MODELS: G: Exploring Omniscient Debugging for Model Transformations

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

***Abstract*—This paper discusses a technique for supporting omniscient debugging for model transformations (MTs). MTs are used to define core operations on models. Like programs developed using general-purpose languages (GPLs), MTs are subject to human error and may possess defects (or bugs). Existing Model-Driven Engineering (MDE) tools provide stepwise execution to aid developers in locating and removing defects. In this paper, I describe my investigation into the application of omniscient debugging features for MTs. Omniscient debugging enables enhanced navigation and exploration features during a debugging session beyond those possible in a strictly stepwise execution environment. Finally, the runtime performance is comparatively evaluated against stepwise execution, and the scalability (in terms of memory usage) is empirically investigated.**

I. Introduction

Debugging is a basic software engineering task. Seifert and Katscher state debugging is a common task for software developers [1]. Despite the common need for debugging in 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 GPL tools (*e.g.,* Eclipse[1] and Visual Studio[2]) and Model-Driven Engineering (MDE) tools (*e.g.,* ATL[3] and TROPIC [3]). Step-wise execution enables developers to control the execution of the system and view normally hidden state information. *Play*, *pause*, and *stop* enable course-grained control of the execution. *StepIn* 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 revert a software system to prior state dynamically at runtime. This allows developers to investigate a system starting from the location where an error was identified and trace to the location of the fault that caused the failure. These three terms (error, failure, and fault) each receive a distinct definition in the IEEE standard glossary of software engineering terminology. This distinction highlights the fact that visible signs of a defect may not manifest at the location of the defect. Omniscient debugging provides facilities to help developers explore these complex errors. A survey of the existing literature suggests that there is a distinct lack of support for omniscient debugging in MDE tools. However, model slicing techniques have been investigated that could aid in addressing these issues [4]. Techniques such as query-based debugging and model slicing would be complimentary to omniscient debugging by aiding the developer in selecting points of interest to explore in execution history.

Existing literature for omniscient debugging focuses on GPLs. However, MTs are also subject to errors, and these errors may not manifest at the location of the defect. If a developer were to misidentify the location of a defect by targeting the location of an error, a traditional debugger would require restarting the system. Restarting can be expensive, requiring a nontrivial amount of time to re-execute or a significant delay due to manual input from the developer. Omniscient debugging avoids the need to re-execute to reach a prior state. MTs also have concerns not traditionally found in GPL systems which would benefit from omniscient debugging. Declarative MTLs commonly provide nondeterministic rule scheduling. The nondeterminism is commonly accepted because the rules should not be dependent on ordering to produce correct results. However, it is frequently 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 enables the developer to fully explore the context in which the bug manifests.

II. 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

[1]http://eclipse.org/
[2]https://www.visualstudio.com/
[3]www.eclipse.org/atl/

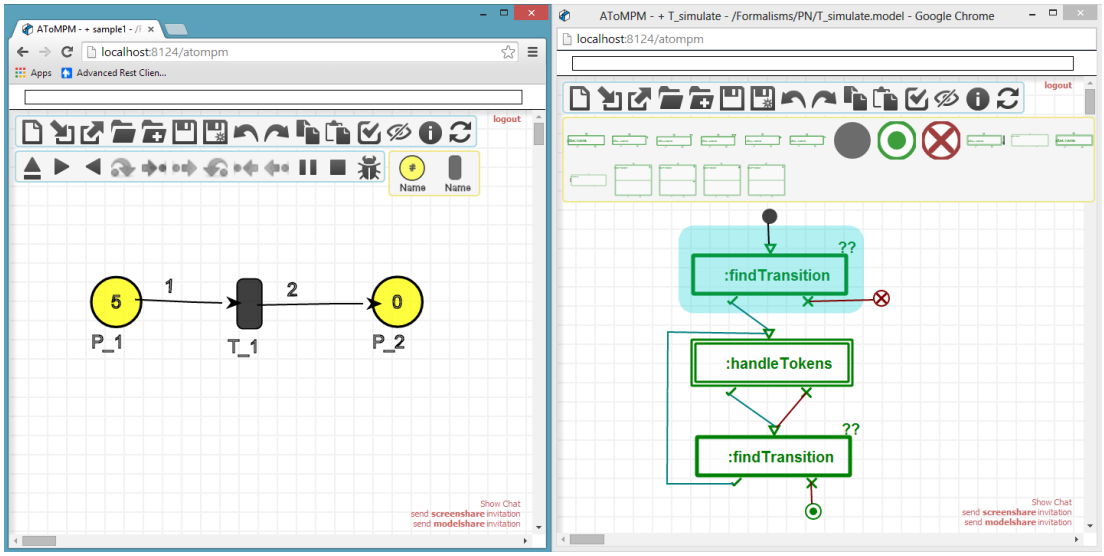

Fig. 1. Screenshot of Debugging Session in AToMPM

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 runtime 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.

Recently, there has been some work in the area of MDE toward the application of omniscient debugging. Simon Van Mierlo presented a proposal toward the debugging of executable models defining simulation semantics [8]. A particular focus of the work addressed handling simulated real-time. My work concerns applying omniscient debugging to model transformations. My work does not concern handling simulated real-time or relating the model entities with generated code.

III. Omniscient Debugging for Model Transformations

Omniscient debugging is a natural extension of step-wise execution enabling reversing the execution of program. I originally developed a prototype within AToMPM as part of my original SRC work [9], but this prototype has since evolved into a more complete tool implementing omniscient debugging for MTs [10]. The AToMPM Omniscient Debugger (AODB) was developed collaboratively [10]. AODB provides the common features of step-wise execution (*i.e., play*, *pause*, *stepIn*, *stepOut*, *stepOver*, and *stop*). However, the features leverage an execution trace history to avoid repeating 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, changes are applied from the stored history. Our debugger also provides a set of features which mimic stepwise execution, but execute in the reverse. These features are *playBack*, *backIn*, *backOver*, and *backOut*. In this section, I will overview the technique described in my SRC submission (Section III-A) as well as the algorithms for more efficient traversal of history (Section III-B) from my collaborative work.

A. Collecting a History of Execution

AODB collects a history of execution to enable traversal without re-executing rules. History is a sequence of steps, each step corresponds to a single transformation rule. A step stores a reference to the rule, a collection of all changes resulting from the executed rule, and any other necessary transformation information such as a set of matched elements. A change is an atomic unit storing the element modified and the associated values (pre and post). Changes are reversible, and can be used to either revert or reapply the associated modification. Thus, we can use the collection of changes in a step to undo or redo that step. As discussed in Section III-B, we can use a subset of changes from a series of steps to undo or redo a set of steps more efficiently.

The space complexity upper bound of history, $O(n * (A + m * B)$, is influenced by two key factors, the number of steps n and the average number of changes per step m. A is a constant referring to the transformation state information,

```
def buildMacroStep(currentStep, targetStep, allModelElements):
  macroStep = Step()
  for element in allModelElements:
    if(currentStep < targetStep):
        stepPos,changePos = revisionsCache[element].before(targetStep)
    else:
        stepPos,changePos = revisionsCache[element].after(targetStep)
    mostLocalChange = history[stepPos][changePos]
    stepPos, changePos = revisionsCache[element].mostRecent(targetStep)
    mostRecentChange = history[stepPos][changePos]
    if((mostRecentChange.getType()=="DELETE")and(mostRecentChange!=mostLocalChange)):
        macroStep.add(mostRecentChange)
    macroStep.add(mostLocalChange)
  return macroStep # macroStep reverts or advances the system to targetStep
```

Listing 1. Macro Step Building Algorithm

```
def modified_build_macro_step(currentStep, targetStep, allModelElements):
  # numChanges represents a sum total of all changes since the transformation began
  changeCount = abs(history[currentStep].numChanges - history[targetStep].numChanges)
  if changesToCheck < allModelElements.size():
    macroStep = Step()
    for step in range(currentStep, targetStep, -1):
      for change in history[step]:
        macroStep.add(change) # add replaces any existing change for the same element
    return macroStep
  else: # use lookup as described in the unmodified algorithm
    return build_macro_step(currentStep, targetStep, allModelElements)
```

Listing 2. Modified Macro Step Building Algorithm

and B is the average size of a change (influenced by the type of data stored in the associated model element). Because we define the change at the smallest unit (*e.g.,* the tokens attribute of a Petri-net place), B will minimally impact the scalability. Thus, for transformations affecting a large number of elements and containing a large number of steps, the structure performs poorly. However, any omniscient debugger will encounter similar scaling concerns due to the need for a trace of execution. The alternative to storing trace information would require converting unidirectional rules into bidirectional rules. This conversion is not always possible. Consider deleting a node. Since any information in the deleted node is lost once deleted, a bidirectional rule would not be possible. Furthermore, other scenarios may exist where reverting the result of a transformation is ambiguous.

Despite storing minimal information, history may eventually exceed the bounds of memory. Thus, AODB maintains a window of active history. As mentioned in Section II, this technique has been explored previously by Lewis [2]. History outside of the current window is stored in permanent storage. 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 impacts the runtime of the system, but the window ensures that the system remains within memory bounds for large-scale scenarios while maintaining access to the full history of execution.

B. Traversing a History of Execution

The goal of most omniscient work is to provide a scalable technique, in terms of memory usage, that enables reversing the execution of a software system. However, we also explored a technique to efficiently, in terms of runtime performance, revert execution by identifying and executing a minimal set of changes. Our technique utilizes the execution history to create a macro step that avoids unnecessary CRUD operations. A macro step contains changes from potentially many traditional steps (*i.e.,* those associated with a single rule). Changes store a complete state for the associated element. Thus, if a model element is found in several changes, then the macro step would use only the most recent change and all other changes can be ignored with one exception. If the element has been deleted, we must recreate the element and then reset the state because the creation of an element always assumes default values within AToMPM. *BackOut*, *backOver*, *stepOut*, and *stepOver* utilize the technique when reverting/replaying previously executed portions of the transformation. These features each have the potential for iterating over an indeterminately large number of changes for the same elements.

AODB constructs macro steps using a structure that provides efficient access to the most recent change. AODB maintains a revisions cache for each element containing a record of every step where the element was altered. The revisions cache enables constant time access to the associated change with minimal lookup time. We can guarantee constant time access for the change, because the history stores all steps in increasing order within a dynamic array structure and each step provides similar facilities for storing changes. Using the revisions cache, we can quickly gather a minimal set of changes for a given set of steps. This algorithm has $O(n*lg(m))$ runtime complexity,

where n is the number of elements in the model that have been altered (only elements that have been altered are stored in the cache) and m is the number of steps where a given element is altered. The $O(n * lg(m))$ runtime complexity upper bound assumes we provide a structure with constant time access for the relevant change and at least $O(lg(m))$ access to change locations stored in the cache. We expect the number of changes for a given element to be small, and the performance to approach $O(n)$ in practice. Listing 1 displays the basic algorithm used to build a macro step.

Thus far we have assumed that when building the macro step, iterating over the changes contained in the step is more costly than iterating over every element in the model to find the required set of changes. When the size of the steps or number of the steps being traversed is large, this assertion does hold true. However, if the size and number of steps is small or the size of the model is large, the cost of iterating over all model elements may be more costly. AODB compares the number of changes that must be iterated over as opposed to the number of elements in the model to decide if iterating over the changes is more or less costly. We still ensure that only a single change is stored for a given element. The upper bound of runtime remains the same since we assure that iterating over the changes will require less iterations than iterating over the full model. AODB maintains a count of the total number of previous changes in history at each step. AODB then uses these counts to determine whether to iterate over the model elements or the full set of changes. The modified algorithm is provided in Listing 2.

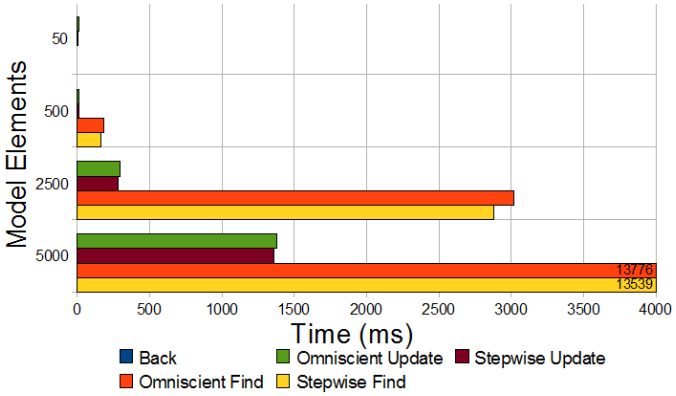

Fig. 2. Summary of Experimental Results

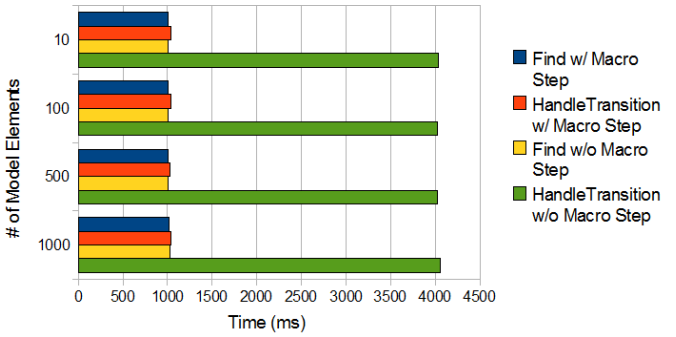

Fig. 3. Comparing Omniscient with and without Macro Step Building Algorithm

IV. Investigating the Performance Impact of Omniscient Debugging

This section offers an overview of the evaluation presented in both my original SRC work [9] and my collaborative work [10]. Section IV-A presents an evaluation of runtime performance as compared to stepwise execution described in my original SRC submission. Next, Section IV-B presents the evaluation of the macro step algorithms from my collaborative work. Finally, Section IV-C presents a discussion of empirical evidence regarding the scalability, in terms of memory usage, as presented in my collaborative work.

A. Runtime Performance: Omniscient v Stepwise

Omniscient debugging literature typically focuses on analyzing performance in terms of memory usage, but runtime performance is also an important concern for online debuggers. To evaluate this concern, an early experiment was performed comparing a stepwise execution only debugger with the prototype which would become AODB. The study focused on determining any impact on runtime performance incurred due to the addition of the omniscient facilities. The experiment measured the time of each intermediate step of execution. Thus, the step features that traverse many steps (*e.g., StepOver*) were not directly tested. The results outlined a baseline difference between the two implementations. The study collected data on the running time of three types of step; a single step forward executing the transformation rule, a single step forward in history replaying a transformation rule, and a single step backward in history reverting a transformation rule. The tests were performed on a series of Petri-nets with an increasing 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. The various model sizes used are displayed in Fig. 2. 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 the results. The Back entry indicates the time required to step either forward or backward in history as the results were identical. The Update entries indicate the time required to update a place in the Petri-net executing the transformation rule. The Find entries indicate the time required to identify a transition that can be fired in the Petri-net executing the transformation rule. Overall, the results indicate that the omniscient debugger performs similarly and occasionally performed 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 approximately 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.

B. Runtime Performance: Impact of Macro Steps

As part of my collaborative work, we performed a modified replication of the previous experiment to illustrate the

performance impact of the macro step building algorithm. The comparison was between the AODB with and without the macro step building algorithm. In the previous experiment the network transmission time was removed when considering time to execute because the network communication overhead was constant for all step operations, but here we include the network communication as the number of communications varies between the two implementations. A summary of the results can be found in Fig. 3. All operations are reverting in history as only the omniscient features performance vary with the inclusion of the macro step building algorithm. The find transformation rule is a single non-composite rule and therefore requires only a single message sent over the network. However, the handle transition rule contains a set of sub-rules dedicated to specific portions of firing a transition and requires four separate messages sent over the network. We analyzed the network communication and found it to add approximately one second (1,000ms) of overhead per message sent. After analyzing the results, we found that the difference in time to re-execute for the handle transition rule without the macro step building algorithm is due to the difference in the number of messages sent. With the macro step building algorithm only a single message is sent for the entire set of sub-rules contained by handle transition, but without the algorithm one message per sub-rule is sent. We also note that the scopes tested do not contain any redundant changes (*i.e.,* multiple changes to the same element). Therefore, we have found that the macro step building algorithm works just as efficiently in the worst case scenario as the standard process, but reduced the network communication overhead for our tested scenario. In other tools where the network communication is not a concern, the macro step algorithm should still prove superior in the case where redundant changes can be eliminated (as discussed in Section III-B), but will not perform worse in the worst case where no redundant changes occur.

C. Memory Performance: The Size of History

We performed a basic analysis of the average size for the constants discussed in Section III-A. Information was collected using the largest model size from the experiments in Section IV-B. The additional information stored for each step, A, was on average 153 bytes. The information stored per change, B,was on average 407 bytes. To provide some context, size for a set of basic elements in Python (language implementing the transformation engine AToMPM) was also recorded. An integer requires 12 bytes. A string with one character requires 22 bytes with each additional character resulting in additional 1 byte added to the size. An object with no fields requires 36 bytes. Other languages provide significantly reduced sizes for similar structures. For example, C uses 2-4 bytes per integer. Therefore, moving to a language such as C could significantly reduce memory consumption.

V. Conclusions

This paper discusses an omniscient debugging technique for MTs along with initial evaluation of AODB, a tool implementing the technique. To my knowledge, there is no existing support for omniscient debugging for MTLs. Omniscient debugging enables free traversal and exploration of the full history of execution dynamically at runtime. The technique also enables capturing a full record of the changes that occurred during execution. Omniscient debugging is a promising technique providing an intuitive evolution of the predominant debugging technique, step-wise execution. The implementation focuses on applying omniscient debugging supporting model transformations in a graphical development environment, including an algorithm for more efficient traversal through history. We identify several concerns unique to the area and evaluate evaluated AODB with regard to scalability, in terms of memory usage, and performance, in terms of time to execute.

At the time of writing, we are conducting a user study. The study will gather empirical evidence regarding the use and impact of the described omniscient features when applied by users. The results of this study will investigate how developers perceive the omniscient features (*e.g.,* are they useful, effective, and efficient) as well as gathering empirical evidence regarding how developers used the features.

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] 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, and 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, pp. 5–20.

[4] Z. Ujhelyi, A. Horvath, and D. Varro, "Dynamic backward slicing of model transformations," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012, pp. 1–10.

[5] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz, "Flow-centric, back-in-time debugging," in *Proc. of Objects, Components, Models and Patterns*, Zurich, Switzerland, 2009, pp. 272–288.

[6] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proc. of 22nd European Conf. on Object-Oriented Programing*, Paphos, Cyprus, 2008, pp. 592–615.

[7] Guillaume Pothier and Èric Tanter, "Back to the future: Omniscient debugging," *IEEE Software*, vol. 26, no. 6, pp. 78–85, nov 2009.

[8] S. V. Mierlo, "Explicit modelling of model debugging and experimentation," in *Proceedings of Doctoral Symposium co-located with 17th International Conference on Model Driven Engineering Languages and Systems*, 2014.

[9] Jonathan Corley, "Towards efficient and scalable omniscient debugging for model transformations," in *Demos/Posters/StudentResearch@ MoDELS*, 2014.

[10] Jonathan Corley, Brian Eddy, and Jeff Gray, "Exploring omniscient debugging for model transformations," in *14th Workshop on Domain-Specific Modeling*, 2014.