Exploring Omniscient Debugging for Model Transformations

Jonathan Corley

Department of Computer Science
The University of Alabama
Tuscaloosa, AL, U.S.A.
corle001@ua.edu

Abstract. Model transformations (MTs) are central artifacts in model-driven engineering (MDE) because they define core operations on models. Like other software artifacts, MTs are also subject to human error and, thus, may possess defects (or bugs). Several MDE tools provide basic support for debugging to aid developers in locating and removing defects. In this paper, I describe my investigation into the application of omniscient debugging features to enhance stepwise execution support for MTs. Omniscient debugging enables enhanced navigation and exploration features during a debugging session.

1 Introduction

Debugging is a basic software engineering task. According to Seifert and Katscher [1], debugging is a common task for software developers. Although debugging is a vital aspect of software development, tool support for debugging has changed little over the past half century [1]. Several novel approaches to debugging have been introduced for general-purpose languages (GPLs), such as omniscient debugging [2]. However, stepwise execution is the most common debugging technique provided in MDE tools (*e.g.,* ATL[1] and TROPIC [3]). Step-wise execution provides basic features that control the execution. *Play*, *Pause*, and *Stop* enable course-grained control of the execution. *StepIn* also executes a single step, moving into any contained scopes as they are encountered. *StepOver* executes until the next step in the current scope is reached. *StepOut* executes until the first statement in the containing scope is reached. The only modeling tool I am aware of that includes an advanced dynamic debugging technique is TROPIC, which provides support for query-based debugging using OCL to pose queries against a Petri-net based translation of the target system.

Omniscient debugging enables a developer to reverse a program's execution history. For example, the developer may start from the location where an error was identified and trace to the location of the fault that caused the failure. This terminology is important to clarify that the underlying cause of an error is not always located at the point in execution where the error is identified. A survey of the existing literature suggests that there is not support yet for omniscient debugging in the MDE context. However, model slicing techniques have been investigated that would aid in identifying these issues [4].

[1] www.eclipse.org/atl/

Omniscient debugging provides a live exploratory approach where the developer may freely traverse the execution history of a given system. Techniques such as query-based debugging and model slicing would be complimentary to omniscient by aiding the developer in selecting points of interest to explore in execution history.

Current work in omniscient debugging is focused on GPLs. However, the technique would also be beneficial in an MDE context. MTs are also subject to errors, and these errors may manifest at a point later in execution than the source of the defect. A common concern is the time and effort required to reach a portion of the system's execution that exercises the defect. If a developer misses the location of a defect by targeting the location of an error, which might be much later than the location of the defect, then the developer would need to restart the execution in a stepwise execution environment. Restarting may be an expensive process as it may require a nontrivial amount of time to reach the desired location and may also require significant manual input from the developer. Omniscient debugging avoids this concern by enabling full traversal (*i.e.,* in either direction) of the system's execution during the debugging session. MDE also exhibits some concerns that are distinct from GPL systems, but would also benefit from omniscient debugging. Declarative MTLs commonly provide nondeterministic rule scheduling. The nondeterminism is acceptable as the rules should not be dependent on ordering to produce correct results. However, it is possible to define transformations improperly such that the ordering of rule execution may vary the final result. In this scenario, it may be difficult to fully trace the source of a defect because the bug may manifest in one execution, but not in a subsequent execution. In these situations, an omniscient debugger would allow the developer to fully explore the context in which the bug manifests.

2 Background and Related Work

Omniscient debugging can be considered an extension of stepwise execution that enables a developer to reverse the execution of the system and revisit previous steps. A key challenge of omniscient debugging is minimizing memory consumption. Several potential solutions have been presented in the GPL literature. Lienhard et al. introduce a strategy similar to garbage collection [2,5,6], removing any elements from history that are no longer referenced. This approach seeks to minimize data collected over time, but in some scenarios these elements may need to be regenerated, thus reducing run-time efficiency. Lewis discussed limiting the portion of history that can be navigated [2], providing a window effect. The advantages and disadvantages of this solution are similar to utilizing garbage collection, but whereas garbage collection would maintain the history of elements currently referenced, the window solution removes all information outside of the current window. Lewis also introduced a third strategy that identifies a subset of the program's elements as being of interest to the debugging process [2] and only records information concerning these elements. This solution can be applied in a static manner (*e.g.,* select elements of interest before playback begins), but Pothier and Tanter also explored a dynamic variant (*e.g.,* select elements no longer of interest during run-time) [7]. This approach creates the challenge of discerning which elements will be of interest. This is particularly a concern for the static approach, which requires foreknowledge of all interesting elements.

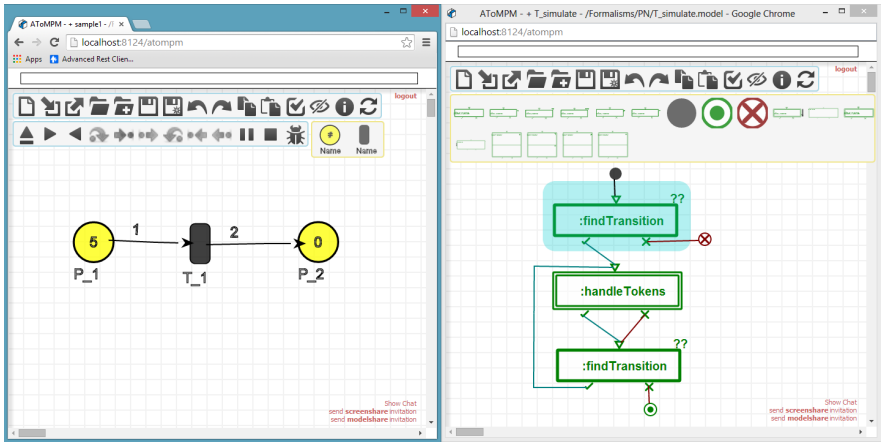

Fig. 1. Screenshot of Debugging Session in AToMPM

3 Omniscient Debugging for Model Transformations

Omniscient debugging is a technique that enables a developer to reverse the execution of a system. It is a logical extension of stepwise execution, that enables a developer to freely traverse through the execution history. The primary goal is to provide free exploration through execution history of a single debugging session dynamically at runtime. I have created a prototype debugger for AToMPM[2] (as seen in Fig. 1) that provides all of the common features of stepwise execution (*i.e., play*, *pause*, *stepIn*, *stepOut*, *stepOver*, and *stop*). However, my implementation of these features takes advantage of the execution trace history to avoid repeating expensive transformation rules. A rule is only executed the first time a particular step in the transformation is reached. If the developer moves back through history and then steps forward again, the model can be altered by simply applying the changes associated with a particular step. My prototype debugger also provides a set of features which mimic stepwise execution, but reverses the flow of the transformation to visit previous points in history. These features are *playBack*, *backIn*, *backOver*, and *backOut*.

The technique collects a history of execution that enables traversal without requiring rules to be re-executed. History is composed of steps, which corresponds to a single transformation rule. A step stores the corresponding rule, a set of all changes that resulted from applying the rule, and any other necessary transformation information (*e.g.,* TCore passes a packet through each subsequent rule [8]). A change stores the element modified and the values associated with the modification. Changes are reversible elements that can be used to either undo or redo the associated modification. Thus, the collection of changes in a step can be used to undo or redo a specific step.

The space complexity upper bound of the history is determined primarily by two factors, the number of steps n and the average number of changes per step m. The structure has a space complexity upper bound of $O(n * (A + m * B))$. The constant

[2] `http://syriani.cs.ua.edu/atompm/atompm.htm`

factor per step A refers to the transformation state information. The additional factor per change B is a minimal set of information regarding the change. Because we define the change at the smallest unit (*e.g.,* a name attribute of a class), B will minimally impact the scalability. Thus, for transformations that affect a large number of elements and have a large number of steps, our structure will perform poorly. However, due to the necessity of maintaining a minimal set of information to provide full traversal, any application of omniscient debugging will encounter similar scaling concerns.

The history stores a minimal set of information, but in extreme circumstances it is not possible to maintain all of the history in memory. While it is most efficient to access history from memory, in practice we must remain within the bounds of memory. Therefore, the history can be configured to utilize a window of active history. As mentioned in Section 2, this technique has been explored previously by Lewis [2]. Our technique adds the ability to store history outside of the current window to permanent storage. Thus, the full history of execution is always available, but accessing some portions of history may require loading a new window from disk. Loading and storing portions of history will impact the runtime performance of the system, but the window enables developers to ensure that the system remains within memory bounds for large-scale scenarios while also having access to the full history of execution as needed. The developer may manually alter the window size at any time. Thus, the developer can control the upper bound of memory consumption incurred by tracing execution.

4 Investigating the Performance Impact of Omniscient Debugging

In addition to space constraints, a debugger must also meet user expectations for runtime performance. I performed a set of experiments to address this concern. Each experiment tested an existing stepwise execution debugger for AToMPM and replicated with the prototype omniscient debugger. The goal of the study was to identify how, if at all, the addition of the required trace collection and other features impact the performance of the debugger. The experiment divided the system execution into the execution of each intermediate step. Thus, the step features that may traverse many steps (*e.g., StepOver*) were not directly tested. Rather, the results were used to determine a baseline of performance. The study compared the running time of a single step forward not executing in history (*i.e.,* actually executing the transformation rather than utilizing history to replay the event), a single step forward in history, and a single step backward in history (the system never executes a rule to move backward). The tests were performed on several Petri-nets with a varying number of simple place-transition-place sets (Fig. 1 illustrates the smallest case tested) using the transformation in Fig. 1 that finds and executes transitions. This experiment was very simple and focused on a basic comparison. Other scenarios may vary from the results reported here, but these results provide initial insight into the impact of providing omniscient debugging features on the runtime performance of the AToMPM debugger.

Fig. 2 provides a summary of these results. The Back entry indicates the time required to step either forward or backward in history. The Update entries, both omniscient and stepwise, indicate the time required to update a place in the Petri-net. The Find entries, stepwise and omniscient, indicate the time required to identify a transition

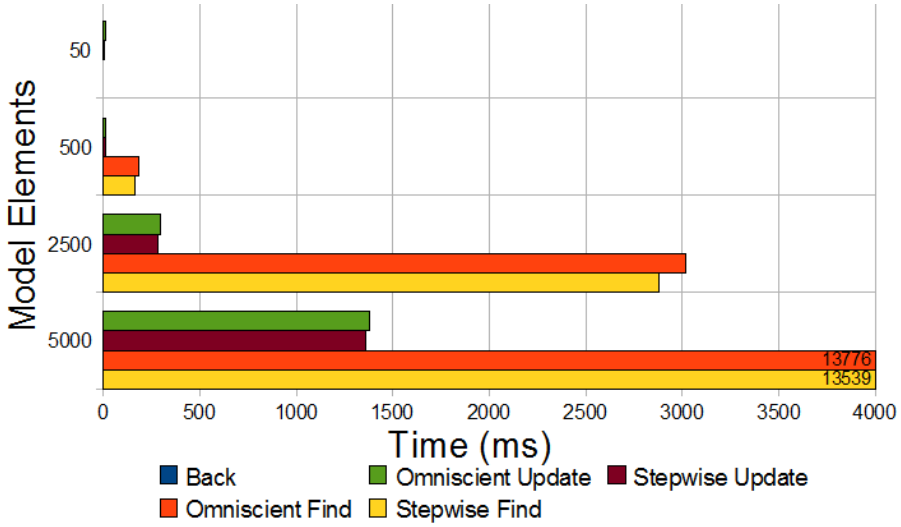

Fig. 2. Summary of Experimental Results

that can be fired in the Petri-net. Overall, the results indicate that the omniscient debugger performs similarly and occasionally better than the stepwise debugger. Additionally, stepping in history (the back entry in Fig. 2 is too small to be visible at current scale) took a maximum of 1ms to process. This is due to the fact that executing in history does not require running the transformation rule. The initial data gathered here indicates that providing the omniscient features has negligible effect on execution time. Future work will further explore the runtime performance as well as memory usage with larger scale models and a larger variety of model transformations.

5 Conclusions

This paper introduces an investigation into applying the basic features of omniscient debugging for MTs. The features discussed enable free traversal and investigation of the full history of execution at run-time. The technique also enables a full record of the changes that occurred during execution. Omniscient debugging enables full traversal and exploration of system execution dynamically at runtime. Omniscient debugging is a promising technique providing an intuitive evolution of current debugging systems.

As future work, I will replicate the performance experiment with a larger scale and variety of models and model transformations. The larger scale and variety will enable more rigorous performance analysis and provide enough data to analyze the memory usage of the debugger. I am also planning a user study to examine the impact of providing omniscient debugging to developers debugging MTs. The user study will also explore how developers use the provided omniscient features.

References

1. Mirko Seifert, Stefan Katscher: Debugging triple graph grammar-based model transformations. In: Proceedings of 6th International Fujaba Days, Dresden, Germany (2008)
2. Bill Lewis: Debugging backwards in time. In: Proc. of the Fifth Int'l Workshop on Automated Debugging, Ghent, Belgium (2003)
3. Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, Manuel Wimmer: Catch me if you can – debugging support for model transformations. In: Proc. of 12th Int'l Conf. on Model-Driven Engineering, Languages, and Systems, Denver, CO, USA (2009) 5–20
4. Ujhelyi, Z., Horvath, A., Varro, D.: Dynamic backward slicing of model transformations. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. (April 2012) 1–10
5. Adrian Lienhard, Julien Fierz, Oscar Nierstrasz: Flow-centric, back-in-time debugging. In: Proc. of Objects, Components, Models and Patterns, Zurich, Switzerland (2009) 272–288
6. Adrian Lienhard, Tudor Gîrba, Oscar Nierstrasz: Practical object-oriented back-in-time debugging. In: Proc. of 22nd European Conf. on Object-Oriented Programing, Paphos, Cyprus (2008) 592–615
7. Guillaume Pothier, Èric Tanter: Back to the future: Omniscient debugging. IEEE Software **26**(6) (nov 2009) 78–85
8. Eugene Syriani, Hans Vangheluwe, Brian LaShomb: T-Core: A Framework for Custom-built Model Transformation Engines. Journal on Software and Systems Modeling **13**(2) (jul 2013)