A Conceptual Generic Framework to Debugging in the Domain-Specific Modeling Languages for Multi-Agent Systems

Baris Tekin Tezel
Department of Computer Science, Dokuz Eylul University, Izmir, Turkey
International Computer Institute, Ege University, Izmir, Turkey
baris.tezel@deu.edu.tr

Geylani Kardas
International Computer Institute, Ege University, Izmir, Turkey
geylani.kardas@ege.edu.tr

Abstract

Despite the existence of many agent programming environments and platforms, the developers may still encounter difficulties on implementing Multi-agent Systems (MASs) due to the complexity of agent features and agent interactions inside the MAS organizations. Working in a higher abstraction layer and modeling agent components within a model-driven engineering (MDE) process before going into depths of MAS implementation may facilitate MAS development. Perhaps the most popular way of applying MDE for MAS is based on creating Domain-specific Modeling Languages (DSMLs) with including appropriate integrated development environments (IDEs) in which both modeling and code generation for system-to-be-developed can be performed properly. Although IDEs of these MAS DSMLs provide some sort of checks on modeled systems according to the related DSML's syntax and semantics descriptions, currently they do not have a built-in support for debugging these MAS models. That deficiency causes the agent developers not to be sure on the correctness of the prepared MAS model at the design phase. To help filling this gap, we introduce a conceptual generic debugging framework supporting the design of agent components inside the modeling environments of MAS DSMLs. The debugging framework is composed of 4 different metamodels and a simulator. Use of the proposed framework starts with modeling a MAS using a design language and transforming design model instances to a run-time model. According to the framework, the run-time model is simulated on a built-in simulator for debugging. The framework also provides a control mechanism for the simulation in the form of a simulation environment model.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Software testing and debugging; Computing methodologies → Multi-agent systems; Computing methodologies → Modeling and simulation

Keywords and phrases debugging, domain-specific modeling languages, multi-agent systems, simulation

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.8

Funding *Baris Tekin Tezel*: The first author would like to thank TUBITAK-BIDEB for financial support during his PhD studies.

1 Introduction

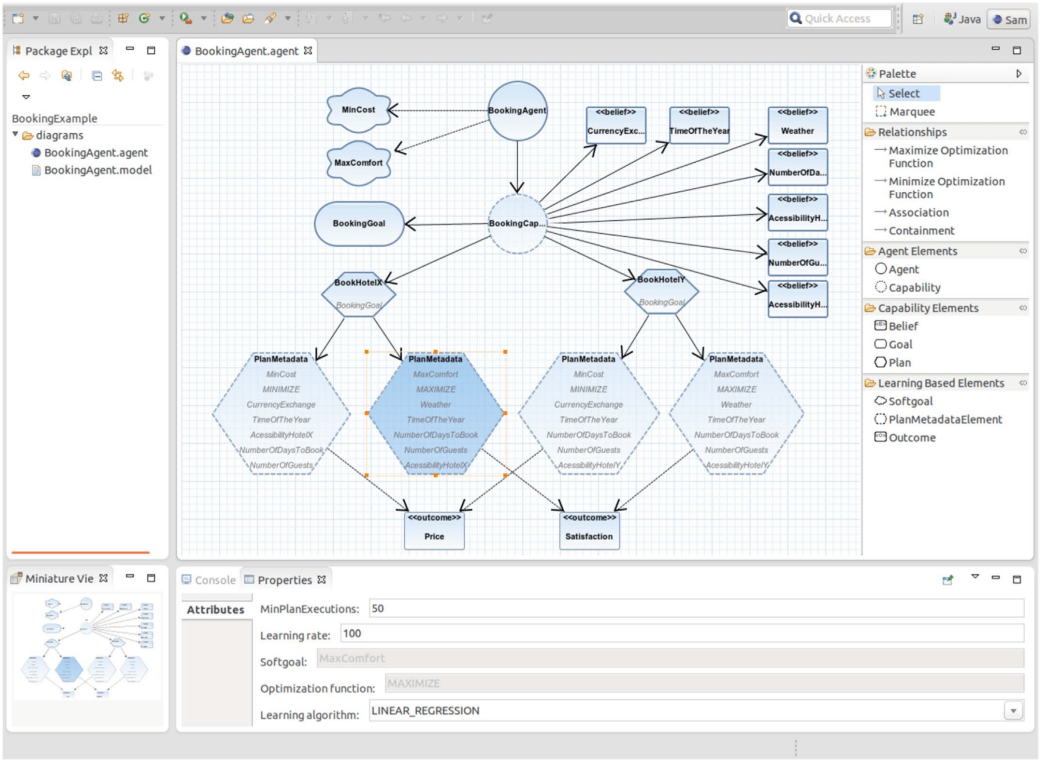
Agents are mostly defined as the computer systems which are capable of autonomous actions inside an environment in order to achieve its design objectives [44]. They may behave as simple as just responding environmental changes or their behavior model can be too complex and the agents may need to proactively act to anticipate future goals. These agents interact with other agents and hence constitute Multi-agent Systems (MASs). Despite the existence of many agent programming environments and platforms such as CLAIM [35], GOAL [19],

JADE [1], JACK [21], MOISE+ [22], the developers may still encounter difficulties on implementing MAS due to above mentioned features of agents and agent interactions inside the MAS organizations [8].

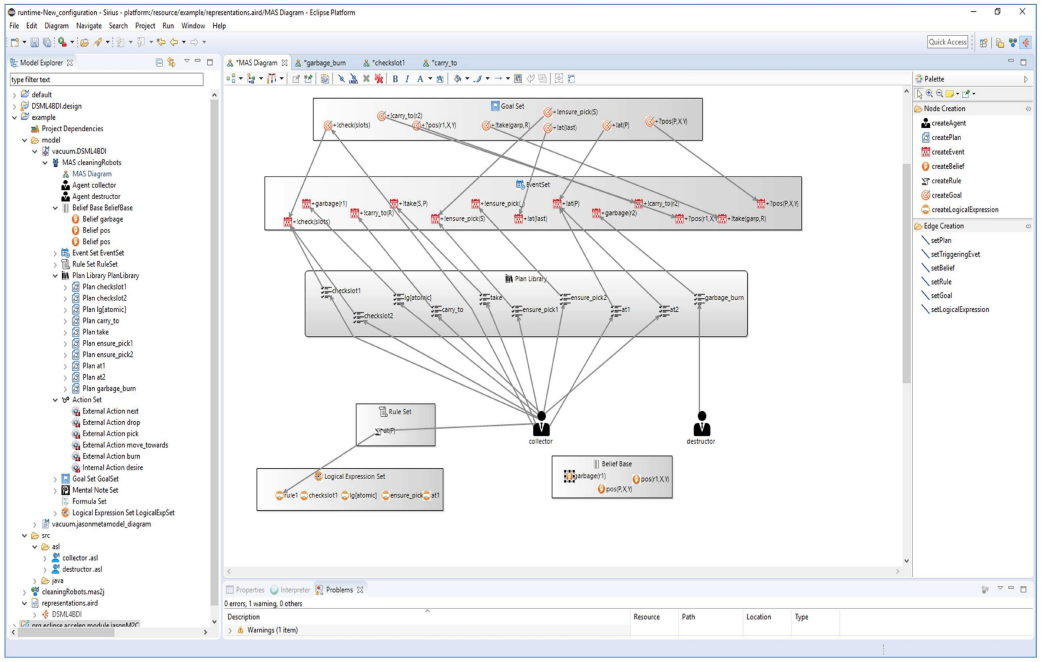
Working in a higher abstraction layer and modeling agent components within a model-driven engineering (MDE) process before going into depths of MAS implementation may help building MAS [23]. Various agent metamodels (e.g. [3, 16, 40]) are defined by the Agent-oriented Software Engineering (AOSE) [36] community for modeling agents, their plans, beliefs, goals and agent interactions inside MAS organizations. Perhaps the most popular way of applying MDE for MASs is based on creating Domain-specific Modeling Languages (DSMLs) with including appropriate integrated development environments (IDEs) in which both modeling and code generation for system-to-be-developed can be performed properly [25]. Proposed MAS DSMLs (e.g. [15, 7, 14, 2, 12, 26] have abstract syntaxes based on the above referred agent metamodels and they usually support modeling both the static and the dynamic aspects of agent software from different MAS viewpoints including agent internal behaviour model, interaction with other agents, use of other environment entities, etc. To give some flavor of current modeling environments of these DSMLs, screenshots taken from the IDEs of two relatively new MAS DSMLs, Sam [12] and DSML4BDI [26], are given in Figure 1.

As can be seen from the screenshots, these IDEs provide a modeling area with a palette (at the right side) with including the graphical representations of MAS components. Developers may drag and drop these components and create the agent models pertaining to the specific MAS viewpoints. Upon completion of modeling, a series of model-to-model and/or model-to-text transformations are applied on the models to generate the executables (e.g. agent codes) required for the exact implementation of the MAS. Although IDEs of these MAS DSMLs provide some sort of checks on modeled systems according to the related DSML's syntax and (mostly static) semantics descriptions, currently they do not have a built-in support for debugging these MAS models [41]. That deficiency causes the agent developers not to be sure on the correctness of the prepared MAS model at the design phase. To help filling this gap, we introduce how a conceptual generic debugging framework can be derived for MAS DSMLs in this paper. The framework is conceptual since its components are defined but not fully implemented yet.

Understanding of software execution behavior has always been very hard task. Debuggers help developers to understand execution behavior of software by accessing directly executed programs [17, 46, 13]. Besides, debugging activities have a broader meaning in the domain-specific modeling (DSM) since a model developer usually needs to debug models at the model level, not at the code level [29]. In our previous study [41], we investigated the ways of creating debugging mechanisms for MAS DSMLs and introduced two possible approaches. The first approach is based on the construction of a mapping between MAS model entities and the generated codes while the second one considers the metamodel-based description of the operational semantics of executing agents. A brief evaluation of these approaches showed that the application of the first approach can be easier since it benefits from using already existing general-programming language (GPL) debuggers. However, use of MAS DSMLs do not only produce executable codes; other artifacts (e.g. agent configuration files, service descriptions) also need debugging. Moreover, generated codes mostly do not contain complete behavioral logic required for the exact implementation of agents. These can make the application of the first approach inefficient. The second approach, utilizing the metamodel-based description of agent operational semantics, seems promising since it is free from underlying GPL structures. However, it is more difficult to apply because



(a) A screenshot from the IDE of Sam [12].



(b) A screenshot from the IDE of DSML4BDI [26].

Figure 1 Screenshots from the IDEs of two different MAS DSMLs.

it needs addition of parts describing the run-time state of MAS model into the language metamodel and writing the corresponding model to model (M2M) transformation rules. Based on the findings of this previous work, in this paper, we present how the current design structures of the existing MAS DSMLs can be enriched with additional run-time, simulation and visualization languages to construct a conceptual debugging framework. That framework is generic to provide the design of both agent internals and communications and it may pave the way for implementing MAS DSMLs with built-in debuggers. Thus, it is possible to complete the debugging phase at the modeling level before the code generation which leads to creating a MAS model conforming to the specifications at the beginning.

The rest of the paper is organized as follows: In section 2, we introduce the overview of the proposed conceptual generic debugging framework. Debugging operations which can be managed inside the framework is given in section 3. Supporting model simulator is described in section 4. A brief review of the related work within the context of debugging on DSMLs, software agents and MAS is given in section 5. Finally, section 6 concludes the paper.

2 The conceptual generic debugging framework for MAS DSMLs

In this study, our goal is to provide a generic framework for debugging during MAS modeling inside IDEs of MAS DSMLs.The framework may enable providing debugging abilities to existing MAS DSMLs as well as guiding DSML developers for the insertion of model debugging features during design and implementation of new MAS DSMLs. The debugging framework is generic enough to be reusable and easily adaptable for various aspects and viewpoints of MAS DSMLs. For instance, it would be possible to model and validate the execution of agents plans, consistency of beliefsets or construction of the agent communications according to the well-defined agent protocols, such as Contract Net [39]. Overview of the proposed framework is shown in Fig 2. The framework is the composition of 4 different metamodels and a simulator enabling the MAS operational semantics. This formalization of the framework is adopted from [43] which provides a structured approach for turning modelling and simulation environments into interactive debugging environments. Furthermore, the overall structure of the proposed framework is inspired from the sub-languages descriptions discussed in the ProMoBox [30] system and the dynamic modeling language composition defined in [18].

Inside the framework, a MAS design language metamodel, MM_{MASDL}, defines the structure of design language which is used for modelling the static structure of a MAS. Existing MAS DSMLs already have this kind of viewpoint(s) usually merged with their behaviour representations. So, if the debugging ability is desired for any existing MAS DSML, probably the metamodel of the language is needed to be refined to achieve the related static structure. For example, MAS DSMLs such as DSML4BDI [26], DSML4MAS [15] and SEA_ML [7] can be considered as the design languages since they does not have any behavioral diagram. However, if a MAS DSML has both behavior and structural viewpoints, while structural viewpoints considered as the design language, behaviour viewpoints or part of these viewpoints can be used in the run time language which is explained in the next. An instance of this metamodel is called the MAS design model (M_{MASDL}). The general structure of the system could be modeled with these instance models. They are designed and created by the users. The concepts such as agents, roles, capabilities, plans or events and the relationships between these concepts would be modelled in this model.

Next, a MAS run-time language metamodel MM_{MASRL} supports modeling in the framework to represent the run-time states of the above discussed design models. For example; meta-entities such as currentPlan, nextPlan, currentAction, nextAction, and currentBelief

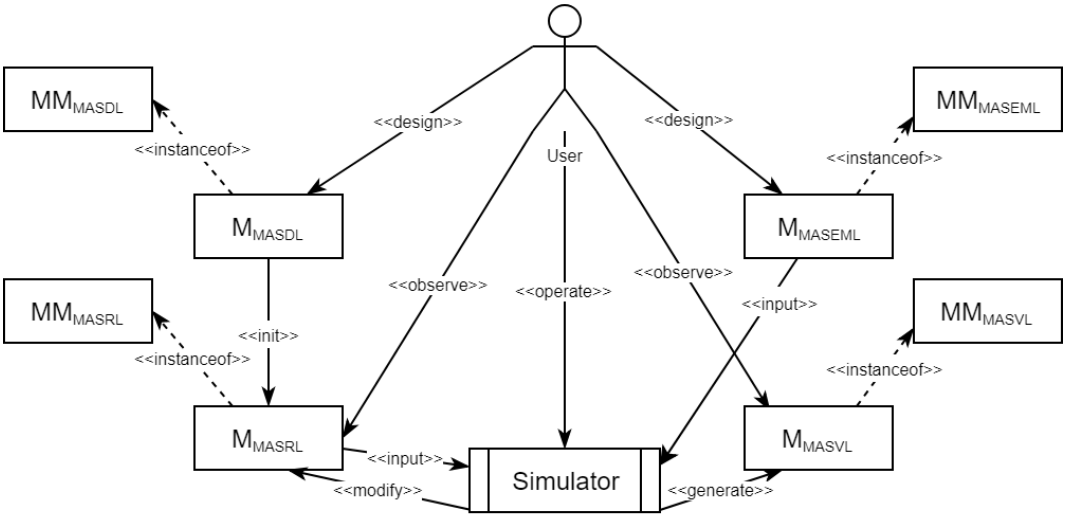


Figure 2 Overview of the conceptual generic debugging framework for MAS DSMLs.

are possibly included in this metamodel. Run-time model (M_{MASRL}) of the system is the instance of MM_{MASRL}. It represents the system run-time states and it is originated from the design model. In other words, these models express the snapshots of the MAS states at run-time.

Third, a metamodel for MAS simulation environment language (MM_{MASEML}) lets modeling of the simulation environment in which the MAS model is simulated. It can also be said that MAS simulation scenarios are modeled as being the instance models (M_{MASEML}) of MM_{MASEML}. Actually, this model represents the behavior of the simulation environment. Situations such as the results of the actions of an agent in the simulation environment, the events that will arise in the environment, communication conditions between agents, resource accesses and resource utilization cases should all be modelled with this simulation environment model. It is worth indicating that it expresses the conditions under which the MAS will be tested.

Finally, a MAS visualization language metamodel, MM_{MASVL}, allows to create customized models which graphically shows the related parts of a MAS in a way that increasing the comprehension of the MAS models by the developers. These models (M_{MASVL}) are requested by the developer. For example, if a developer is just interested in the communication between the specified agents, then the simulator generates the visualization of a complete simulation trace that is used to present the communication step-by-step. Moreover, the simulation trace is generated for not only MAS views but also inner views of an agent such as plan execution.

This conceptual framework supports debugging on a MAS, which is designed with the help of MAS DSML models created by the agent developers before the implementation phase. Thus, it is aimed to minimize the bugs on the MAS to be constructed. According to proposed framework, run-time model is initialized from the design model. Initialized run-time model represents the zero-moment of the system. At the same time, the simulated environment in which the run-time model is located, should be modeled by the simulation environment modeling language by the developer.The run-time model and the simulation environment model are given as inputs to the simulator, while the simulator builds the next state of the MAS by modifying the run-time model. The user is responsible for initiating and using the simulator with the debugging operations. In the next section of this paper, debug operations are introduced.

If a MAS DSML developer wants to re-tailor an existing MAS DSML with the proposed framework, s/he has to derive the abstract syntaxes of the above mentioned sub-languages. The metamodels of these languages could be generated from the existing DSML metamodel. For this purpose, the DSML metamodel should be divided into parts which are static / dynamic, agent internal / MAS organization, etc. and the developer determines which of them can be input into the simulation environment. By this way, initial version of the metamodels of the sub-languages become evident. This process can be completed automatically (or at least) semi-automatically in case the metamodel of the DSML is already annotated to provide additional information. After the abstract syntax of the sub-languages are generated, the developer has to create transformation rules between the sub-languages. As a final step, a simulator has to be implemented.

Once a concrete implementation of the framework is already available for a specific MAS DSMLs, it is also possible to inherit the same implementation for another MAS DSML by just re-engineering the model transformations via constructing transformations between the metamodel of the new DSML and current run-time language inside the framework. It is worth indicating that providing a horizontal transformation between the metamodels of MAS DSMLs can also enable debugging for the target DSML when the framework is already applied for the source DSML in the transformation. A discussion on how the mechanism for bridging MAS DSMLs with horizontal model transformations can be found in [24].

3 Debugging Operations

In this section, we discuss some debugging operations that may take place in the simulation environment to be built within the proposed framework.

3.1 Steps

In code debugging, stepping through code is often used by the users to understand how system states change during execution. This brings an opportunity for exposing a detailed representation of the behavior of the relevant system. If we want to use this kind of approach for debugging on models, it can be considered as changing current model state to the next state according to operational semantics provided by the simulator. Since the coding structure in the current agent programming languages / environments (such as JADE [1]) is generally similar to the coding structure of the object-oriented paradigm, the run-time states of the program is a kind of call hierarchy. In its simplest form, we can observe that the software agents can call their plans, which can be triggered by the events in the environment and, if necessary, call for more sub-plans to achieve their goals. As in object-oriented paradigm, stepping over code can usually be achieved in three possible operators: Step into, Step over, and Step out. The definition of these operators within our debugging framework are presented as follows:

Step into: As it is well known, an object is an encapsulation unit in object-oriented paradigm. It encapsulates data and methods and also hides its current state from the outside. Similar to object-oriented paradigm, an agent is also an encapsulation unit covering agent beliefs, goals and plans. These encapsulated elements could be considered as the composite elements of an agent entity in a model. Step into refers to stepping through the model element including any composite elements defined within this model element. For example, when the user tries to step into an agent's internal state, stepping contains all possible plans with all possible actions or sub-plans of the selected plans.

Step over: Step over and step into concepts state a duality. Step over refers to the stepping through the model at the composite level. It may be considered as some kind of filters while debugging. However, it does not mean that underlying elements of the model at the composite level are executed. It only hides them. For example, when the user steps over a plan of an agent, the user just observes elements which compose this agent plan. Actions or sub-plans can still trigger something at lower levels, but that is hidden.

Step out: Step out refers to stepping through a model element from inner composition level of it. For example while the user steps into a plan of an agent, the user has the option to step out from inside this plan to the composition level of the plan. At that state, the simulation continues at the composite level and the details inside the plan are hidden.

3.2 Breakpoints

Basic assertions in programming are breakpoints. When the assertion fails, execution of the program is interrupted. Generally, in breakpoints concept, this situation can be triggered by reaching specific code line. For debugging on models, interruption is defined as pausing the simulation in our framework. Here, the user will place specific breakpoints on the model, allowing the simulation to pause when certain conditions are met. Three possible stopping points are introduced:

State based: This is the case when the simulation is stopped if the model reaches a certain predefined state or a specific pattern within the state. For example, the emergence of a particular event may trigger a plan of an agent, reaching a pattern of beliefs for a specific agent or capturing a predefined communication pattern between agents.

Condition based: When some agent features are defined in the model, provide certain logical conditions which will be checked on these features. For example, if the agent's beliefs are not valid anymore due to the changes in the environment facts, update the agent belief set.

Time based: A predefined time has been reached in the simulation environment. However, this is directly related to the time definition of the simulation environment. (Real-time, scaled real-time, as fast as possible, timeless (discrete event-based), etc.)

3.3 Execution Modes

A simulator allows the transition from one execution mode to another. By this way, the control of the simulation is left to the user. The user can stop or pause and then resume the simulation of the run-time model at any point in time. We define 4 different execution modes and transitions between them. The different execution modes and transitions are illustrated in Figure. 3. These execution modes are briefly described as follows:

Ready: It is the default state of the run-time model. Also, it refers to the first state of the simulation which is also the initial state of a MAS. That state may consist e.g. the initialization of agent beliefs, goals and plans.

Running: When the user starts the simulation, the simulation environment switches to Running mode. In Running mode, the simulation can switch to different simulation modes. However, this is just possible if something such as breakpoint interrupts interrupts the run of simulation. For instance, in a state of the running mode, each agent of a MAS executes an atomic event that is simply a action in a plan of an agent.

Pause: While the simulation continues, the simulation environment can be passed to Pause mode. In Pause mode, the simulation is freezed in the current state. The user can switch back to Running mode at any time. In pause mode, a user can see a screenshot of current

state of a MAS including execution traces of the plans of each agent, communication traces between agents, etc.

Stopped: If the simulation is terminated, it switches to Stopped mode. In this case, it can only be passed to Ready mode. When simulator stops, current MAS model is transformed to the initial state of itself. From this moment on, it cannot be possible to turn back to continue the simulation.

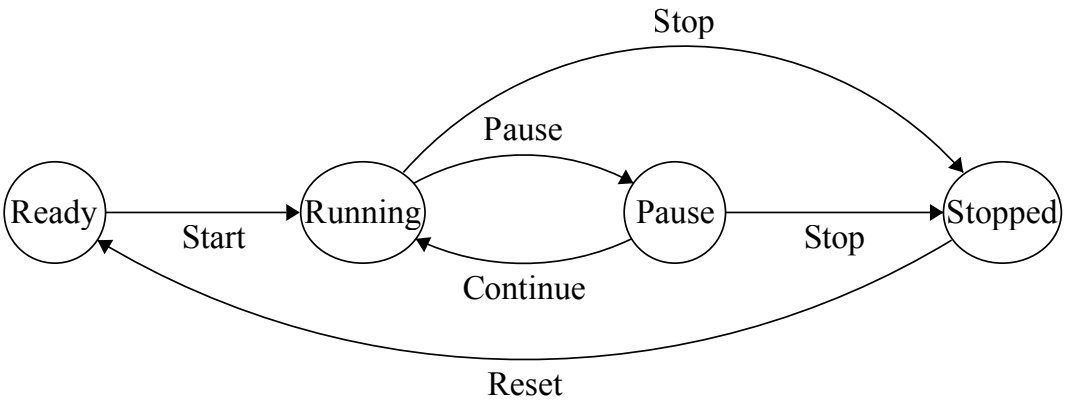


Figure 3 Execution modes transition graph of the simulator.

4 Model Simulator

In the proposed framework, operational semantics of the modelled MAS is provided by a simulator. A number of options are available to define the simulator. For example, directly modifying the run-time model by model transformation rules to obtain next state of the system is one of these options. However, general structure of the simulation algorithms are very similar at a higher level abstraction. The algorithm 1 is an abstract pseudocode of the simulator considered in the framework. As can be seen from the algorithm 1, the simulation is divided into the following functions:

initialize: This function is responsible for generating the initial state of the simulation, which is represented by the MAS run-time model.

terminationCondition: This function takes the run-time model, simulation environment model and the time as inputs and returns "true" if the desired termination condition is provided.

updateBeliefs: This function updates the current beliefs of an agent according to the run-time model and the simulation environment model.

updateGoals: After updating the belief of an agent, the simulator has to decide which goals to be achieved by an agent. In order to achieve updating goal, simulator calls this function with the run-time model, current state of an agent and the simulation environment model.

updateActivePlan: This function uses reasoning mechanism to find a plan to achieve goals of an agent. If there is a plan already in progress, it expects the plan to end in order not to disturb the integrity. If the active plan has ended, it determines a new active plan.

nextAction: This function executes next action of an agent according to the active plan.

increaseTime: This function increases the simulation time by adhering to the time semantics determined according to the simulation setup.

checkForBreakpoints: This function supports to check breakpoints to pause the simulation

pauseSimulation: The simulation environment allows users to set breakpoints and when the condition holds, this function pauses the simulation.

Algorithm 1: Generic Simulation Algorithm.

Input: to be simulated Model (M), Simulation Environment Model (E)
Output: Run-time Model (RM), time (t)
(time) t, (run-time model) $RM \leftarrow$ 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

Every turn of the while loop in the Algorithm 1 runs one step of the simulation . By applying endogenous transformation to run-time model with the help of the defined functions, it creates a snapshot of the next state of the system.

5 Related Work

The most common approaches to debugging of MASs have been to adopt visualization techniques [42, 32, 31, 34].van Liedekerke and Avouris [42] introduce a technique using abstractions to reduce the amount of information being presented to the user during agent development. Nwana et al. [32] provide debugging tools based on multiple views of the computation to limit the information flow by combining results from different views. Poutakidis et al. [34] suggest applying a method for the translation from Agent UML (AUML) to Petri nets that should enable developers to construct, or convert their own protocols into an equivalent Petri net. This generated Petri net is used for debugging MASs by monitoring the exchange of messages between agents. Also, there exist some basic tools visualizing agent states in the development environments [10, 11, 19, 33]. Such tools give information about the current state of an agent while running MAS. Some of them allow users to modify the state of an agent too. In addition, Hindriks[20] presents a more advanced approach enabling users to ask questions about MAS behaviors.

All above approaches are primarily concerned with only providing a visual representation of message exchange between agents in a MAS and do not consider the embracing model of agent internals and agent interactions altogether as in existing MAS DSMLs. We believe that the study introduced in this paper contributes those efforts by composing the static and dynamic aspects of MAS modeling as well as covering the debugging needs of full-fledged MAS DSMLs which own much more complicated modeling environments comparing the visual environments considered in the above studies.

On the other hand, the studies on DSML debugging are rare and recently emerging. Mannadiar and Vangheluwe [29] propose the matching of the concepts of debugging between the programming languages and the DSMLs. This study can be accepted as the starting point for the development of the DSML debuggers. As an application of the conceptual mapping described in [29], Kosar et al. [27] present a DSML with the debugging tool called Sequencer, which is developed for the measurement systems. A similar approach appears in [9] to present a debugger development framework for domain-specific languages (DSLs).

One of the most advanced domain-specific debuggers to date is presented in [45]. Wu et al. provide the reuse of existing, tried and familiar debugging facilities in DSLs. The approach allows the DSL designer to use the internal debugging possibilities of a general programming language by the detailed mapping between model elements and the synthesized code. Lindeman et al. [28] enrich a language definition to support debugging similar to Wu et al.

Blunk et al. [4] propose a method to model debuggers for a DSML. This approach requires a metamodel-based description of the abstract syntax of the language. Debugging is defined by the process semantics at the meta-modeling level where possible run-time states are modeled as part of a DSL metamodel and the transitions are defined as a transformation from one model to another. The omniscient debugger in executable DSMLs (xDSML) is explored in [5]. In that study, the domain-specific metamodels are produced, as well as a domain tracking manager is generated to enable developers to use a general all-inclusive debugger with xDSMLs.

Although these noteworthy DSML debugging studies have guided us on evaluating design issues and helped the derivation of the framework proposed in this paper, none of them considers the needs of debugging in MAS DSMLS and in fact, the agent domain is already out of these studies' scope. Being inspired from these efforts, possible ways of constructing debugger mechanisms for MAS DSMLs are investigated in our previous work [41] which, to the best of our knowledge, is the only work so far concerned with providing debugging on MAS DSMLs.

6 Conclusion

A conceptual generic debugging framework supporting the design of both agent internals (plans, goals, etc.) and interaction between agents inside the modeling environments of MAS DSMLs has been introduced in this paper. The debugging framework is composed of 4 different metamodels and a simulator. Use of the proposed framework starts with modeling a MAS using a design language and transforming design model instances to a run-time model. According to the framework, the run-time model is simulated on a built-in simulator for debugging. The framework also provides a control mechanism for the simulation in the form of a simulation environment model. With this model, simulation scenarios can be sent to the simulator as an input. Hence, it allows a user to model agent behaviors and environment responses while the simulation is running. Finally, the framework supports the visualization of models while they are executed on the simulation environment.

In our future work, a concrete implementation of the generic framework will be completed for one of the newest MAS DSMLs, called DSML4BDI [26] in order to support the debugging of software agents according to Belief-Desire-Intention model. In fact, we already divided the metamodel of DSML4BDI into the parts to generate the metamodels of the sub-languages. So, we will provide transformation rules among the metamodels of the design language and the visual language. Finally, we will implement a simulator, which takes the instances

of the run-time language and the simulation environment language as inputs in order to modify both the run-time model step-by-step and generate visual models for each step in the simulation. Beside that, we will investigate the ways of adopting different debugging approaches such as omniscient debugging[6], model slicing[38] and algorithmic debugging[37] in the framework.

References

1. Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*. John Wiley & Sons, 3 edition, 2007.
2. Federico Bergenti, Eleonora Iotti, Stefania Monica, and Agostino Poggi. Agent-oriented model-driven development for JADE with the JADEL programming language. *Computer Languages, Systems & Structures*, 50:142–158, 2017.
3. Ghassan Beydoun, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge J Gomez-Sanz, Juan Pavon, and Cesar Gonzalez-Perez. FAML: A Generic Metamodel for MAS Development. *IEEE Transactions on Software Engineering*, 35(6):841–863, 2009.
4. Andreas Blunk, Joachim Fischer, and Daniel 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*, volume 5719, pages 149–164. Springer, 2009.
5. Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xDSMLs. In *Proceedings of the 2015 ACM SIGPLAN Int. Conf. Software Language Engineering (SLE 2015)*, pages 137–148, 2015.
6. Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for executable DSLs. *Journal of Systems and Software*, 137:261–288, 2018.
7. Moharram Challenger, Sebla Demirkol, Sinem Getir, Marjan Mernik, Geylani Kardas, and Tomaz Kosar. On the use of a domain-specific modeling language in the development of multiagent systems. *Engineering Applications of Artificial Intelligence*, 28:111–141, 2014.
8. Moharram Challenger, Marjan Mernik, Geylani Kardas, and Tomaž Kosar. Declarative specifications for the development of multi-agent systems. *Computer Standards & Interfaces*, 43:91–115, 2016.
9. Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures*, 44(Part A):89–113, 2016.
10. Rem Collier. Debugging agents in agent factory. In *International Workshop on Programming Multi-Agent Systems*, pages 229–248. Springer, 2006.
11. Mehdi Dastani, Jaap Brandsema, Amco Dubel, and John-Jules Ch Meyer. Debugging BDI-based multi-agent programs. In *International workshop on programming multi-agent systems*, pages 151–169. Springer, 2009.
12. Joao Faccin and Ingrid Nunes. A tool-supported development method for improved BDI plan selection. *Engineering Applications of Artificial Intelligence*, 62:195–213, 2017.
13. Josep Silva Galiana. The New Generation of Algorithmic Debuggers. In *1st Symposium on Languages, Applications and Technologies (SLATE 2012)*, page 3, 2012.
14. Enyo José Tavares Gonçalves, Mariela I Cortés, Gustavo Augusto Lima Campos, Yrleyjander S Lopes, Emmanuel SS Freire, Viviane Torres da Silva, Kleinner Silva Farias de Oliveira, and Marcos Antonio de Oliveira. MAS-ML 2.0: Supporting the modelling of multi-agent systems with different agent architectures. *Journal of Systems and Software*, 108:77–109, 2015.
15. Christian 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*, pages 233–240, Estoril, Portugal, 2008. International Foundation for Autonomous Agents and Multiagent Systems. AAMAS '08. URL: `http://dl.acm.org/citation.cfm?id=1402383.1402420`.

16 Christian Hahn, Cristian Madrigal-Mora, and Klaus Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, 2009.

17 Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

18 Ábel Hegedüs, István Ráth, and Dániel Varró. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica Electrical Engineering and Computer Science*, 56(3):71–82, 2012.

19 Koen V Hindriks. Programming rational agents in GOAL. In *Multi-agent programming: languages and tools and applications*, pages 119–157. Springer, New York, 2009.

20 Koen V Hindriks. Debugging is explaining. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 31–45. Springer, 2012.

21 Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.

22 Jomi F Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous agents and multi-agent systems*, 20(3):369–400, 2010.

23 Geylani Kardas. Model-driven development of multiagent systems: a survey and evaluation. *The Knowledge Engineering Review*, 28(04):479–503, 2013.

24 Geylani Kardas, Emine Bircan, and Moharram 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):875–912, 2017.

25 Geylani Kardas and Jorge J. Gomez-Sanz. Special issue on model-driven engineering of multi-agent systems in theory and practice. *Computer Languages, Systems & Structures*, 50:140–141, 2017.

26 Geylani Kardas, Baris Tekin Tezel, and Moharram Challenger. Domain-specific modelling language for belief–desire–intention software agents. *IET Software*, 12(4):356–364, 2018.

27 Tomaž Kosar, Marjan Mernik, Jeff Gray, and Tomaž Kos. Debugging measurement systems using a domain-specific modeling language. *Computers in Industry*, 65(4):622–635, May 2014. doi:10.1016/j.compind.2014.01.013.

28 Ricky T Lindeman, Lennart C L Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 127–136, 2012. doi:10.1145/2189751.2047885.

29 Raphael Mannadiar and Hans Vangheluwe. Debugging in Domain-Specific Modelling. In *Lecture Notes in Computer Science*, volume 6563, pages 276–285. Springer, 2011.

30 Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: a framework for generating domain-specific property languages. In *International Conference on Software Language Engineering*, pages 1–20. Springer, 2014.

31 Divine T Ndumu, Hyacinth S Nwana, Lyndon C Lee, and Jaron C Collis. Visualising and debugging distributed multi-agent systems. In *Proceedings of the third annual conference on Autonomous Agents - AGENTS '99*, pages 326–333, New York, New York, USA, 1999. ACM Press. doi:10.1145/301136.301220.

32 Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collis. Zeus: A toolkit for building distributed multiagent systems. *Applied Artificial Intelligence*, 13(1-2):129–185, January 1999. doi:10.1080/088395199117513.

33 Alexander Pokahr, Lars Braubach, Andrzej Walczak, and Winfried Lamersdorf. Jadex-engineering goal-oriented agents. *Developing multi-agent systems with JADE*, pages 254–258, 2007.

34 David Poutakidis, Lin Padgham, and Michael 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, pages 960–967. ACM, 2002.

35 Amal El Fallah Seghrouchni and Alexandru Suna. Claim and sympa: A programming environment for intelligent and mobile agents. In *Multi-Agent Programming*, pages 95–122. Springer, 2005.

36 Onn Shehory and Arnon Sturm. *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*. Springer-Verlag Berlin Heidelberg, 2014.

37 Josep Silva. A survey on algorithmic debugging strategies. *Advances in engineering software*, 42(11):976–991, 2011.

38 Josep Silva. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR)*, 44(3):12, 2012.

39 Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *EEE Transactions on computers*, 12:1104–1113, 1980.

40 Baris T. Tezel, Moharram Challenger, and Geylani Kardas. A metamodel for Jason BDI agents. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51, pages 8:1—-8:9, 2016.

41 Baris Tekin Tezel and Geylani 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.

42 Marc H Van Liedekerke and Nicholas M Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995.

43 Simon Van Mierlo. *A multi-paradigm modelling approach for engineering model debugging environments*. PhD thesis, University of Antwerp, 2018.

44 Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

45 Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073–1103, 2008.

46 Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.