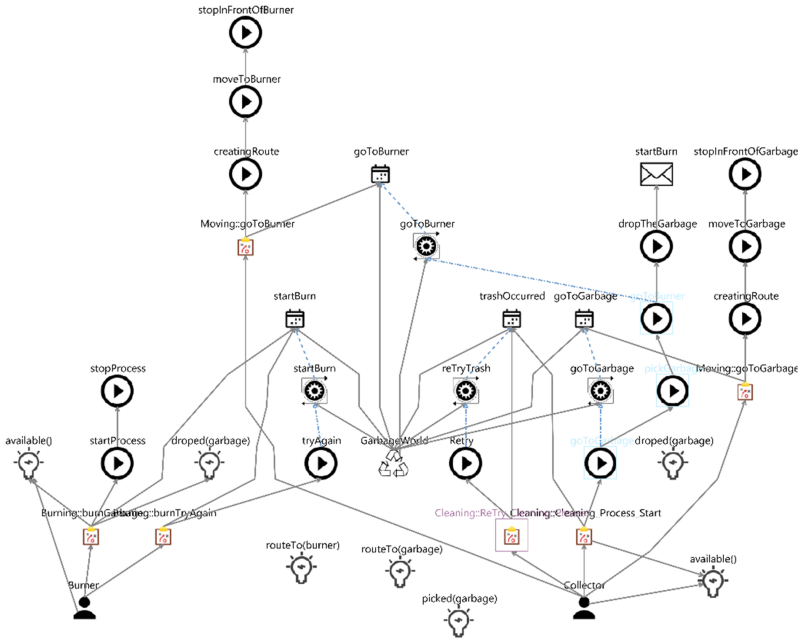


Figure 19: The moment when the dropTheGarbage action occurs in the MASRL model of the Garbage Collection System and the related plan fails.

assignment of participants to groups alternated across sessions to ensure a balanced evaluation.

5.1.1. Selection of Participants

Participating candidates are graduate students, academics, and/or software developers who have taken or are taking the postgraduate courses about MASs and DSMLs such as Advanced Software Engineering, Model-Based Software Engineering, Agent-Based Software Development, and MASs given at Ege University International Computer Institute and Computer Engineering Department. Persons involved in the development processes of the tool proposed in this study were not invited to participate in the experiments in order not to create bias and maintain impartiality. Invitations were sent to 8 candidates who met these criteria within the

scope of the relevant evaluation. However, 6 candidates accepted the invitation and volunteered to participate in the experiment. Consent forms were signed by all the volunteered participants. All participants have at least 1.5 years of MAS design and implementation experience, which includes applying AOSE methodologies and using at least one agent development API. In addition, all participants are familiar with at least one of the software engineering methodologies, mostly based on UML, and some of them have been working in the industry and have experience in using modeling tools and DSMLs to develop industrial products for more than 5 years on the average.

5.1.2. Hypotheses

The following 4 hypotheses were attempted to test with the experiment:

H_0^1: SAML and debugging tools (MASRL, MASEML, MASVL) developed in accordance with the debugging framework proposed in the study have no effect on the number of faults that can be detected during debugging.

H_0^2: SAML and debugging tools (MASRL, MASEML, MASVL) developed in accordance with the debugging framework proposed in the study have no effect on the number of faults to be diagnosed correctly during debugging.

H_0^3: SAML and debugging tools (MASRL, MASEML, MASVL) developed in accordance with the debugging framework proposed in the study have no effect on the fixing of faults that can be detected during debugging.

H_0^4: SAML and debugging tools (MASRL, MASEML, MASVL) developed in accordance with the debugging framework proposed in the study have no effect on the time spent for debugging.

The significance level to be used in statistical testing of the hypotheses given above was chosen as 0.01. The significance level here indicates that the risk of making an error of 0.01 (1%) in rejecting the hypotheses is accepted. Although it is desirable to keep the level of importance low, as the level of importance decreases, the claim put forward by the hypothesis will be very difficult to reject, even if it is

not true. For this reason, generally, the significance levels are preferred between 1% and 10% in order to ensure that the tests performed are meaningful [87, 88]. In the sessions held during the evaluation of the application of MASDebugFW, there were observations that the application was extremely effective. As a result, although it will be difficult to detect the differences with statistical tests, the significance level was accepted as 0.01 and it was desired to reduce the false positive probability as much as possible.

5.1.3. The Variables of the Experiment

Experimental variables are presented below as factors, non-factor independent variables, and dependent variables.

- **Factors**

  Debugging tool support is the only factor in the designed experiment. This factor is measured on a nominal scale and will have two levels: exist, not exist.

- **Non-factor Independent Variables**

  There are two independent variables kept fixed throughout the experiment. These are *MAS models* and *injected faults*.

  MAS is a paradigm used in modeling and implementing autonomous and complex systems. Therefore, it has been put forward with the expectation that all the examples proposed for evaluation will explain and represent the such complex and autonomous systems that may exist. There are 10 different MAS examples planned to be used in the experiment: Home Servant System, Garbage Collection System, Elevator Management System, Taxi Management System, Smart Home Management System, Online Store System, Air Traffic Control System, Paramedic and Ambulance Management System, Supply Chain Management System, Mine Management System.

  Brief descriptions of the scenarios are as follows:

  - Home Servant System: A MAS that enables service tasks in an environment. There are 3 different agents in the system to be implemented. The

"Robot" agent will respond to the beverage requests from the "Owner" agent. If it has a drink in its stock, it will serve the beverage to the "Owner" agent. Otherwise, it will order from the "Supermarket" agent. When its order arrived, it will serve this drink. Meanwhile, the "Owner" agent will request a new drink each time it runs out.

- Garbage Collection System: A MAS that collects and destroys garbage in an environment. There are 2 different agents in the system to be implemented. The "Collector" agent collects the garbage in the environment and brings it to the "Burner" agent. The "Burner" agent destroys the garbage.
- Elevator Management System: A MAS is responsible for operating elevators in a building. There are 3 different agents in the system to be implemented. The "ManagerAgent" evaluates incoming elevator calls as single or double floors and transfers them to the agents responsible for single or double floors. If the relevant agents are available, they go to the relevant floor, and if they are not available, they notify the "ManagerAgent". In this case, the "ManagerAgent" makes a request to the elevator agent again.
- Taxi Management System: A MAS that manages a taxi company, primarily answering customer calls and determining taxi routes. The related system is designed for the management and administration of taxis at a taxi stand. There are 4 different agents in the system to be implemented. The "Manager" agent responds to the taxi calls coming to the center and sends this call to the next taxi. If the "Driver" agents who receive the call, think that it is their turn and if it is available, they go to pick up the customer who is the subject of the call. If not, they report this situation to the "Manager".
- Smart Home Management System: A MAS that provides the entire management of a smart house. There are 2 different types of agents in the system to be implemented. The "Manager" agent constantly monitors

the smart house and orders the “Housekeeper” agent in the face of certain events, waiting for actions that will ensure the stability of the house in the face of this event. When situations such as the cooling or heating of the house, or the change in the day or night are conveyed to the “Housekeeper” agent, the agent takes some precautions regarding these situations.

- Online Store System: A MAS that enables customers to be accurately matched with real shops and shop safely in an online environment. There are 4 different types of agents in the system to be implemented. The “Customer” agent represents the customers who come to the online shopping system. It orders books, electronic goods, or furniture taking into consideration the customer’s needs. According to the type of order placed, the agent representing the appropriate store meets the relevant request, prepares, packs and ships the product, and informs the customer of this situation.
- Air Traffic Control System: A MAS that controls air traffic. There are 4 agents in 2 different types in the desired system. The “Controller” agent responds to aircraft landing or taking off requests from the “Plane” agents. If the runway is available, the "Controller" agent allows landing or take-off. If not, it cancels the landing or take-off. While the “Plane” agents make a request again in case of cancellation, the aircraft will land or take off if allowed
- Paramedic and Ambulance Management System: A MAS that enables ambulance and paramedic services to reach the patients in the fastest way. There are 4 agents in 2 different types in the desired system. The “Coordinator” agent forwards the ambulance calls from 3 different locations to the “Ambulance” agent responsible for the relevant region. The relevant agent deals with this call if the ambulance is available.
- Supply Chain Management System: A MAS that manages a company’s procurement processes. There are 3 different agents in the system to be

implemented. The “Seller” agent responds to orders from customers. If it has a product, it gives the good to the customer. If not, it asks for the "Warehouse" agent that represents the warehouse. If the good exists in the warehouse, the "Warehouse" agent sends it to the "Seller" agent. If the good is not in the warehouse, this time it is requested from the "Supplier" agent. The “supplier” agent sends the goods to the warehouse. The “Warehouse” agent also sends the goods, it receives from the “Supplier” agent to the “Seller” agent. The "Seller" agent also sends the incoming goods to the customer.

- Mine Management System: A MAS that manages the whole mining process of a company. There are 3 different agents in the system to be implemented. The “Seeker” agent is looking for gold ore in the mining area. When it finds a new ore, it notifies the "Supervisor" agent. The “Supervisor” agent reports the status to the “Collector” agent for the collection of the found gold ore. The “Collector” agent collects the ore and notifies the “Supervisor” agent that the collection process is finished.

Models, documents and all other materials for these case studies can be found in the accompanying Mendeley repository [83].

- **Dependent Variables**

  There are 4 different dependent variables here:

  - The first is the amount of **time** spent by each participant during each case study.
  - The second independent variable is the ratio of the **location of faults** detected correctly.
  - The third independent variable is the ratio of the **diagnosis of faults** to be correct.
  - The fourth independent variable is the ratio of the **repair suggestions of faults** to be correct.

5.1.4. Experiment Design

An experiment with one factor and two levels was designed. The relevant factors and their levels have been specified in the previous section. Since the number of potential participants qualified to participate in the experiment is known to be limited, a within-subject design [88] was preferred as previously discussed. Thus, it was ensured that the participants, who could be divided into control and test groups, were not divided into two due to the lack of subjects, and the individual differences in the general performance levels of the participants did not affect the experiment. This is important because the performances of the participants were always different. In an experiment specifically related to debugging, some participants were outperforming others regardless of their status. Thus, all participants had completed the experiment once for each factor level.

Two important disadvantages may arise in such a design: The order effect and the Carry-over effect. To minimize the negative effects of these two conditions on the experiment, a certain time interval was placed between the two applications. Also, during two consecutive sessions, the same participant was not allowed to do the same practice (each case study is called practice) at the same factor level or the other factor level. Here, after a participant has already joined an application, they must wait for at least 2 applications before performing the same application. In other words, the minimum period for a participant to perform the same application at a different factor level is 21 days. As stated earlier, the experiment consisted of ten applications, each lasting 1 hour. The entire evaluation process spanned 65 days, with a 5-day interval between each application. Participants were given a 2-day window to start the relevant application, during which they completed it in a single 1-hour session at their convenience. In this case, the experiment took about 3 months with the preparation phase. The general view of the experimental design is presented in Table 1.

5.1.5. Injected Faults

In particular, a literature search on the classification of error types in MASs was conducted to identify the faults to be injected into the case study models to be used

			Participants					
			Participant_1	Participant _2	Participant _3	Participant _4	Participant _5	Participant _6
Sessions	1	With Tool Support	1	1	1	2	2	2
		Without Tool Support	2	2	2	1	1	1
	2	With Tool Support	3	3	3	4	4	4
		Without Tool Support	5	5	5	6	6	6
	3	With Tool Support	6	6	6	5	5	5
		Without Tool Support	4	4	4	3	3	3
	4	With Tool Support	2	2	2	1	1	1
		Without Tool Support	1	1	1	2	2	2
	5	With Tool Support	7	7	7	8	8	8
		Without Tool Support	8	8	8	7	7	7
	6	With Tool Support	5	5	5	6	6	6
		Without Tool Support	3	3	3	4	4	4
	7	With Tool Support	9	9	9	10	10	10
		Without Tool Support	10	10	10	9	9	9
	8	With Tool Support	4	4	4	3	3	3
		Without Tool Support	6	6	6	5	5	5
	9	With Tool Support	8	8	8	7	7	7
		Without Tool Support	7	7	7	8	8	8
	10	With Tool Support	10	10	10	9	9	9
		Without Tool Support	9	9	9	10	10	10

Table 1: Number of practices performed by each participant at different factor levels in the sessions

in the qualitative evaluation. Although failure types are examined under certain classes in a limited number of studies, failures are usually emphasized at the point of communication (for example, see [60, 89, 90, 91]). In addition, the failure types seen in MAS have been observed in various MAS development studies [92, 86, 93]. In this context, a classification proposal was made by combining the types of failures that can be seen in various MASs. When debugging MASs, the types of failures that can be seen will be different when testing a single agent or MAS. The types of failures seen in a single agent are mostly related to the functionality of the relevant factor. The reviewed literature and sample MASs show that developers often make mistakes on one of the three main characteristics of a single agent. These basic characteristics can be expressed as reactivity, proactivity, and social skills. Here, failures that can be seen in an agent are grouped under two main classes according to their results. These are failures in social skills and cognitive skills (inability to provide reactivity and proactivity properties correctly). On the other hand, in MASs, failure types that concern the entire agent group are also examined under two main cate-

gories. These can be listed as failures that cause communication problems and organizational problems. Communicative problems cause a community of agents to not communicate as expected. Here we examine the issues that might prevent the message flow between agents in the agent community from happening as expected. The second covers the problems that prevent the agent communities within the system or the common goals of the whole system from being successful by showing the expected interactions of the agents.

5.1.6. Instrumentation

The instruments of this experiment are:

- Detailed documentation, and slides of the languages and tools to be used,
- Case study scenarios,
- SAML models in which we inject faults to cause defined failures,
- Personal data confidentiality agreement.

All relevant sources can be found in the accompanying Mendeley repository [83] for the interested readers.

5.1.7. Performing the Experiment

The operational phase of the experiment was carried out in two steps. These are preparation and execution, respectively.

Preparation

During the preparation phase, training videos, which took totaling 2 hours, were prepared for the participants, describing all processes from installation to use of the debugging language and tool. After the relevant videos and the tool to be used in the evaluation process were shared with the participants, an online pre-evaluation study was conducted. This pre-evaluation study aims to present the experimental instruments to the participants in their first forms and thus prevent the negative effects that may arise in the experiments to be carried out. The data obtained from

this session were not evaluated within the scope of the empirical evaluation. After the relevant preliminary study, the experimental instruments were made more useful according to both the data obtained and the feedback received from the participants.

Execution

Participants attended assessment sessions in their home or office settings. Therefore, each participant was assessed at any time during the 2 days determined for the session. If all participants completed the evaluation before the 2-day period expired, the relevant session was finished and the waiting period for the new session was started.

For each evaluation session, the scenarios of 2 case studies, the design models of the related case studies, and the test results of the MAS implemented according to the development processes of the related models were shared with the participants. In other words, each participant completed 2 applications in each evaluation session. It was requested from participants to detect and diagnose the faults that constitute the source of the failures expressed in the test results, whether using the debugging tool on the model or not and repair the relevant model in the light of their findings.

The time each participant can work through a case study was limited to 30 minutes, whether they used the tool. Therefore, each session did not exceed 1 hour.

In addition, signed consent forms were obtained from the participants certifying they voluntarily participated in the evaluation process.

5.1.8. Data Analysis

The accuracy in detecting the locations of the faults, diagnosing, and repairing recommendations was calculated as a percentage from the raw data. Since error numbers are not standard among case examples, ratios are used here. However, the normalization process was not applied due to the upper limit on time. The time variable is recorded in minutes. Data analysis and hypothesis testing in the entire quantitative evaluation process were performed with the IBM SPSS Statistics ver.

24[3].

Descriptive statistics of the data obtained from the experiment are given in Table 2. When the data are analyzed, it is seen that the use of tools at fault localization makes an average of 15% contribution. In addition, though it was observed that while the participants using the tool could detect 33% of the errors in the worst case, they could not detect any errors when the tool was not used. Considering the correct diagnosis of faults and correct fault repair suggestion rates, it is seen that the use of debugging tool contributes 20% on average.

The tool appears to have contributed significantly to the time it took to complete the debugging process. Using the tool shortened the debugging process approximately 10 minutes on the average. In addition, considering the minimum times, some participants completed the case study in 3 minutes among the participants who used the tool, while this time was 13 minutes for those who did not use the tool.

An important point to be noted here is that the effectiveness of tool use is similar according to the values in the average when the 5% cut averages are considered. This indicates that outliers in the data do not have a strong influence on the effectiveness of the tool.

When the 95% confidence interval values for the mean are examined, it is seen that the confidence interval in all independent variables is better in the case of tool support than in the case without tool support. If this procedure could be repeated multiple times with different samples, the varying mean calculation for each sample would be within this range of 95%. Therefore, the absence of any independent variable where the confidence intervals of the two situations intersect leads to the expectation of similar results from multiple replications of the experiment.

Paired sample t-test was planned to determine whether there was a statistically significant difference between dependent variables. However, for the related test to be used, the difference between the paired values of the data must conform to the

[3]International Business Machines Corporation (IBM) SPSS Statistics, https://www.ibm.com/products/spss-statistics

		Localization of faults		Diagnosis of faults		Repair suggestions of faults		Time	
Tool Support		Not exists	Exists	Not exists	Exists	Not exists	Exists	Not exists	Exists
N		60	60	60	60	60	60	60	60
Range		1,00	,67	1,00	,67	1,00	1,00	17,00	27,00
Interquartile range		,50	,50	,67	,50	,94	,50	7,5	12,00
Minimum		,00	,33	,00	,33	,00	,00	13,00	3,00
Maximum		1,00	1,00	1,00	1,00	1,00	1,00	30,00	30,00
Median		0,8750	1	,5	1	,5	,7083	26,891	12,500
Mean		,6958	,8472	,5611	,7778	,5388	,7361	25,5333	14,1000
95% Confidence Interval	,6078	,7868	,4643	,7122	,4384	,6661	24,2437	12,1799	121,799
	,7838	,9076	,6579	,8433	,6393	,8061	26,9229	16,0201	160,201
5% trimmed mean		,7176	,8642	,5679	,7901	,5431	,7500	28,0000	13,8333
Std. deviation		,34061	,23378	,37484	,25382	,38893	,27108	53,2174	74,3264
Variance		,116	,055	,141	,064	,151	,073	28,321	55,244

Table 2: The descriptive statistics of the data collected from the experiment

normal distribution. Table 3 shows the normality tests of the differences between the tool-supported and the not tool-supported measurements obtained from the experiment. According to both Kolmogorov-Smirnov and Shapiro-Wilk tests, it is seen that the data other than time differences do not fit the normal distribution.

	Kolmogorov-Smirnov			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
Difference Between Times	,098	60	,200*	,954	60	,024
Difference Between Fault Localization	,426	60	,000	,604	60	,000
Difference Between Fault Diagnostic	,343	60	,000	,742	60	,000
Difference Between Fault Repair	,356	60	,000	,734	60	,000

Table 3: Normality Tests of Differences

The Wilcoxon Signed Rank Test, which is the non-parametric equivalent of the Dependent Samples t-Test, was used to test the hypotheses (H_0^1, H_0^2 and H_0^3) where the differences between the tool-supported and not tool-supported measurements were not normal according to the normality tests. Although the number of data is sufficient to use the t-Test, the assumptions of the t-test, which is a parametric test, could not be met due to the anomalies in the distribution of the differences in the measurements. However, since the difference between the times fit the normal dis-

tribution, the more reliable parametric Dependent Samples t-Test was used there. Table 4 contains test statistics for the first three hypotheses.

	Difference Between Fault Localization	Difference Between Fault Diagnostic	Difference Between Fault Repair
Z	-3,712	-4,581	-4,488
Asymptotic Sig.	,000	,000	,000

Table 4: Wilcoxon Test results used to test the H_0^1, H_0^2 and H_0^3 hypotheses

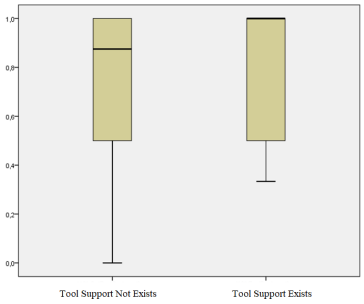
As can be seen from Table 4, H_0^1, H_0^2 and H_0^3 hypotheses were rejected. So, it can be said that the tool use had a significant effect on the localization, diagnosis, and repair suggestion of faults. Table 5 shows the results of the Paired Sample t-Test for the difference between times. It is clear from the table that in this case, the H_0^4 hypothesis was rejected. It is seen that the use of the tool has a significant effect over time.

	Mean	Std. deviation	Std. error mean	95% Confidence Interval		t	df	p
				Lower bound	Upper Bound			
Difference Between Times	11,43333	7,31464	,94432	9,54376	13,32291	12,108	59	,000

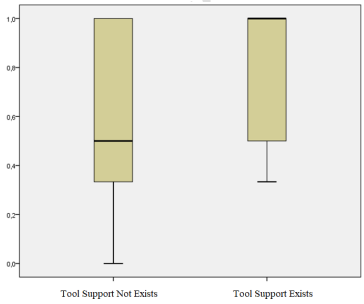
Table 5: Results of the paired sample t-Test used to test the H_0^4 hypothesis

Looking at Figure 20a, it is seen that the participants, who used the tool, found the correct localization of at least 50% of the faults. Although the range of the distribution of half of the data is equal in both cases, the distribution of 25% of the participants who do not use the tool, was extended until the lowest point of the definition range. This situation does not fall below 0.33 in tool use. Though the distance between the midpoints (medians) of the data is very high in Figure 20b, there are also serious differences in the distribution of the data, too. Here, the distribution of 50% of the data of the participants who do not use the tool, is scattered almost over the entire definition range, while 50% of the data of the participants who use the tool are scattered between the maximum point and the midpoint of the definition range. In Figure 20c, although the difference between the midpoints is relatively small, there are significant differences between the distributions. Here, it is seen that the participants who do not use the tool are distributed over almost the entire definition

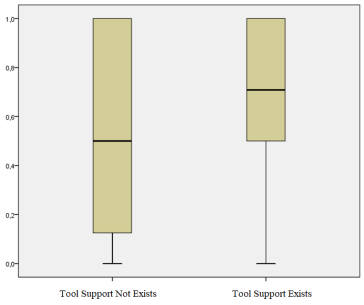
range, while 75% of this distribution is concentrated above 0.5 in the participants who use the tool. In Figure 20d, there are clear differences between the box plots that represent the results of time spent debugging, both in terms of midpoint and distribution.



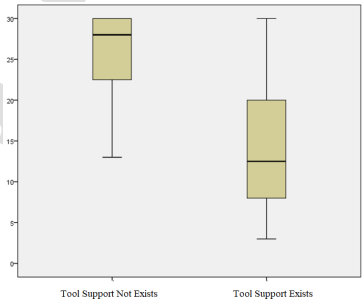
(a) Box plot of the results of correct localization of faults.



(b) Box plot of the results of correct diagnosis of faults.



(c) Box plot of the results of correct repair suggestions of faults.



(d) Box plot of the results of time spent for debugging.

Figure 20: Box plots of the results.

5.2. *Qualitative Evaluation*

Technology Acceptance Model (TAM) [51] with the intention of testing end-user acceptance of a new information system technology was used for this evaluation. TAM tends to explain why individuals adopt a particular information system [94]. In software engineering, TAM is used to control technology adoption in different soft-

ware domains such as Object-Oriented Programming [95], Software Quality Measurements [96], and Meta-Modeling [97]. Researchers use TAM to explain not only the acceptance of an information system or a particular software but also the processes within these systems [98, 99]. In this study, it is aimed to analyze the acceptance of the proposed debugging approach by the users, both in terms of ease of use and usability. Although the quantitative evaluations described in the previous section show that the proposed approach will contribute to the model-driven development of MASs, it may be difficult to be adopted by users if the perception of usability and ease of use is not high. Understanding why MAS developers will accept the proposed approach will also be an important step toward increasing the effectiveness of MAS use in the software industry.

According to TAM, two main factors can influence technology adoption. These two factors are Perceived Ease of Use and Perceived Usability. Perceived Ease of Use refers to the degree to which using technology requires minimal physical or mental effort. On the other hand, Perceived Usability refers to an individual's beliefs about improving job performance while using a particular technology. Actual System Usage is the endpoint where people use technology. Behavioral Intention to Use is a factor that drives people to use technology. Behavioral Intent to Use, or simply Intention to Use, is influenced by Attitude to Use, which is the overall impression of the technology. External Variables such as the maturity, durability, and integrability of the technology can be seen as important factors that directly affect the two main factors in the acceptance of the technology. Considering that the acceptance of such External Variables by users exists in the technology under consideration, users will have the attitude and intention to use this technology. In the technology acceptance model, the assumption is that if a person believes that the use of a particular technology will be uncomplicated and will increase the usability of the final product, they will be more likely to use that technology. Figure 21 shows the TAM first introduced by Davis et al. [51].

For qualitative evaluation, 12 questions were asked under 4 factors based on the technology acceptance model. The aim here is to examine the acceptance status of the tool, which can be seen as the proxy of MASDebugFW. A total of 12 questions

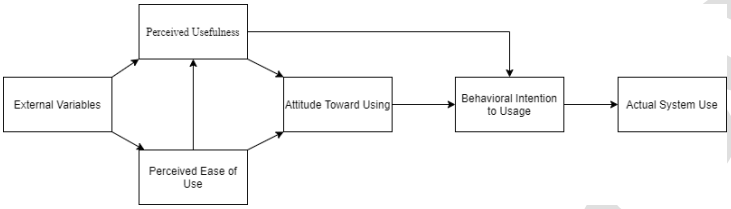


Figure 21: Original TAM Model. [51]

(See the following link for related questions: https://forms.gle/XLuJ6ACcEUo6L34i9), with 3 different questions under each factor, were asked to the participants with a 7-point Likert-type scale after all the case studies were completed. Also, the questionnaire includes an open-ended question regarding the problems and proposals for the acceptance of the tool. The proposed research model and related hypotheses are shown in Figure 22.

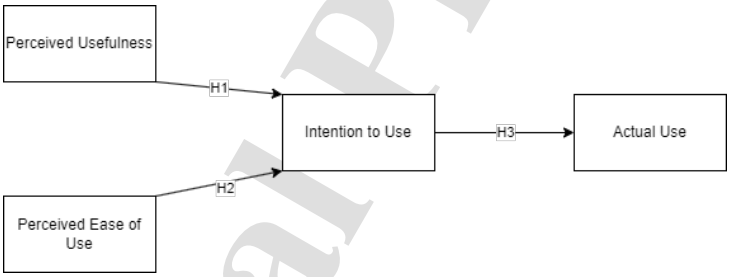


Figure 22: Research model and hypotheses.

According to Figure 22, the followings are hypothesized:

H_1: Perceived Usefulness has a positive effect on the intention to use MASDebugFW.

H_2: Perceived Ease of Use has a positive effect on the intention to use MASDebugFW.

H_3: Intention to Use has a positive effect on MASDebugFW acceptance.

However, before testing the hypotheses in the model, the accuracy of the measurement model must be determined, i.e. the internal consistency of the model should be measured. For this purpose, the Cronbach's Alpha values of each factor should be checked. All Cronbach's Alpha values obtained are between 0.724 and 0.944 which are all above the threshold value of 0.7. Therefore, the internal consistency reliability in terms of Cronbach's Alfa has been verified.

Spearman correlation analysis, which is a nonparametric approach, was carried out to verify the hypothesis tests in the relevant research model. The results are seen in Table 6.

		Actual Use	Intention to Use	Perceived Usefulness	Perceived Ease of Use
Actual Use	Correlation Coefficient	1,000	,940*	,803	,836*
	Sig. (2-tailed)	.	,005	,054	,038
	N	6	6	6	6
Intention to Use	Correlation Coefficient	,940*	1,000	,836*	,971**
	Sig. (2-tailed)	,005	.	,038	,001
	N	6	6	6	6
Perceived Usefulness	Correlation Coefficient	,803	,836*	1,000	,806
	Sig. (2-tailed)	,054	,038	.	,053
	N	6	6	6	6
Perceived Ease of Use	Correlation Coefficient	,836*	,971*	,806	1,000
	Sig. (2-tailed)	,038	,001	,053	.
	N	6	6	6	6

Table 6: Spearman Correlation Analysis

When the results were checked, it was seen that all the hypotheses were confirmed. Moreover, another relationship that was not suggested in the research model could be highlighted. There is a positive relationship between Perceived Usability and Actual Use. In other words, users who perceive the proposed approach as useful declare that they will actually use the system. The absence of a similar relationship between ease of use and actual use indicates that users give more importance to the perception of usability than the perception of ease of use in terms of the use of the proposed approach. In general, all correlation coefficients obtained within the scope of this study were found to be highly acceptable.

According to the answers received from the participants during the qualitative evaluation, the average response scores for each topic are given in Figure 23. When

the results are examined, the score of Perceived Ease of Use is relatively less than all other factors. As can be seen from the open-ended questions asked to the users here, the participants mostly gave feedback on the Perceived Ease of Use. Particularly, users emphasized that the menu design, where it provides access to debugging capabilities, could be better.

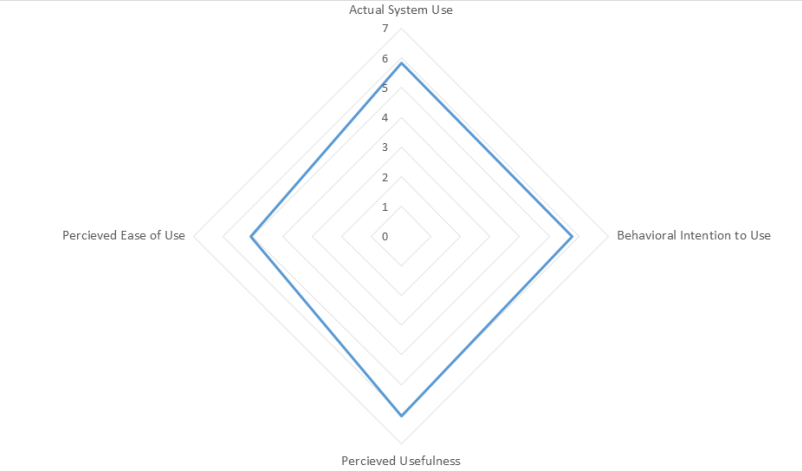


Figure 23: Radar diagram of the results for each factor.

6. Threats to the Validity

This section describes the threats to the validity of the conducted experiment evaluation and its results. The threats can be originated from different validity types including the conclusion, internal, construct, and external validity as described by Wohlin et al. [100].

6.1. Conclusion Validity

This validity is related to the statistical analysis of the results and the composition of the subjects. The main threat to outcome validity in this study is that the data on which the hypothesis will be analyzed and tested is insufficient due to the

lack of participants. Compared to many other computer science and software engineering disciplines, AOSE is a young research area and the number of developers familiar with MAS applications appears to be relatively low compared to other software industries. When the recent studies on using MAS DSMLs [7, 101, 80, 19, 47] are examined, it is seen that the number of participants is similar as in this study. In addition, during the evaluation, the participants were asked not only to fill out the questionnaires in which they could share their usage experiences but also to implement the software development processes covering the development of various MAS in 10 different case studies took a long time. It should also be taken into account that this situation reduces the number of volunteers. On the other hand, according to the usability scale of Neilsen [102], the number of participants in this study is at an acceptable level. However, a multi-case study and within-subject experimental design were used to eliminate this threat, which may arise from the small number of participants. Thus, the participants did not have to be divided into control and test, at the same time, the number of results obtained through multiple case studies was increased. In addition, non-parametric statistical tests were preferred when the assumptions of parametric tests were not met. Thus, statistically significant results were obtained with this small number of participants.

In addition, the relative scarcity of data in the qualitative analysis phase prevents the TAM research model from being handled by using structural equation models like the examples in the literature. Therefore, it was necessary to consider the relevant research model outside of the conventional approaches in the literature. Although the way of handling the research model is quite unusual, it is statistically consistent and significant.

6.2. Internal Validity

Internal validity deals with issues that could cause the measurement processes in the experiments to affect the independent variables without the knowledge of the researcher. If a relationship is observed between treatment and outcome, it should be ensured that it is a causal relationship and not the result of an uncontrolled or unmeasurable factor. Two independent variables are kept constant throughout the

experiment. These are the MAS models and the faults injected into these models. In the experiment, MASDebugFW was evaluated in MAS models where each model contains only two failures, but multiple injected failures due to these failures. Faults are injected into all case studies, taking into account all the types of failures they cause. However, considering each case study, errors may differ depending on the types of failures. Here, this threat has been tried to be reduced by balancing the distribution of failure types in the sum of all case studies. The case studies, each of which is represented by a MAS model, were created to include understandable and simple scenarios as much as possible, depending on the literature and the experience of the researchers. On the other hand, the selection of case studies is not random, which poses a slight threat to the validity of the experiment.

Since the number of potential participants who are qualified to participate in the experiment is known to be limited, the within-subject design was preferred. This preference may threaten the internal validity of the study. To avoid the negative effects of the within-subject design on the experiment, a certain time interval was placed between the two treatments. Also, during two consecutive treatments, the same participant was not allowed to work on the same case at the same factor level or the other factor level. Here, a participant has to wait for at least 2 treatments to be able to do the same treatment at different factor levels after already participating in any treatment.

6.3. Construct Validity

The main threat to construct validity comes from inappropriate or incorrectly defined metrics. For this purpose, we mainly used rational or normalized metrics to analyze the results obtained from the experiment.

The participants did not know which hypotheses were stated and were not involved in any discussion about the advantages and disadvantages of MASDebugFW, so they could not predict what the expected results from the experiment would be. Therefore, this can be regarded as a minor threat to the construct validity.

6.4. External Validity

For the threats to the external validity, generalization of the achieved results should be considered. The experimental environment needs to be more realistic. We believe that this threat was mitigated in this study first by selecting a group of evaluators covering both graduate students and software developers with at least two years of industry experience as well as being graduate students. Second, rather than being trivial examples, the multi-case studies herein consider the development of MAS with varying complexities, i.e. they need to design and implement various MASs having different numbers and types of agents interacting with each other to achieve goals defined for different agent domains.

7. Conclusions

This study aimed to contribute to the development of debugging environments for MAS DSMLs and the development of MAS modeling software tools.

The main motivation for this study is the lack of debugging methods required by the model-driven development processes of MASs. This deficiency is thought to be an important reason for the limited industrial use of MASs. As software complexity increases, it seems that traditional debugging approaches developed for procedural and sequential code are no longer sufficient. Moreover, when the system under consideration is deployed on a parallel or distributed platform, which is quite normal for MASs, non-deterministic synchronization failures can occur frequently. Especially, the synchronous, non-deterministic, and continuous agent behaviors exhibited inside MAS environments make both MAS development and MAS testing/debugging extremely difficult.

MASDebugFW leads to both enriching existing MAS DSMLs with debugging capabilities and creating the new MAS DSML from scratch with debugging capabilities. It enables handling different viewpoints, components and features of a MAS DSML and the relationships between them. In addition, a MAS DSML has been developed by MASDebugFW and the integration of the above-mentioned features has been achieved. Thus, it has become possible to complete the debugging at the

modeling level and to create a MAS software according to the developer's request, before the modeled MASs are implemented. The second contribution of the study is a reference framework that may be adopted by the language engineers who want to create complex visual debugging environments for their MAS DSMLs. The third contribution was demonstrated by creating a model debugging environment for a MAS DSML, and the evaluation of the usability of MASDebugFW inside the debugging process was performed quantitatively and qualitatively.

The quantitative evaluation showed that the use of the tool developed based on MASDebugFW provides a significant reduction in the time spent by the developers for debugging, and the results are very satisfactory in locating, diagnosing, and fixing faults. By analyzing the data collected during the experiments, it was determined that a participant with tool support fixed approximately 75% of the faults in 14 minutes on average. It has been observed that a participant who does not have tool support spends 25.5 minutes on average to fix the same faults and leaves approximately 50% unfixed faults in the models. These results show that the tool reduces the time a participant spends for debugging by approximately 45% and improves the proportion of bugs repaired in models from 50% to 75%, a 25- percentage-point increase in fault resolution rates. Finally, a qualitative evaluation was carried out with a questionnaire administered to the participants. It was seen that this new technology proposed with the obtained data was acceptable by the participants according to the TAM.

In future work, we aim to investigate the methods of incorporating the omniscient debugging [36], model slicing [103] and algorithmic debugging [104], into our framework in addition to the current debugging operations on MAS models, i.e., it will be explored and demonstrated how more advanced debugging approaches can be applied for MAS DSMLs to support e.g. MAS DSMLs enriched with these approaches can speed up or simplify the debugging process.

The framework presented in this study can be revised to generalize debugging for languages with different semantics. It allows system developers to debug the system at the most appropriate level of abstraction using the abstractions that the program co-determines. However, most software systems, such as MASs, distributed

systems, and other complex software architectures, are typically specified using general-purpose programming languages. Debugging these systems presents significant challenges, particularly in concurrent, parallel, and distributed environments, due to the limitations of traditional code debugging tools. In future work, MASDebugFW could be extended to support these types of systems, enabling debugging at higher levels of abstraction and improving fault detection and resolution in such complex environments.

Acknowledgement

This study was funded by the Scientific and Technological Research Council of Türkiye (TÜBİTAK) under grant 115E591.

References

[1] G. Weiss (Ed.), Multiagent Systems, Second Edition, MIT press, 2016.

[2] M. Wooldridge, An introduction to multiagent systems, John wiley & sons, 2009.

[3] N. Howden, R. Rönnquist, A. Hodgson, A. Lucas, Jack intelligent agents-summary of an agent infrastructure, in: 5th International conference on autonomous agents, Vol. 6, 2001, pp. 1–27.

[4] F. Bellifemine, A. Poggi, G. Rimassa, Developing multi-agent systems with a fipa-compliant agent framework, Software: Practice and Experience 31 (2) (2001) 103–128.

[5] A. Pokahr, L. Braubach, A. Walczak, Jadex - engineering goal-oriented agents, Developing Multi-Agent Systems with JADE.

[6] R. H. Bordini, J. F. Hbner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak using Jason, Wiley Series in Agent Technology, John Wiley & Sons, Ltd, Chichester, UK, 2007. doi:10.1002/9780470061848.

[7] M. Challenger, G. Kardas, B. Tekinerdogan, A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems, Software Quality Journal 24 (3) (2016) 755–795.

[8] V. Mascardi, D. Weyns, A. Ricci, C. B. Earle, A. Casals, M. Challenger, A. Chopra, A. Ciortea, L. A. Dennis, Á. F. Díaz, et al., Engineering multi-agent systems: State of affairs and the road ahead, ACM SIGSOFT Software Engineering Notes 44 (1) (2019) 18–28.

[9] O. Shehory, A. Sturm, Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks, Springer-Verlag Berlin Heidelberg, 2014.

[10] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Synthesis lectures on software engineering 3 (1) (2017) 1–207.

[11] B. Lelandais, M.-P. Oudot, B. Combemale, Applying model-driven engineering to high-performance computing: Experience report, lessons learned, and remaining challenges, Journal of Computer Languages 55 (2019) 100919.

[12] A. Bucchiarone, J. Cabot, R. F. Paige, A. Pierantonio, Grand challenges in model-driven engineering: an analysis of the state of the research, Software and Systems Modeling 19 (1) (2020) 5–13.

[13] E. de Araújo Silva, E. Valentin, J. R. H. Carvalho, R. da Silva Barreto, A survey of model driven engineering in robotics, Journal of Computer Languages 62 (2021) 101021.

[14] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, M. Wimmer, Low-code development and model-driven engineering: Two sides of the same coin?, Software and Systems Modeling 21 (2) (2022) 437–446.

[15] G. Kardas, Model-driven development of multiagent systems: a survey and evaluation, The Knowledge Engineering Review 28 (04) (2013) 479–503.

[16] G. Kardas, J. J. Gomez-Sanz, Special issue on model-driven engineering of multi-agent systems in theory and practice., Comput. Lang. Syst. Struct. 50 (2017) 140–141.

[17] O. F. Alaca, B. T. Tezel, M. Challenger, M. Goulão, V. Amaral, G. Kardas, Agentdsm-eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems, Computer Standards & Interfaces 76 (2021) 103513.

[18] F. Ihirwe, D. Di Ruscio, S. Gianfranceschi, A. Pierantonio, Chessiot: A model-driven approach for engineering multi-layered iot systems, Journal of Computer Languages 78 (2024) 101254.

[19] G. Kardas, B. T. Tezel, M. Challenger, Domain-specific modelling language for belief–desire–intention software agents, IET Software 12 (4) (2018) 356–364.

[20] B. T. Tezel, M. Challenger, G. Kardas, Dsml4bdi: A modeling tool for bdi agent development, in: 12th turkish national software engineering symposium (uyms 2018), 2018, pp. 1–8.

[21] A. Siabdelhadi, A. Chadli, H. Cherroun, A. Ouared, H. Sahraoui, Motrans-bdi: Leveraging the beliefs-desires-intentions agent architecture for collaborative model transformation by example, Journal of Computer Languages 74 (2023) 101174.

[22] G. Kardas, F. Ciccozzi, L. Iovino, Introduction to the special issue on methods, tools and languages for model-driven engineering and low-code development, Journal of Computer Languages 74.

[23] G. Beydoun, G. Low, B. Henderson-Sellers, H. Mouratidis, J. Gomez-Sanz, J. Pavon, C. Gonzalez-Perez, FAML: A Generic Metamodel for MAS Development, IEEE Transactions on Software Engineering 35 (6) (2009) 841–863. doi:10.1109/TSE.2009.34.

[24] C. Hahn, C. Madrigal-Mora, K. Fischer, A platform-independent metamodel for multiagent systems, Autonomous Agents and Multi-Agent Systems 18 (2) (2009) 239–266.

[25] B. T. Tezel, M. Challenger, G. Kardas, A metamodel for Jason BDI agents, in: 5th Symposium on Languages, Applications and Technologies (SLATE'16), Vol. 51, 2016, pp. 8:1—-8:9.

[26] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, T. Kosar, On the use of a domain-specific modeling language in the development of multiagent systems, Engineering Applications of Artificial Intelligence 28 (2014) 111–141.

[27] E. J. T. Gonçalves, M. I. Cortés, G. A. L. Campos, Y. S. Lopes, E. S. Freire, V. T. da Silva, K. S. F. de Oliveira, M. A. de Oliveira, Mas-ml 2.0: Supporting the modelling of multi-agent systems with different agent architectures, Journal of Systems and Software 108 (2015) 77–109.

[28] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Agent-oriented model-driven development for JADE with the JADEL programming language, Computer Languages, Systems & Structures 50 (2017) 142–158.

[29] D. Sredojević, M. Vidaković, M. Ivanović, Alas: agent-oriented domain-specific language for the development of intelligent distributed non-axiomatic reasoning agents, Enterprise Information Systems 12 (8-9) (2018) 1058–1082.

[30] S. HoseinDoost, T. Adamzadeh, B. Zamani, A. Fatemi, A model-driven framework for developing multi-agent systems in emergency response environments, Software & Systems Modeling (2017) 1–28doi:10.1007/s10270-017-0627-4.

[31] H. Wu, J. Gray, M. Mernik, Grammar-driven generation of domain-specific language debuggers, Software: Practice and Experience 38 (10) (2008) 1073–1103.

[32] A. Blunk, J. Fischer, D. A. Sadilek, Modelling a Debugger for an Imperative Voice Control Language, in: SDL 2009: Design for Motes and Mobiles. SDL 2009. Lecture Notes in Computer Science, Vol. 5719, Springer, 2009, pp. 149–164.

[33] R. Mannadiar, H. Vangheluwe, Debugging in Domain-Specific Modelling, in: Lecture Notes in Computer Science, Vol. 6563, Springer, 2011, pp. 276–285.

[34] T. Kosar, M. Mernik, J. Gray, T. Kos, Debugging measurement systems using a domain-specific modeling language, Computers in Industry 65 (4) (2014) 622–635. doi:10.1016/j.compind.2014.01.013.

[35] A. Chiş, M. Denker, T. Gîrba, O. Nierstrasz, Practical domain-specific debuggers using the Moldable Debugger framework, Computer Languages, Systems & Structures 44 (Part A) (2016) 89–113.

[36] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, B. Baudry, Omniscient debugging for executable dsls, Journal of Systems and Software 137 (2018) 261–288.

[37] B. Liu, S. Nejati, L. C. Briand, et al., Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy, Empirical Software Engineering 24 (1) (2019) 444–490.

[38] S. Van Mierlo, A multi-paradigm modelling approach for engineering model debugging environments, Ph.D. thesis, University of Antwerp (2018).

[39] B. T. Tezel, G. Kardas, Towards providing debugging in the domain-specific modeling languages for software agents, in: Proceedings of the Second International Workshop on Debugging in Model-Driven Engineering (MDEbug 2018) co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), 2018, pp. 1–3.

[40] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2009.

[41] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, A. Zeller, Where is the bug and how is it fixed? an experiment with practitioners, in: Proceedings of the 2017 11th joint meeting on foundations of software engineering, 2017, pp. 117–128.

[42] E. Soremekun, L. Kirschner, M. Böhme, A. Zeller, Locating faults with program slicing: an empirical analysis, Empirical Software Engineering 26 (3) (2021) 1–45.

[43] R. N. Charette, Why software fails [software failure], IEEE spectrum 42 (9) (2005) 42–49.

[44] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, B. Combemale, Execution framework of the gemoc studio (tool demo), in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, 2016, pp. 84–89.

[45] S. Van Mierlo, E. Bousse, H. Vangheluwe, M. Wimmer, C. Verbrugge, M. Gogolla, M. Tichy, A. Blouin, Report on the 1st International Workshop on Debugging in Model-Driven Engineering (MDEbug17), in: Proceedings of the 1st International Workshop on Debugging in Model-Driven Engineering, 2017, pp. 441–446.

[46] S. Van Mierlo, H. Vangheluwe, Debugging Non-determinism: a Petrinets Modelling, Analysis, and Debugging Tool, in: Proceedings of the 1st International Workshop on Debugging in Model-Driven Engineering (MDEbug 2017), 2017, pp. 460–462.

[47] T. Z. Asici, B. T. Tezel, G. Kardas, On the use of the analytic hierarchy process in the evaluation of domain-specific modeling languages for multi-agent systems, Journal of Computer Languages 62 (2021) 101020.

[48] T. Ahlbrecht, An algorithmic debugging approach for belief-desire-intention agents, Annals of Mathematics and Artificial Intelligence (2023) 1–18.

[49] F. Santos, I. Nunes, A. L. Bazzan, Quantitatively assessing the benefits of model-driven development in agent-based modeling and simulation, Simulation Modelling Practice and Theory 104 (2020) 102126.

[50] B. T. Tezel, G. Kardas, A Conceptual Generic Framework to Debugging in the Domain-Specific Modeling Languages for Multi-Agent Systems, in: R. Rodrigues, J. Janousek, L. Ferreira, L. Coheur, F. Batista, H. G. Oliveira (Eds.), 8th Symposium on Languages, Applications and Technologies (SLATE 2019), Vol. 74 of OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 8:1–8:13. doi:10.4230/OASIcs.SLATE.2019.8.

[51] F. D. Davis, Perceived usefulness, perceived ease of use, and user acceptance of information technology, MIS quarterly (1989) 319–340.

[52] J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 485–495.

[53] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, S. P. Reiss, The design and implementation of a dataflow language for scriptable debugging, Automated Software Engineering 14 (1) (2007) 59–86.

[54] A. Zeller, D. Lütkehaus, Ddd—a free graphical front-end for unix debuggers, ACM Sigplan Notices 31 (1) (1996) 22–27.

[55] J. H. Cross, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, L. N. Montgomery, Robust generation of dynamic data structure visualizations with multiple interaction approaches, ACM Transactions on Computing Education (TOCE) 9 (2) (2009) 1–32.

[56] A. Risberg Alaküla, G. Hedin, N. Fors, A. Pop, Property probes: Source code based exploration of program analysis results, in: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, 2022, pp. 148–160.

[57] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.

[58] J. Silva, A survey on algorithmic debugging strategies, Advances in engineering software 42 (11) (2011) 976–991.

[59] C. Artho, Iterative delta debugging, International Journal on Software Tools for Technology Transfer 13 (3) (2011) 223–246.

[60] M. H. Van Liedekerke, N. M. Avouris, Debugging multi-agent systems, Information and Software Technology 37 (2) (1995) 103–112. doi:10.1016/0950-5849(95)93487-Y.

[61] H. S. Nwana, D. T. Ndumu, L. C. Lee, J. C. Collis, Zeus: A toolkit for building distributed multiagent systems, Applied Artificial Intelligence 13 (1-2) (1999) 129–185. doi:10.1080/088395199117513.

[62] M. Dastani, J. Brandsema, A. Dubel, J.-J. C. Meyer, Debugging bdi-based multi-agent programs, in: International workshop on programming multi-agent systems, Springer, 2009, pp. 151–169.

[63] S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2008. doi:10.1002/9780470249260.

[64] L. Safa, The Making of User-Interface Designer: a Proprietary DSM Tool (2007).

[65] M. G. Van Den Brand, B. Cornelissen, P. A. Olivier, J. J. Vinju, Tide: A generic debugging framework—tool demonstration—, Electronic Notes in Theoretical Computer Science 141 (4) (2005) 161–165.

[66] R. T. Lindeman, L. C. L. Kats, E. Visser, Declaratively Defining Domain-Specific Language Debuggers, in: International Conference on Generative Programming and Component Engineering (GPCE), 2012, pp. 127–136. doi:10.1145/2189751.2047885.

[67] E. Bousse, J. Corley, B. Combemale, J. Gray, B. Baudry, Supporting efficient and advanced omniscient debugging for xDSMLs, in: Proceedings of the 2015 ACM SIGPLAN Int. Conf. Software Language Engineering (SLE 2015), 2015, pp. 137–148.

[68] F. Khorram, E. Bousse, A. Garmendia, J.-M. Mottu, G. Sunyé, M. Wimmer, A language-parametric test coverage framework for executable domain-specific languages, Journal of Systems and Software 211 (2024) 111977.

[69] M. Pasquier, C. Teodorov, F. Jouault, M. Brun, L. Le Roux, L. Lagadec, Temporal breakpoints for multiverse debugging, in: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, 2023, pp. 125–137.

[70] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. F. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Transactions on Software Engineering 28 (6) (2002) 595–606.

[71] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Črepinšek, C. D. Da, R. P. Henriques, Comparing general-purpose and domain-specific languages: An empirical study, Computer Science and Information Systems 7 (2) (2010) 247–264.

[72] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments, Empirical software engineering 17 (2012) 276–304.

[73] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al., Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems & Structures 44 (2015) 24–47.

[74] T. Kosar, Ž. Kovačević, M. Mernik, B. Slivnik, The impact of code bloat on ge-

netic program comprehension: Replication of a controlled experiment on semantic inference, Mathematics 11 (17) (2023) 3744.

[75] J. Corley, B. P. Eddy, E. Syriani, J. Gray, Efficient and scalable omniscient debugging for model transformations, Software Quality Journal 25 (1).

[76] R. G. Smith, The contract net protocol: High-level communication and control in a distributed problem solver, IEEE Transactions on computers 12 (1980) 1104–1113.

[77] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, M. Wimmer, Promobox: a framework for generating domain-specific property languages, in: International Conference on Software Language Engineering, Springer, 2014, pp. 1–20.

[78] Á. Hegedüs, I. Ráth, D. Varró, Replaying execution trace models for dynamic modeling languages, Periodica Polytechnica Electrical Engineering and Computer Science 56 (3) (2012) 71–82.

[79] C. Hahn, A Domain Specific Modeling Language for Multiagent Systems, in: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1, International Foundation for Autonomous Agents and Multiagent Systems, Estoril, Portugal, 2008, pp. 233–240, aAMAS '08.

[80] G. Kardas, E. Bircan, M. Challenger, Supporting the platform extensibility for the model-driven development of agent systems by the interoperability between domain-specific modeling languages of multi-agent systems, Comput Sci Inf Syst 14 (3) (2017) 875–912.

[81] S. João, A. Barisic, V. Amaral, M. Goulao, B. Tezel, T. Alaca, F. Omer, M. Challenger, G. Kardas, Comparing the developer experience with two multi-agents systems dsls: Sea_ml++ and dsml4mas-study design., in: Human Factors in Modelling (HUFAMO) workshop in 21th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2018, pp. 770–777.

[82] T. Miranda, M. Challenger, B. T. Tezel, O. F. Alaca, A. Barišić, V. Amaral, M. Goulão, G. Kardas, Improving the usability of a mas dsml, in: International workshop on engineering multi-agent systems, Springer, 2018, pp. 55–75.

[83] B. T. Tezel, G. Kardas, Dataset for: Debugging in the domain-specific modeling languages for software agents", https://data.mendeley.com/datasets/tvy82j656w/4 (2024). doi:10.17632/tvy82j656w.4.

[84] R. H. Bordini, J. F. Hübner, R. Vieira, Jason and the golden fleece of agent-oriented programming, in: Multi-agent programming, Springer, 2005, pp. 3–37.

[85] J. Collenette, B. Logan, Multi-agent control of industrial robot vacuum cleaners, in: International Workshop on Engineering Multi-Agent Systems, Springer, 2020, pp. 87–99.

[86] M. Challenger, B. T. Tezel, O. Alaca, B. Tekinerdogan, G. Kardas, Development of Semantic Web-Enabled BDI Multi-Agent Systems Using SEA_ML: An Electronic Bartering Case Study, Applied Sciences 8 (5) (2018) 688. doi:10.3390/app8050688.

[87] J. H. Kim, I. Choi, Choosing the level of significance: a decision-theoretic approach, Abacus 57 (1) (2021) 27–71.

[88] D. C. Montgomery, Design and Analysis of Experiments, 10th Edition, John wiley & sons, 2019.

[89] D. Poutakidis, L. Padgham, M. Winikoff, Debugging multi-agent systems using design artifacts: The case of interaction protocols, in: Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, ACM, 2002, pp. 960–967.

[90] C. Rouff, A test agent for testing agents and their communities, in: Proceedings, IEEE Aerospace Conference, Vol. 5, IEEE, 2002, pp. 5–2638.

[91] M.-P. Huget, Y. Demazeau, Evaluating multiagent systems: a record/replay approach, in: Proceedings. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004.(IAT 2004)., IEEE, 2004, pp. 536–539.

[92] Y. E. Cakmaz, O. F. Alaca, C. Durmaz, B. Akdal, B. Tezel, M. Challenger, G. Kardas, Engineering a bdi agent-based semantic e-barter system, in: 2017 International Conference on Computer Science and Engineering (UBMK), IEEE, 2017, pp. 1072–1077.

[93] M. Challenger, B. T. Tezel, V. Amaral, M. Goulao, G. Kardas, Agent-based cyber-physical system development with sea_ml++, in: Multi-Paradigm Modelling Approaches for Cyber-Physical Systems, Elsevier, 2021, pp. 195–219.

[94] M. Turner, B. Kitchenham, P. Brereton, S. Charters, D. Budgen, Does the technology acceptance model predict actual use? a systematic literature review, Information and software technology 52 (5) (2010) 463–479.

[95] B. C. Hardgrave, R. A. Johnson, Toward an information systems development acceptance model: the case of object-oriented systems development, IEEE Transactions on Engineering Management 50 (3) (2003) 322–336.

[96] L. G. Wallace, S. D. Sheetz, The adoption of software measures: A technology acceptance model (tam) perspective, Information & Management 51 (2) (2014) 249–259.

[97] V. Mezhuyev, M. Al-Emran, M. Fatehah, N. C. Hong, Factors affecting the metamodelling acceptance: a case study from software development companies in malaysia, IEEE Access 6 (2018) 49476–49485.

[98] M. Umarji, C. Seaman, Predicting acceptance of software process improvement, ACM SIGSOFT Software Engineering Notes 30 (4) (2005) 1–6.

[99] V. Mezhuyev, M. Al-Emran, M. A. Ismail, L. Benedicenti, D. A. Chandran, The acceptance of search-based software engineering techniques: An empirical evaluation using the technology acceptance model, IEEE Access 7 (2019) 101073–101085.

[100] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[101] J. Faccin, I. Nunes, A tool-supported development method for improved bdi plan selection, Engineering Applications of Artificial Intelligence 62 (2017) 195–213.

[102] J. Nielsen, How many test users in a usability study, https://www.nngroup.com/articles/how-many-test-users/ (2012).

[103] J. Silva, A vocabulary of program slicing-based techniques, ACM computing surveys (CSUR) 44 (3) (2012) 12.

[104] T. Ahlbrecht, An algorithmic debugging approach for bdi agents, in: International Workshop on Engineering Multi-Agent Systems, 2022, pp. 1–12.

Information to appear on the first page of your article

Title: Debugging in the Domain-Specific Modeling Languages for Multi-Agent Systems

Authors (including degrees): Baris Tekin Tezel (PhD), Geylani Kardas (PhD)

Affiliations for all authors (Department (spelt in full), School, Hospital, or Organization, City, State/Province, Country):

Department of Computer Science, Dokuz Eylul University, Buca, Izmir, 35390, Türkiye,

International Computer Institute, Ege University, Bornova, Izmir, 35100, Türkiye,

Conflict of Interests:

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Updated November 2021

Corresponding author address (Street and full postal address, without the institution or department):

Department of Computer Science, Dokuz Eylul University, Buca, Izmir, 35390, Türkiye,

Correspondence telephone (please follow format +1 XXX XXX XXXX for US Nos):

+9 0553 956 00 94

Correspondence email:

baris.tezel@deu.edu.tr

Information to appear on the end pages of your article before the references

Acknowledgment and funding sources:

Updated November 2021

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Baris Tekin Tezel reports was provided by Dokuz Eylül University. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.