Towards Efficient and Scalable Omniscient Debugging for Model Transformations

Jonathan Corley Brian P. Eddy Jeff Gray

The University of Alabama
Tuscaloosa, AL, USA
(corle001, bpeddy)@crimson.ua.edu, gray@cs.ua.edu

Abstract

Model transformations (MTs) are central artifacts in model-driven engineering (MDE) that define core operations on models. Like other software artifacts, MTs may possess defects (bugs). Some MDE tools provide support for debugging. In this paper, we describe an omniscient debugging technique. Our technique enhances stepwise execution support for MTs by providing the ability to traverse, in either direction, the execution history of a live debugging session. We also introduce a proof of concept prototype applying the described technique and a preliminary study of the scalability, in terms of memory consumption, and performance, in terms of time to execute.

Keywords Debugging, Model Transformation, Omniscient

1. Introduction

According to Seifert and Katscher [12], "the search for defects in programs has become a common activity of every software developer's life." Although debugging is a vital aspect of development, tool support has changed little over the past half century [12]. Novel approaches to debugging have been introduced for general-purpose languages (GPLs), such as omniscient debugging [3]. However, stepwise execution is the most common debugging technique provided in MDE tools (*e.g.,* ATL [7]). The only modeling tool we are aware of that demonstrated an advanced dynamic debugging technique is TROPIC[10], which provides support for query-based debugging using OCL to query a Petri-net based translation of the target system.

Omniscient debugging enables a developer to trace through 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. It is important to clarify that the underlying cause of an error is not always located at the point where the error is identified. A survey of the existing literature suggests that there is no support for omniscient debugging in the MDE context.

Current work in omniscient debugging focuses on GPLs. However, the technique would also be beneficial in the MDE context. MTs, like GPLs, are subject to errors. 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 the developer misses the location of the defect by targeting the location of an error then the developer would need to restart the execution in a stepwise execution environment. Restarting may be an expensive process due to the time required to reach the desired location, and may also require manual input from the developer. Omniscient debugging avoids the concern by enabling full traversal (*i.e.,* in either direction) of the system's execution during debugging. MDE also exhibits concerns distinct from GPL systems, but would benefit from omniscient debugging. Declarative MTLs provide nondeterministic rule scheduling, as the rules should not depend on order to produce correct results. However, improperly defined transformations may cause the order of rule execution to 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.

In this paper, we discuss a technique and a proof of concept prototype for omniscient debugging of model transformations. Our discussion will focus on scalability, in terms of memory consumption, as well as the performance of the technique, in terms of execution time. The technique enables for full traversal of the execution (*i.e.,* forward and backward) during a live debugging session.

2. Background and Related Work

In this section, we overview step-wise execution and discuss existing work in the area of omniscient debugging.

2.1 Step-wise Execution

Step-wise execution of a program is the most common form of tool support for debugging and available in the majority of IDEs (*e.g.,* Eclipse). Step-wise execution allows a developer to examine the state of a program during execution. Step-

wise execution allows the developer to examine a section of a program more closely during execution.

Step-wise execution tools include three basic step types: *stepOver*, *stepIn*to, and *stepOut* [9]. *StepOver* executes a single unit. The unit stepped over may be a composite unit, such as a rule composed of several rules. *StepOver* enables developers to maintain focus within a certain scope. *StepInto* executes the next unit, moving into a composite unit if necessary. *StepOut* moves execution to the first unit in the containing scope, executing remaining units in the current scope. Step-wise execution also supports three other basic commands; *play*, *pause*, and *stop*.

2.2 Omniscient Debugging

Key challenges for omniscient debugging are scalability and performance. Several potential solutions have been presented for these concerns.

Utilize garbage collection, similar to the facilities available in modern GPLs such as Java [1–3]. Several solutions incorporate facilities similar to a garbage collector, removing elements that are no longer referenced. This approach attempts to minimize data collected over time, but in some scenarios these elements may need to be regenerated, thus reducing run-time efficiency.

Lewis proposed limited navigation through execution history [3], providing a window effect. The advantages and disadvantages of this solution are similar to utilizing garbage collection, but where garbage collection maintains some history of elements, the window solution removes all information outside of the current window.

Lewis also proposed identifying a subset of the program as being of interest to the debugging process [3] and only record information concerning these elements. Elements of interest may be selected before playback begins or elements no longer of interest during run-time may be selected dynamically [8]. This approach requires knowledge of which elements will be of interest. Several analysis techniques could be used to inform this decision (*e.g.,* impact analysis or program slicing), but the potential exists for both false positives and false negatives is still present.

Supporting omniscient debugging for model transformations encounters challenges not present in GPLs. Omniscient debuggers typically support imperative languages. However, MTLs are typically either declarative or a hybrid of declarative elements with limited imperative structuring. Several MTLs exist that support an imperative approach (*e.g.,* QVT-Operational [13]), but we focus on declarative and hybrid approaches as supported in AToMPM. These languages support (as discussed in Section 1) nondeterministic scheduling, but most significantly hybrid and declarative languages do not explain the operations taken. A typical format for declarative MTLs is graph grammars where a starting slice of the model is specified as a graph pattern and the result of applying the rule is specified as another graph pattern. The relation between the two graph patterns defines the operation. AToMPM supports T-Core [5] and MoTif [4]. Also, MDE tools often provide a graphical environment (*e.g.,* , AToMPM) as opposed to the traditionally textual environments of GPLs.

2.3 AToMPM

Our prototype is implemented within the context of AToMPM which is a cloud-based modeling solution with an associated graphical, browser-based user interface. The backend structure of AToMPM is intended to provide a scalable solution to modern modeling concerns. The current release version of AToMPM was demmonstrated at MoDELS 2013 [6]. AToMPM provides two basic transformation languages: MoTif and T-Core. MoTif provides basic support for rule scheduling and control flow with graph transformation rules defining the primitive operations. As discussed by Syriani et al., T-Core provides a set of primitives derived from studying existing MTLs [5].

3. Omniscient Debugging for Model Transformations

Omniscient debugging enables developers to reverse the execution of a system. It can be considered a logical opposite of step-wise execution. The goal is to provide free exploration of the execution history at runtime. We created a debugger for AToMPM [6] (as seen in Figure 1) that 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*.

3.1 Collecting a History of Execution

We collect a history of execution to enable traversal without re-executing rules. The history is composed of a sequence of steps, each step corresponds to a single transformation rule. A step stores a reference to the rule, a reversible set of all changes applied by the rule, and any other necessary transformation information (*e.g.,* TCore maintains a packet that gets passed to each subsequent rule invocation [5]). A change stores the element modified and the associated values. Changes are reversible, and can be used to either undo or redo a modification. Thus, we can use the collection of changes in a step to undo or redo that step. As discussed in Section 3.2, we can use a select set of changes from these steps to undo or redo a set of steps more efficiently.

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

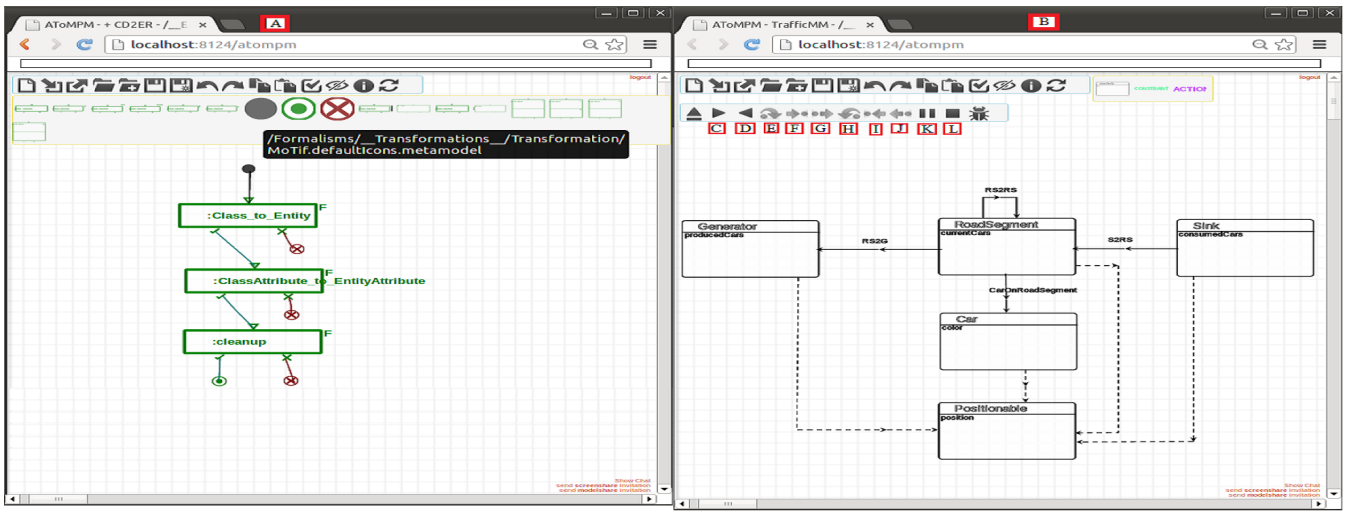

Figure 1. Screenshot of Debugging Session in AToMPM (A) MT being executed (B) Model being executed (C) *play* (D) *playBack* (E) *stepOver* (F) *stepIn* (G) *stepOut* (H) *backOver* (I) *backIn* (J) *backOut* (K) *pause* (L) *stop*

a space complexity upper bound of $O(n * (A + m * B))$. A is a constant refering to the transformation state information. B is a minimal set of information regarding the change. 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 have a large number of steps, the structure performs poorly. However, because trace information is required for full traversal, any omniscient debugger will encounter similar scaling concerns. The alternative to storing trace information would require converting unidirectional rules into bidirectional rules. This conversion is not always possible. A counterexample is deleting a node since any information in the deleted node is lost once deleted. Other scenarios may exist where reverting the result of a transformation is ambiguous.

Despite storing minimal information, in extreme circumstances the full history will exceed the bounds of memory. For this reason, history maintains a window of active history. As mentioned in Section 2, this technique has been explored previously by Lewis [3]. 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. The developer may alter the window size. Thus, the developer can control the upper bound of memory consumption without losing the ability to traverse the full execution history.

3.2 Traversing a History of Execution

The various control features all serve the purpose of traversing history (and adding to history for step-wise execution features). However, traversing through history can be made more efficient by using the fact that all changes have been recorded. Our technique utilizes the execution history create macro steps that avoid unnecessary CRUD operations. A macro step is a step that contains changes from potentially many rules. Macro steps are used to transition from one step to another without executing all of the changes in the intervening steps. For example, if an element is altered numerous times, then only the most recent update needs to be applied and all other updates can be ignored. We ignore these updates safely because our history stores the resulting value rather than computed differences. Both *backOut* and *backOver* are obvious targets for this technique, but *stepOut* and *stepOver* utilize the technique when replaying previously executed portions of the transformation. The number of steps traversed by these features is related to the size of the scope being traversed. Because a scope can contain many steps, the potential for iterating over an indeterminately large number of changes for the same elements is present.

In order to construct a macro step efficiently, with regard to runtime complexity, a structure is needed that provides efficient access when there is a large number of changes in the set of steps being traversed. To accomplish this task, we store a revisions cache for each element recording each step where that element was altered (created, updated, or deleted). The cache stores the position for a particular change, *i.e.,* the step and the change's location within the step. Using the location, the change can be accessed in constant 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 check each element within the model to determine if a change must be applied for the element and where the change is located. This

```
1  def buildMacroStep(currentStep, targetStep, allModelElements):
2    # Return a macroStep which will revert or advance the system to the targetStep
3    macroStep = Step()
4    for element in allModelElements:
5      if(currentStep < targetStep):
6          stepPos,changePos = revisionsCache[element].before(targetStep)
7      else:
8          stepPos,changePos = revisionsCache[element].after(targetStep)
9      mostLocalChange = history[stepPos][changePos]
10     stepPos, changePos = revisionsCache[element].mostRecent(targetStep)
11     mostRecentChange = history[stepPos][changePos]
12     if((mostRecentChange.getType()=="DELETE")and(mostRecentChange!=mostLocalChange)):
13         macroStep.add(mostRecentChange)
14     macroStep.add(mostLocalChange)
15   return macroStep
```

Listing 1. Macro Step Building Algorithm

algorithm has $O(n * lg(m))$ runtime complexity, where n is the number of elements in the model that have been altered (we only store a cache for elements that have been altered) 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.

As illustrated in line 4 of Listing 1, we must iterate over all model elements that have been altered, including those no longer present. We can store these as a collection of all model element references. The structure must then be iterated over and we perform a read in the revisions cache. For each model element, the revisions cache stores an associated array of change locations (within history). The array associated with each element contains an entry for each step where an element was altered. The entry for a given step contains a reference to the most recent step that altered the element. For example, if a model element is updated in step 1 and 3. The array structure would contain 0, 1, 3. The 0 references the initial step which precedes the execution of the model transformation. A binary search on this array finds the entry that is closest to and either before or after a given step.

The algorithm, provided in Listing 1, assumes that building the macro step by 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. This is correct if there are a large number of changes in the steps or a large number of steps, but in some scenarios there is a relatively small number of steps and the steps contain a relatively small number of changes. Because we must check every change if we iterate over the steps, we define the switch between iterating over the steps and using the lookup table at the point where the sum total number of changes contained in the steps being traversed is greater than the number of elements in the model. Even if an element is changed many times (*e.g.,* in the case when we would be checking changes that would not be included in the resulting set), we must perform a check to ensure the change is not required. Thus, we can ensure an upper bound on our runtime complexity by checking every element of the model as described in the algorithm previously. However, in some scenarios we could achieve a potentially significant speedup by only checking the changes contained in the steps. In order to achieve this speedup and maintain our current runtime complexity upper bound, we must be able to check the number of changes that would need to be traversed (we assume the number of elements in the model can be known) without affecting the overall runtime complexity. Our technique adds to each step a count of the total number of changes since the transformation began. Using this value we can modify our algorithm to test the number of changes that must be iterated, and either iterate over the steps or check every element. Thus, we can check the number of changes in constant time. The modified algorithm is provided in Listing 2. There is one additional complexity, if an element has been deleted it must be recreated before being updated with the relevant change. We can perform a constant time lookup to find the most recent change and if it is a delete then we add it to the macro step.

4. Quantifying the Impact of Omniscient Debugging

Our prototype provides a proof of concept implementation of the omniscient debugging features described in Section 3. The impact with regards to performance and scalability associated with incorporating the omniscient features has been reported in a previous work [11], the raw data has been made available online[1]. The prototype and a step-wise debugger

[1] `http://corle001.students.cs.ua.edu/files/omniscientData.zip`

```
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

implemented in AToMPM were both used to run a Petri-net simulator over various size models and the performance metrics were recorded. A table summarizing the raw data available online from the previous study is included in Figure 2. These results illustrate the average time required to perform a step in the step-wise implementation and the omniscient implementation were approximately equal when executing a rule for the first time. The average time recorded for the omniscient implementation was actually slightly more efficient. The results also show that re-executing a rule (stepping through history) took on the order of milliseconds. In Figure 2, back represents re-executing a rule; stepwise find and stepwise update represent executing two distinct transformation rules in the stepwise implementation; and omniscient find and omniscient update represent executing two distinct transformation rules in the omniscient implementation.

	10	**100**	**500**	**1000**
BACK	0.5	0.18	0.31	0.11
OMNISCIENT FIND	7.29	181.65	3083.67	14001.26
STEP-WISE FIND	14.7	511.48	3000.12	13824.51
OMNISCIENT UPDATE	3.57	19.81	348.47	1648.12
STEP-WISE UPDATE	3.1	19.57	345.47	1628.93

Figure 2. Summary of Average Results Comparing Stepwise Execution and Omniscient Debugging Features

We have performed a modified replication of the previous experiment to illustrate the performance impact of the macro step building algorithm. We ran the same Petri-net simulator with the same input models as previously. However, we used the omniscient debugger with the macro step algorithm and the omniscient debugger without the macro step algorithm as the two tools to compare. It is worth noting, in the previous experiment the network transmission time was removed when considering time to execute as the network communication overhead was constant for all step operations. In the results of our modified replication we have included the network communication overhead as the *backOver* function

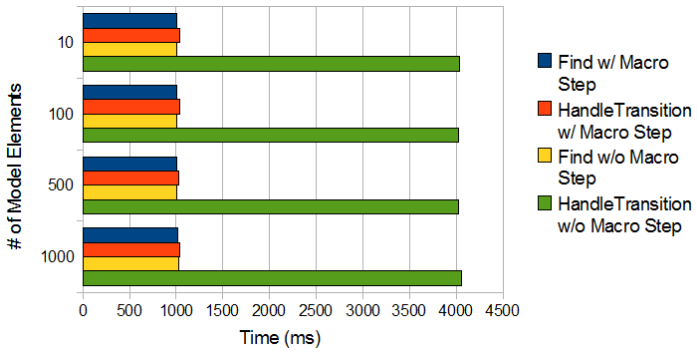

Figure 3. Comparing Omniscient with and without Macro Step Building Algorithm

was used which causes multiple sends when multiple steps are traversed. A summary of our results can be found in Figure 3. All operations recorded in Figure 3 are the *backOver* step feature which executes exclusively in history (*i.e.,* re-executing a rule) and therefore would compare with the back entry in Figure 2. 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 reduces the network communication overhead. In other tools where the network communi-

cation is not a concern, the macro step algoritm should still prove superior in the case where redundant changes can be eliminated (as discussed in Section 3.2) and will perform the same in the worst case where no redundant changes occur.

Finally, we performed a basic analysis of the average size for the constants discussed in Section 3.1. We collected the information at the end of the run over our largest model. The constant A, the additional information stored per step other than the list of changes, was on average 152.99 bytes. The constant B, the information stored per change, was on average 406.87 bytes. To give these numbers some context, we also recorded the size required for a set of basic elements in python (language implementing AToMPM's transformation engine). An integer requires 12 bytes. A string with one character requires 22 bytes with each additional character requiring 1 more byte. An object with no fields requires 36 bytes. Other languages could significantly reduce the memory overhead. For example, C uses 2-4 bytes per integer. Therefore moving to a language such as C could reduce the memory consumption by a factor of 10. We are also planning as part of our future work to investigate a common range for the number of changes per step and number of steps in an industrial scale transformation to further evaluate the memory usage of the technique.

5. Conclusions

This paper discusses an omniscient debugging technique for MTs and a proof of concept prototype applying the technique. To our 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 run-time. 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. We discuss an omniscient debugging technique 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 the technique through a prototype. Finally, the prototype was evaluated for scalability, in terms of memory usage, and performance, in terms of time to execute.

As future work, we plan to replicate the performance and scalability experiment with a larger scale and variety of models and model transformations which will enable more rigorous analysis. We limit ourselves to a simple Petri-net simulator for our initial analysis, but we ackowledge that industrial models will possess a large variety of complexity than can't be explored with any single transformation. We are also planning a user study to explore how developers use the provided omniscient features and the impact of the technique on the effectiveness of developers.

References

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

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

[3] Bill Lewis. Debugging backwards in time. In *Proc. of the Fifth Int'l Workshop on Automated Debugging*, Ghent, Belgium, 2003.

[4] Eugene Syriani and Hans Vangheluwe. A modular timed model transformation language. *Journal on Software and Systems Modeling*, 12(2):387–414, jun 2011.

[5] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-Core: A Framework for Custom-built Model Transformation Engines. *Journal on Software and Systems Modeling*, 13(2), jul 2013.

[6] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, Miami FL, USA, 2013. CEUR-WS.org.

[7] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: A qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[8] Guillaume Pothier and Èric Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85, nov 2009.

[9] Hans Vangheluwe and Raphiel Mannadiar. Debugging in domain-specific modelling. In *Proc. of 3rd Int'l Conf. on Software Language Engineering*, pages 272–285, Eindhoven, The Netherlands, 2010.

[10] 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*, pages 5–20, Denver, CO, USA, 2009.

[11] Jonathan Corley. Exploring omniscient debugging for model transformations (under review). In *Demos/Posters/StudentResearch@ MoDELS*, 2014.

[12] Mirko Seifert and Stefan Katscher. Debugging triple graph grammar-based model transformations. In *Proceedings of 6th International Fujaba Days*, Dresden, Germany, 2008.

[13] Raphael Romeikat, Stephan Roser, Pascal Mllender, and Bernhard Bauer. Translation of qvt relations into qvt operational mappings. In *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin Heidelberg, 2008.