Debugging for Model Transformations

Jonathan Corley

Department of Computer Science, University of Alabama
Tuscaloosa, AL, USA 35401
corle001@ua.edu

Abstract. In Model-Driven Engineering, the evolution of models is commonly defined using model transformation languages, which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level. Model transformations are also a type of software abstraction that can be subject to human error. This paper presents a research proposal to investigate applying three traditional bug localization approaches (i.e., Stepwise Execution with Breakpoints, Omniscient Debugging, and Query-based Debugging) to model transformations. The research proposal outlines a plan to investigate each technique along with empirical assessments to evaluate each technique individually and in combination.

1 Problem and Motivation

Model-Driven Engineering (MDE) has emerged as a software development paradigm that can assist in separating the issues of the problem space of a software system from the accidental complexities of implementation in the solution space. MDE approaches often use customized domain-specific modeling languages (DSMLs) that capture the intent of a particular group of end users through abstractions and notations that fit a specific domain of interest. Like all software systems, evolution also occurs in software models. In MDE, the evolution of models is commonly defined using model transformation languages (MTLs), which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level. Model transformations are also a type of software abstraction that can be subject to human error. Thus, traditional approaches to bug localization may also be applied to assist in locating errors in model transformations.

Debugging is a common task that all software developers encounter across different software artifacts. Seifert and Katscher [1] state that "the search for defects in programs has become a common activity of every software developer's life." Despite the longstanding need for debugging support, the state of tool support for debugging has changed little over the past half century [1] compared to other advances in development methodologies and techniques. Several novel approaches to debugging have been introduced, such as omniscient (or back-in-time debugging) [7, 15] and query-based debugging [2, 8]. However, commercial debugging tools available to programmers are focused primarily on stepwise execution of code and utilization of breakpoints [2]. The tools available for MDE are not exceptional in this regard and are, in fact, less mature than tools available for traditional general-purpose programming languages (GPPLs). With respect to MDE research, Mannadier and Vangheluwe [6] observed, “very little attention has been paid to debugging.”

One of the goals of MDE is to improve developer productivity. This goal is obtained primarily through raising the level of abstraction away from the solution domain by focusing on models and model transformations that focus on the problem domain. Despite the focus on models and model transformations as opposed to GPPLs, traditional development concerns such as debugging must still be undertaken by developers adopting MDE practices. Bran Selic [16] commented that if developers are not satisfied with the “day-to-day” application of MDE, then MDE will be rejected in practice. One of the most common tasks undertaken by software developers is debugging. Therefore, improved debugging approaches for model transformations may improve the acceptability of MDE from an industrial perspective. An advanced debugging focus in MDE may also aid in attaining the goal of improved developer productivity.

My research explores the application of several debugging techniques, including stepwise execution with breakpoints, omniscient debugging, and query-based debugging. The research is conducted within the context of AToMPM [19], a graphical modeling tool that supports DSMLs, as well as custom MTLs. The research plan will empirically evaluate these techniques to identify the effectiveness of each approach, as compared to debugging without any formal tool support. The shortcomings or tradeoffs associated with each debugging approach will also be observed and investigated. In addition, the synergistic advantages of combining multiple debugging techniques will be explored.

2 Related Work

The majority of existing work concerning debugging has focused on the application of debugging approaches to GPPLs, such as C and Java. A wide variety of tools and techniques that aid developers in the process of debugging have been created, studied, and evolved. A number of different approaches have been introduced including the traditional combination of breakpoints and stepwise execution, as well as more advanced approaches, such as Omniscient (also referred to as Back-In-Time) Debugging and Query-Based Debugging.

2.1 Stepwise Execution with Breakpoints

Stepwise execution is the most commonly implemented feature for debugging support. Stepwise execution allows the developer to observe hidden state information dynamically during execution and in many implementations to alter state information, or even the behavior of the system. Some minor differences were observed between tools that were largely derived from differing features of the transformation language (e.g., AToM3 [11] allows developers to manually control rule scheduling, which would normally be scheduled using a nondeterministic method). A stepwise execution environment generally possesses the following features: play, pause, stop, and step. Play allows for continuous execution; pause suspends execution at the current step allowing the developer to closely examine and possibly alter details of the current system state; stop terminates execution leaving the system in the current state and closes the dynamic environment. Step allows the developer to incrementally progress the execution environment in three different ways: step-over, step-in, and step-out. Numerous MDE tools (e.g., TROPIC [4], GReAT [14], ATL [12], TEFKAT [13], UML Model Debugger [6], AToM3 [11], VIATRA2 [10], AGG [5], and Fujaba [9]) provide basic debugging support in the form of stepwise execution facilities. A breakpoint is a feature commonly associated with stepwise execution environments that enables the developer to select a specific point in the execution to pause the system in advance. TROPIC, ATL, TEFKAT, and UML Model Debugger were identified as providing support for breakpoints and stepwise execution.

2.2 Query-based Debugging

An exploratory survey I conducted on the state of debugging support for model transformations revealed only one MDE tool, TROPIC [4], which offers capabilities beyond stepwise execution and breakpoints. TROPIC supports QVT, the transformation language proposed by the OMG [17], in a novel way. TROPIC converts the transformation specified in QVT and the associated source and destination models into a customized Petri Net referred to as a Transformation Net (TN) [4]. The conversion to a TN provides an explicit definition for the operational semantics of the transformation. TROPIC provides the standard stepwise execution, as well as an interactive query console. A query console enables developers to specify OCL queries referring to the TN, which represents the transformation system. The tool support focuses on the TN, which consists of places, transitions, and tokens. A naming scheme links the places and transitions to the original source and destination models, as well as the transformation language, but any carefully constructed formalism is lost in translation and replaced with the TN formalism.

Query-based debugging has been explored more extensively in the context of GPPLs. Both static and dynamic approaches to query-based debugging have been investigated [2, 8]. The Whyline [2] is a noteworthy query-based debugger identified from the literature on GPPLs. The Whyline focuses on providing a guided debugging session by offering suggested queries to the user. The queries presented by The Whyline focus on questions such as "why does property x of object y have this value?" and "why is property x of object y not set to this value?" The suggested queries focus on exploring the reasons behind a failure. This differs from other query-based debuggers that offer a query language which requires the developer to provide custom queries (e.g., the language introduced by Lencevicius [8]). The Whyline also presents one type of query language that uses a natural language, which contrasts sharply with some other languages that adopt a syntax closer to the programming language being debugged (e.g., the query language introduced by Lencevicius is based on the Java conditional language elements [8]).

2.3 Omniscient Debugging

In my survey of debugging support for MDE, no tool was identified that supports Omniscient (back-in-time) debugging. Omniscient debugging is a technique that enables a developer to reverse the execution of a system. This technique can be considered an extension to stepwise execution with a step-back or even step-back-to option. Omniscient debugging has been explored in the context of GPPLs [7, 15], but it appears from our survey that no current work exists in the context of MDE.

The existing work regarding omniscient debugging is often challenged by issues regarding efficiency. In order to step backwards through the execution of the system effectively and efficiently, a trace of the execution history must be maintained. Here, an effective technique is described as one which enables developers to reach all points of interest; and an efficient technique is one which minimizes two competing concerns, time to reach a point of interest and memory consumption. However, maintaining the needed trace information can introduce a state space explosion which opposes the goal of minimizing space consumption. Current research indicates several basic techniques to control the state space explosion. The first technique maintains all information within a rolling window of the execution history. The rolling window is a specified number of steps and represents the portion of the execution history that is able to be navigated. The second technique enables traversal through the entire execution history, but only remembers items of interest, which can be chosen either before or during execution depending on implementation. The final technique maintains all trace information initially and dynamically removes trace information related to any element that is no longer referenced in the current step of execution. This final technique functions similar to garbage collection in many modern languages (e.g., Java). Each technique focuses on limiting the amount of information maintained while maximizing the ability to review prior states of the system.

3 Proposed Solution: Evaluating MT Debugging Approaches

The two primary goals of my research are: 1) to investigate Step-wise Execution with Breakpoints, Query-based debugging, and Omniscient debugging in a manner that can support any transformation language, and 2) to evaluate Query-based debugging and Omniscient debugging empirically in comparison to the more commonly supported technique of stepwise execution with breakpoints. This will be a human-based evaluation using a control group that will not be provided any formal debugging support as a base point for comparison.

3.1 Plans for Supporting Debugging of Model Transformations

To achieve the first goal of my research, my target platform will be AToMPM (A Tool for Multi-Paradigm Modeling) [19], a generic graphical modeling environment. AToMPM runs in the browser and is extensible through a plugin feature. AToMPM supports TCore as the root of its transformation engine, which can be used to implement any arbitrary MTL [18]. Using TCore as the basis for all transformation languages will enable the investigation of debugging tool support in a generic manner for any transformation language. For example, stepwise execution will be tied to TCore in the underlying implementation, and will thus be available to any language that utilizes the TCore implementation.

The investigation of the debugging facilities for AToMPM will be completed in three phases. The first phase, currently in progress, will introduce stepwise execution with breakpoints and tracing for model transformations. Stepwise execution will include the three basic steps (step-over, step-in, and step-out) and a preliminary implementation is in place that targets TCore and MoTif [20], a declarative transformation language provided with AToMPM. Currently, I am investigating an approach that will incorporate a higher-order transformation (HoT) that converts MoTif into TCore as an example for applying the approach to an arbitrary MTL. After stepwise execution is completed, breakpoints will be introduced using a similar strategy.

The second phase of the investigation will introduce tracing facilities into the transformation engine, which can track the complete history of a model transformation execution. The tracing facilities will be needed to implement both query-based debugging and omniscient debugging. The key difficulty will be the management of space consumption in the tracing strategy. A potential solution could be to maintain either a set of large deltas (more than a single step in the model transformation execution) or jump points (a complete copy of the model at a given point in the execution history). Large deltas could prove superior in the case where a small portion of the model is being altered frequently. Maintaining a large delta rather than incremental deltas would eliminate any changes that are only temporary. Jump points could prove superior in the case where large portions of the model are being altered frequently. The jump point would contain more information than the large delta, but would allow for state information to be reached more quickly because the entire model could be reloaded from the jump point. These techniques focus on limiting the amount of space consumed by reducing redundancies in the trace and could be further enhanced by applying any of the previously discussed techniques from the existing omniscient debugging literature. After tracing facilities have been explored, omniscient debugging will be investigated using the trace information to allow the developer to traverse back through the execution of a model transformation.

The third, and final, phase of the research will study query-based debugging in the MDE context. The development of the query language will be informed by the results of a study that will be completed in Fall 2013. This study will explore how developers form and use queries during a debugging session. The intent is to provide a query language most closely resembling that used naturally by developers, which will also enable accurate and efficient query evaluation. These competing concerns will likely involve some measure of compromise in the development of the query language. I would also like to explore the possibility of introducing suggested queries based on current contextual information during a debugging session similar to The Whyline. A query may have many results and, depending on the scope of the query, each result could be very large. Therefore, one challenge that will need to be addressed is how to display query results in an efficient and effective manner. An efficient visualization technique will minimize screen space required to display query results. An effective visualization technique will convey the query results in a manner that enables developers to select the relevant results quickly, which minimizes the impact of searching query results on the bug localization process.

3.2 Plans for Evaluation and Validation

The second goal of my research is to grow the body of empirical evidence regarding debugging for model transformations. Currently, there is a significant lack of empirical research in this area. Existing work largely focuses on case studies that step through a contrived scenario to illustrate the usefulness of the proposed work. This work does illustrate the usefulness of a technique, but does not provide any empirical evidence of the gains in productivity provided by using the technique or any other insight into the effect of the technique in use. I plan to empirically evaluate three techniques separately and in combination. The studies will be a series of human-based experiments. The experiments will utilize a control group that is not provided any debugging tool support. The focus of the experiments will be on evaluating several criteria, such as those addressed by the following questions:

1. What, if any, gains can be achieved by utilizing the experimental techniques as compared to the control group with regards to accuracy, recall, and efficiency?
2. How do developers use the new techniques? This will be accomplished with qualitative analysis of observations made during the experiment.
3. How do developers feel about the techniques and corresponding implementation? This will be accomplished with a post-survey that asks developers about their opinions concerning the technique used.
4. Are techniques synergistic? This will be accomplished by replicating the experiment providing an environment with all experimental debugging techniques.

The results of this series of experiments will be used to identify the positive and negative aspects of the proposed techniques, both individually and in combination, which could lead to further research questions. In addition, the experiments will be designed to be replicated by other researchers. A packet including how to implement the experiment and all relevant materials (e.g., transformations and models) will be made publicly available. The replication packets will enable future researchers to validate and directly compare their results to my experiments. The replication packets will enable future researchers to broaden the scope of these results with their own work.

I also plan to evaluate the various tracing implementations described in Section 3.1. The experiment will be an automated laboratory experiment in which several benchmark transformations will be used to identify space consumption and any slowdown produced by each implementation. These experiments will evaluate the strengths and weaknesses of each tracing technique.

4 Expected Contributions in Model Transformation Debugging

My proposed research will provide two primary contributions. The first expected contribution is an example implementation of generic debugging facilities for MTLs, including generic tracing facilities, as well as three debugging techniques: stepwise execution with breakpoints, omniscient debugging, and query-based debugging. A technique will be investigated that provides support to arbitrary MTLs utilizing TCore as a basis for the underlying implementations. Several additional challenges will be explored including: efficient algorithms for collecting trace information for model transformations, development of an expressive and intuitive query language that also enables accurate and efficient queries, and techniques for displaying query results in an efficient and effective manner.

The second expected contribution is a series of replicated experiments that will investigate the three identified debugging techniques, as well as provide a basis for comparing future research in this area. The experiments will investigate the effectiveness of the three debugging techniques (stepwise execution with breakpoints, omniscient debugging, and query-based debugging) as compared to a control group that will not utilize formal debugging facilities. The experiments will also evaluate a number of qualitative measures, including: how developers utilize the debugging techniques, how developers feel concerning the debugging techniques (e.g., comfort with the technique and perception of effectiveness), and if the techniques are synergistic (i.e., if an environment that provides facilities for multiple techniques prove more or less effective than an environment that does not). Finally, a replication packet containing all of the necessary information for other researchers to replicate the experiments independently will be provided. The replication packet will enable future researchers to independently verify the results of these experiments, as well as compare future novel debugging techniques directly with the three techniques investigated in my research. The experiments have the potential to cultivate the existing body of empirical evidence regarding debugging of model transformations, as well as enable future researchers to add to the body of evidence provided by the experiments.

5 Current Status and Timeline

Currently, I am in the first phase of the investigation, which is focused on adding stepwise execution with breakpoints to AToMPM. I am also currently working with a small team to rework the backend of AToMPM to provide a scalable model for the cloud based multi-paradigm modeling tool. I expect to complete the first phase before the end of Fall 2013. The second phase of investigation will include generic model transformation tracing facilities and the addition of omniscient debugging for AToMPM. Additionally, in this phase I will design and begin the series of replicated experiments that will evaluate the three identified debugging techniques, as well as design and run a separate experiment to evaluate the various tracing implementations. The second phase is planned to be completed before the end of Spring 2014. The third, and final, phase will include the investigation into query-based debugging in

AToMPM and completing the series of replicated experiments. The third phase is planned to be completed before the end of Fall 2014. After all phases are completed, I will begin writing my dissertation with a planned graduation date of May 2015.

References

1. Mirko Seifert and Stefan Katscher: Debugging triple graph grammar-based model transformations. In Proceedings of 6th International Fujaba Days, Dresden, Germany (2008)
2. Andrew J. Ko and Brad A. Myers: Debugging Reinvented: Asking and answering why and why not questions about program behavior. In Proceedings of the 30th International Conference on Software Engineering (ICSE '08). Leipzig, Germany, 301-310. (2008)
3. Raphael Mannadiar and Hans Vangheluwe: Debugging in domain-specific modelling. In Proceedings of the Third International Conference on Software Language Engineering (SLE'10), Springer-Verlag LNCS 6563, Eindhoven, The Netherlands, 276-285. (2010)
4. Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer: Catch me if you can – debugging support for model transformations. In Proceedings of the International Conference on Model-Driven Engineering, Languages, and Systems (MODELS'09). Springer-Verlag LNCS 5795, Denver, CO, 5-20. (2009)
5. Gabrielle Taentzer: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Proceedings of the Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE), Springer-Verlag LNCS 3062, Charlottesville, VA, 446-453. (2003)
6. UML Model Debugger, http://www.research.ibm.com/haifa/projects/services/uml/
7. Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz: Flow-centric, back-in-time debugging. In Proceedings of Objects, Components, Models and Patterns (TOOLS Europe 2009). Springer-Verlag LNBIP 33, Zurich, Switzerland, 272-288. (2009)
8. Raimondas Lencevicius: Query-Based Debugging, *dissertation*, University of California at Santa Barbara, http://www.cs.ucsb.edu/research/tech_reports/abstract.php?id=749. (1999)
9. Fujaba, www.fujaba.de
10. VIATRA, wiki.eclipse.org/VIATRA2
11. AToM3, atom3.cs.mcgill.ca
12. ATL, www.eclipse.org/atl/
13. Tefkat, tefkat.sourceforge.net
14. GReAT, www.isis.vanderbilt.edu/tools/GReAt
15. Guillaume Pothier and Éric Tanter: Back to the future: Omniscient debugging. IEEE Software, 26(6) (November 2009), 78-85. (2009)
16. Bran Selic: The Pragmatics of Model-Driven Development. IEEE Software. 20(5) (September 2003), 19-25. (2003)
17. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1 (formal/2011-01-01), January 2013.
18. Eugene Syriani, Hans Vangheluwe, and Brian LaShomb: T-Core: A Framework for Custom-built Model Transformation Engines. Journal of Software Systems Modeling. (2013)
19. Raphael Mannadiar: A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling, *Dissertation*, McGill University, http://msdl.cs.mcgill.ca/people/raphael/files/thesis.pdf. (2012)
20. Eugene Syriani and Hans Vangheluwe: A Modular Timed Model Transformation Language. Journal on Software and Systems Modeling, 11:1–28 (2011)