Towards Providing Debugging in the Domain-Specific Modeling Languages for Software Agents

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

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

ABSTRACT

Domain-specific modeling languages (DSMLs) for Multi-agent Systems (MAS) mostly provide checks and validations on modeled systems according to the related syntax and semantics descriptions. However, they do not have a built-in support for debugging MAS models which makes the control of model correctness difficult. Hence, in this paper, we present our ongoing work which aims at providing debugging inside MAS DSMLs. We describe two possible ways of deriving debuggers for MAS DSMLs. The first alternative is based on the construction of a mapping between MAS model entities and the generated code while the second one considers the metamodel-based description of the operational semantics of executing agents. Pros and cons of each approach are also discussed.

1 INTRODUCTION

Software agents are autonomous software entities acting to fulfill its duties on behalf of users. Multi-agent systems (MASs) include multiple interacting software agents within an environment to provide solutions for complex systems which cannot be easily solved with individual agents or monolithic systems. However, the development of MASs is not trivial due to the various agent properties such as autonomy, responsiveness, and proactiveness, and the need for realization of the many different agent interactions [7].

Agent-oriented software engineering (AOSE) [18] researchers define various agent metamodels (e.g. [3, 13, 19]), which include fundamental MAS entities and relations. Originating from these metamodel definitions, many model-driven agent development approaches [14] are provided in order to facilitate design and implementation of software agents by enriching MAS metamodels with some defined syntax and semantics (usually translational semantics). In AOSE, perhaps the most popular way of applying model-driven engineering (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 [15]. Proposed MAS DSMLs (e.g. [2, 6, 11, 12, 16] 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. Although IDEs of these MAS DSMLs provide some sort of check and validation on modeled systems according to the related DSML's syntax and semantics descriptions, 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. Hence, in this paper, we present our ongoing work which aims at providing debugging inside MAS DSMLs. 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. We currently investigate how MAS DSMLs and debugging concepts and procedures pertaining to general-purpose languages (GPLs) can be bridged. The paper presents the initial findings of this investigation by describing two ways of deriving debuggers for MAS DSMLs. A brief evaluation of these two approaches are also included.

In section 2, we introduce possible debugging approaches which can be applied for MAS DSMLs. In section 3, we discuss the advantages and the disadvantages of the proposed debugging approaches by taking into account software agents and MAS context. Finally, section 4 concludes the paper.

MDEbug'18, October 2018, Copenhagen, Denmark
2018.

2 DEBUGGING APPROACHES FOR MAS DSMLS

In the context of software development, debugging support is mostly provided by a language and an IDE which enable to watch and change executed programs [23]. As indicated in [20], various debugging techniques (e.g. using breakpoints, stepping operators, symbolic execution) are used for GPLs. However, model developers need to debug models at the model level, not at the code level in the domain-specific modeling [17], and this new requirement caused the researchers on developing new debugging approaches for model-driven development and DSMLs. Compared with GPLs, very few debugging methods and tools currently exist for DSMLs. For instance, Moldable Debugger [8] provides the construction of domain-specific debuggers by creating and combining domain-specific debugging operations with domain-specific debugging views. Omniscient debugging, which allows free traversal of the states reached by a system during an execution, is also used in creating debuggers for executable DSMLs (xDSMLs) [5] or improving model transformations [9]. Motivating from these efforts, we investigate different debugging approaches which can be used for MAS DSMLs. Within the scope of our study, two different approaches have been derived so far. The first one is focused on constructing a mapping between MAS model entities and the generated code while the second approach covers the metamodel-based description of the operational semantics of executing agents. The following subsections briefly discuss these alternatives.

2.1 Construction of a Mapping Between MAS Model Entities and the Generated Code

In this approach, we adopt existing, tried and well-known debugging facilities for the domain-specific models of software agents. Constructing a mapping between model entities and the generated code allows the DSML developer to use target language debugging facilities for generating debugging perspectives. For this purpose, we propose a debugging approach based on the DSL Debugging Framework (DDF) presented by Wu et al. [22].

The key technique of the proposed method is the mapping process. This process is recording the link between an agent DSML (e.g. DSML4MAS [12] or SEA_ML [6]) and the generated target language code conforming to a MAS development framework (such as JACK[1] or JADE[2]) implemented with a GPL (e.g. Java). The mapping information required by the approach depends on both the source language (an agent DSML) and the target language (a GPL). The mapping components consist of the "source code mapping", "debugging methods mapping", and "debugging results mapping".

The results of these first two mapping processes along with the generated GPL code are re-interpreted to generate GPL debugging commands for the GPL debugger. The "source code mapping" component is used to determine which entity of the DSML model is mapped to which segment of GPL code. As a side effect of model-to-text transformation, the source code mapping is generated when an agent DSML model is transformed to target language code.

The traditional GPL debugging activities may not be appropriate for the end user of an agent DSML. For this reason, domain-specific debugging activities should be defined to be used in the debugging perspective of the DSML level. So, the "debugging methods mapping" component is used for receiving DSML user debugging commands from the DSML-level debugging perspective to determine what types of debugging commands are needed from the GPL-level command line debugger and explaining how debugging activities at the DSML level are expressed at the GPL level.

The GPL-level debugger sends debugging results to the DSML debugging perspective with the help of the "debugging results mapping" component, which converts GPL debug output messages back to the DSML level. Since the messages in the GPL debugger are command line output that does not contain any information pertaining to the DSML debugging perspective, it is necessary to reconstruct the results to the DSML user perspective.

2.2 Metamodel-based Description of Agent Operational Semantics

Originating from the methods described in [10] and [4], the debugging process, in this approach, is accomplished by the metamodel-based description of a MAS DSML's operational semantics where possible runtime states are modeled as part of the DSML metamodel and transitions are defined as model-to-model (M2M) transformations. To apply this approach, an operational semantics based on the metamodel is required in addition to the abstract syntax of the language. By this way, it allows to access runtime state directly on a model instance and to control execution by the M2M transformation.

[1]JACK Autonomous Software, http://aosgrp.com/products/jack/
[2]JAVA Agent DEvelopment Framework, http://jade.tilab.com/

The key technique of the approach is the step-by-step execution of DSML instances (MAS models). This makes it possible to introduce a generic debugger. Such a debugger can control a program to suspend execution according to active breakpoints, which are based on model elements. The breakpoint can be placed in model elements representing program locations. If an element marked as a breakpoint is included in a M2M transformation query, execution is automatically suspended.

Step Operations are interpreted with model transformation queries for the target model locations, which extract the model elements. Such target locations are loaded with temporary breakpoints, and when one of these breakpoints is reached, execution is automatically suspended. Thus, one step of the execution is provided on the model.

It is worth indicating that the definition of a debugging perspective of a MAS model instance based on the metamodel is possible in this approach, and hence the runtime state of agents can be entirely contained in the MAS model. Thus, the debugging perspective that uses domain-specific concepts at the model level, can be provided to the users.

3 DISCUSSION

At first glance, it may seem that the first approach can be applied to DSMLs developed for software agents. However, there are some difficulties in implementation at this point, mainly originating from using the GPL debuggers for debugging. First, the approach assumes that all generated artifacts of a MAS DSML are executable. However, many MAS DSMLs produce MAS specification / configuration files (e.g. for defining agent beliefs in OWL ontology documents or setting agent goals and plans in XML-encoded files) and hence they can not be included in the debugging process within this approach. For example; some of the artifacts generated from a SEA_ML [6] MAS model instance are ontology documents of the semantic web services interacting with the agents while some of them are the codes pertaining to the target agent execution platform. This is an important shortage of following the first approach in implementing debuggers for MAS DSMLs. Another difficulty is that the generated GPL code has to be used in GPL debugging tools, so the GPL code must be complete. However, due to the high level of abstraction of MAS DSMLs, the generated GPL codes are generally code fragments / templates which are architectural and do not mostly have behavioral logic. This causes a problem before using the generated code in the GPL debugger since existing MAS DSMLs do not have the ability to generate complete codes for implementing MAS [7]. In addition, the DSML users with limited programming skills may not have the ability to complete the generated code.

The second approach seems to be more appropriate than the previous one while developing debuggers for MAS DSMLs since the application of this approach is independent from the GPL debuggers. The implementation would not need to consider whether generated outputs are not executable or the executable artifacts are just template codes. Hence, that approach, i.e. constructing a debugger over metamodel-based description of the agent operational semantics, looks more suitable for complex modeling environments of MAS DSMLs. However, applying this approach also has several difficulties in the implementation phase. The difficulties will be encountered at (1) parts describing the runtime state of a MAS model to be added to the metamodel of the language, and (2) the writing of the rules of the M2M transformation which will refer to transitions between states. Considering the parts that represent the runtime state to be added to the metamodel, if the complexity and the size of the metamodel increase, the number of these elements and the number of relations between both themselves and other elements will also increase dramatically. This is a challenging situation for the language developer who will add these elements to the MAS metamodel. The second problem that may arise in practice is that most of the languages and technologies for M2M conversions are based on rewriting a graph. However, the proposed approach should not only rewrite MAS model instances that are caught by traversing in accordance with the target metamodel, but rather it has to represent the transitions in the runtime by looking at the relevant model elements, relations and properties. In this case, the M2M transformation environment has to be constructed from scratch solely for a specific MAS DSML in order to be implemented and can be used during transformations.

In fact, the difficulty of implementing both of these approaches mainly arises from incomplete and/or informal modeling of runtime agent behaviors in the current MAS DSMLs. In order to eliminate this deficiency, one option can be the construction of transformations between MAS DSMLs and a formalism, such as Petri nets. Hence, models for agent plans and tasks can be complete for execution. Although such a model will include a non-deterministic formalism due to the nature of agent programs, debugging of such models can be assisted with newly emerging tools (e.g. [21]). Model instrumentation [1] can also be used for debugging without modifying the metamodels. Another alternative for debugging agents at runtime can be providing xDSMLs for MAS at first and then benefiting from the existing approaches on debugging xDSMLs (e.g. [5], [9]). However, we should re-engineer both the syntax and the semantics of each different MAS DSML as to be executable or construct a brand new MAS xDSML in order to use this approach.

4 CONCLUSION

We have discussed some possible debugging approaches which may be used for debugging MAS models conforming to MAS DSML specifications. A brief evaluation of these approaches showed that the application of the first approach is easier since it benefits from using already existing 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 it needs addition of parts describing the runtime state of MAS model into the language metamodel and writing the corresponding M2M transformation rules. Implementation of these debuggers can be facilitated by methods consisting of strengthening MAS model formalism and/or re-shaping existing MAS DSMLs as xDSMLs. However, both providing implementation platforms for the proposed approaches and enriching them with these methods need further investigation which will be our future work.

ACKNOWLEDGMENTS

The first author would like to thank TUBITAK-BIDEB for their financial support during his Ph.D. studies.

REFERENCES

[1] M. Bagherzadeh, N. Hili, and J. Dingel. 2017. Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In *Proc. 11th Joint Meeting on Foundations of Software Engineering.* 419–430.

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

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

[4] A. Blunk, J. Fischer, and Daniel A. Sadilek. 2009. Modelling a Debugger for an Imperative Voice Control Language. In *Lecture Notes in Computer Science.* Vol. 5719. 149–164.

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

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

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

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

[9] J. Corley, B. P. Eddy, E. Syriani, and J. Gray. 2017. Efficient and scalable omniscient debugging for model transformations. *Software Quality Journal* 25, 1 (2017).

[10] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. 2000. Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In *Lecture Notes in Computer Science.* Vol. 1939. 323–337.

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

[12] C. Hahn. 2008. A domain specific modeling language for multiagent systems. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)-Volume 1.* 233–240.

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

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

[15] G. Kardas and J. J. Gomez-Sanz. 2017. Special issue on model-driven engineering of multi-agent systems in theory and practice. *Computer Languages, Systems & Structures* 50 (2017), 140–141.

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

[17] R. Mannadiar and H. Vangheluwe. 2011. Debugging in Domain-Specific Modelling. In *Lecture Notes in Computer Science.* Vol. 6563. 276–285.

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

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

[20] S. Van Mierlo, E. Bousse, H. Vangheluwe, M. Wimmer, C. Verbrugge, M. Gogolla, M. Tichy, and A. Blouin. 2017. 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.* 441–446.

[21] S. Van Mierlo and H. Vangheluwe. 2017. 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).* 460–462.

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

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