Software Qual J (2017) 25:7–48
DOI 10.1007/s11219-015-9304-4

Efficient and scalable omniscient debugging for model transformations

Jonathan Corley[1] · Brian P. Eddy[2] · Eugene Syriani[3] · Jeff Gray[1]

Published online: 18 January 2016

Abstract This paper discusses a technique for supporting omniscient debugging for model transformations, which are used to define core operations on software and system models. Similar to software systems developed using general-purpose languages, model transformations are also subject to human error and may possess defects. Existing model-driven engineering tools provide stepwise execution to aid developers in locating and removing defects. In this paper, we describe our investigation into a technique and associated algorithms that support omniscient debugging features for model transformations. Omniscient debugging enables enhanced navigation and exploration features during a debugging session beyond those possible in a strictly stepwise execution environment. Finally, the execution time performance is comparatively evaluated against stepwise execution, and the scalability (in terms of memory usage) is empirically investigated.

Keywords Omniscient debugging · Model-driven engineering · Model transformation · Empirical evaluation

✉ Jonathan Corley
corle001@crimson.ua.edu

Brian P. Eddy
beddy@uwf.edu

Eugene Syriani
syriani@iro.umontreal.ca

Jeff Gray
gray@cs.ua.edu

[1] Department of Computer Science, The University of Alabama, Tuscaloosa, AL, USA

[2] Department of Computer Science, University of West Florida, Pensacola, FL, USA

[3] Department of Computer Science, University of Montreal, Montreal, Canada

1 Introduction

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 (Combemale et al. 2014). MDE approaches often use customized domain-specific modeling languages that capture the intent of a particular group of end users through abstractions and notations that fit a specific domain of interest (Gray et al.2007). Thus, the domain-specific abstractions and notations aid in eliminating the accidental complexities of implementation. In MDE, the evolution, simulation, generation, and translation of models are commonly defined using model transformation languages (MTLs), which can be used to specify the distinct needs of a requirement or engineering change at the software modeling level (Lúcio et al. 2016). Model transformations are also a type of software abstraction that can be subject to human error, and traditional approaches to bug localization have also been applied to assist in locating errors in model transformations (Schönböck et al. 2009). Despite the focus on models and model transformations, traditional development concerns such as debugging must still be undertaken by developers adopting MDE practices. Bran Selic (2003) commented that if developers are not satisfied with the day-to-day application of MDE, then MDE will be rejected in practice.

Debugging is a fundamental software engineering task. However, despite the common need for debugging in software development, tool support for debugging has changed little over the past half century (Seifert and Katscher 2008). Several novel approaches to debugging have been introduced for general-purpose languages (GPLs), such as omniscient debugging (Engblom 2012). However, stepwise execution is the most common debugging technique provided in both GPL tools (e.g., Eclipse and Visual Studio) and MDE tools [e.g., ATL (Jouault et al. 2008)]. Stepwise execution enables developers to control the execution of the system and view normally hidden state information through a set of execution traversal features enabling continuous execution, pausing or stopping execution, and traversing execution in a stepwise manner. The only modeling tool we are aware of that includes an advanced dynamic debugging technique for model transformation is TROPIC (Schönböck et al. 2009), 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 a 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 (informally referred to as the bug) that caused the failure. These three terms (error, failure, and fault) each receive a distinct definition in the IEEE 610.12-1990 standard glossary of software engineering terminology (IEEE 2002). 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, other techniques such as model slicing (Ujhelyi et al. 2012; Androutsopoulos et al.2013) and query-based debugging (Schönböck et al. 2009) have been explored in the context of models and transformations that could also aid in identifying similar issues. 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

debugging by aiding the developer in selecting points of interest in the execution history to explore via omniscient traversal.

Existing literature for omniscient debugging focuses on GPLs (e.g., Java and C++). However, model transformations (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 non-trivial 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 non-deterministic rule scheduling. The non-determinism 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 may be observed.

As we consider support for omniscient debugging of MTs, we must also consider the need for an efficient and scalable solution. The organizers of the scalability in model-driven engineering workshop state that current modeling and MT environments are being pushed to the limits of the capability, and further research and development is imperative (Di Ruscio et al. 2013). Numerous works have been presented in recent years discussing topics such as parallel processing of MTs (Burgueño et al. 2015), techniques supporting incremental processing of model transformations (Szárnyas et al. 2014), and cloud-based architectures for modeling and transformation (Szárnyas et al. 2014; Basciani et al. 2014; Corley et al. 2016). Therefore, we have undertaken to utilize a minimal structure to store required information in support of omniscient debugging, and we have developed a set of algorithms designed to support efficient omniscient traversal of MTs. As models continue to grow in size, Kolovos et al. (2013) define large-scale models as being on the order of millions of elements, the transformations and supporting transformation tooling operating on such large models must be designed for efficiency and scalability.

The major contributions of this paper focus on providing and evaluating an efficient and scalable technique supporting basic omniscient debugging features for model transformations. The paper's contributions can be summarized as:

- We define basic omniscient debugging features and extend traditional stepwise execution features to support omniscient traversal (i.e., both executing a transformation forward and reverting it back).
- We define a minimal structure to store history for a MT engine.
- We discuss how the minimal history structure can be used to provide efficient omniscient traversal.
- Finally, we provide an empirical evaluation of an implementation of our technique as compared to a traditional stepwise execution debugger within the same context. The empirical evaluation includes evaluating execution time variance, memory consumption, and re-executing a transformation as opposed to using omniscient traversal.

The remainder of the paper is structured as follows. Section 2 will overview related work in the area of omniscient debugging. Section 3 will present an illustrative scenario of a developer using omniscient debugging. Section 4 will describe our technique enabling

omniscient debugging for model transformations, including theoretical analysis of execution time and space complexity bounds. Section 6 will discuss the design and results of an empirical analysis of the performance and scaling of our technique. Finally, Sect. 7 will provide concluding remarks and briefly discuss our ongoing research efforts in the area of omniscient debugging for model transformations.

2 Background and related work

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. However, before discussing debugging, we first introduce basic concepts and terminology concerning MT and highlight differences between MTs and GPLs.

2.1 Models and model transformations

Just as high-level languages provide abstractions to reduce accidental complexity from low-level languages, MDE seeks to shift the focus of developers away from the solution domain, and any accidental complexity inherent to the solution domain, to bring development closer to the problem domain (Schönböck 2012). This goal is achieved in MDE by focusing on models. Models conform to a given metamodel. A metamodel is a model that formally defines the structure and static semantics of a set of models (Kühne 2006).

MTs are core operations that drive evolution and maintenance within the MDE paradigm. A MT converts source model(s) to target model(s) by following a set of rules. The rules may be purely imperative, such as in QVT-O (QVT 2015); purely declarative, such as in a graph transformation (Czarnecki and Helsen 2006); or combine imperative and declarative elements, such as in ATL (Jouault et al. 2008). An imperative language functions similarly to traditional GPLs (e.g., Java or C++) by having a structured and rigid control flow scheme. In an imperative approach, the conversion process is defined explicitly, similar to a GPL (QVT 2015).

Declarative approaches do not express *how* a transformation is implemented, but rather focus on *what* should occur during the transformation. A common declarative approach is graph transformation, which includes a left-hand side (LHS), a model pattern that is matched to a subset of the source model(s) to be transformed, and a right-hand side (RHS), a model pattern that is matched to a portion of the target model(s) to be created or updated (Czarnecki and Helsen 2006). The combination of LHS and RHS rules produces pre- and post-condition definitions of what should occur during the transformation, but the details of how are not specified. Additionally, some graph transformation environments enable the use of one or more negative application conditions (NAC). A NAC specifies a constraint which, if true, prevents execution. Figure 1 presents a sample graph transformation rule with a NAC. The LHS specifies two nodes (labeled 1 and 2) connected by a link. The rule will only be applied if the LHS is properly matched, but may be applied to any element set within the model that matches the LHS. The RHS then adds a new node (labeled 3) which is linked to both of the existing nodes (1 and 2). However, the NAC may prevent executing the rule if the two existing nodes (1 and 2) are already

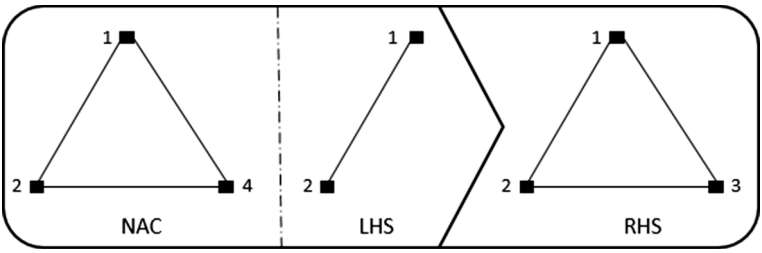

Fig. 1 Sample graph transformation rule

connected to another node (labeled 4). Thus, the rule will only be applied if two nodes are connected by a link and the two nodes are not already connected to a shared third node. Furthermore, graph transformation rules may be used in hybrid approaches (e.g., ATL or MoTif (Syriani and Vangheluwe 2011)) that focus on defining declarative rules in combination with a mechanism to schedule and combine rules. In MoTif, the order of rule execution is defined by a structure resembling an activity diagram. Each rule may potentially call numerous graph transformation rules, and the selection and order of graph transformation rules are potentially non-deterministic.

2.2 Non-determinism in model transformations

Non-determinism is a key feature of hybrid and declarative MTLs, but not found commonly in GPLs. Scheduling of rules (i.e., how the order of rule application is decided at runtime) may be non-deterministic (Czarnecki and Helsen 2006). Non-deterministic rule scheduling depends on the rules being defined in a way that prevents variations in order from providing incorrect results. However, in practice it is possible to define transformations such that the order of rules can produce incorrectly varying results. In this scenario, a traditional debugger that relies on restarting the transformation to revisit a past state suffers from more than needing to re-execute the transformation. If the error is due to a specific ordering of rule application, then non-deterministic rule scheduling prevents a stepwise execution debugger from guaranteeing the ability to revisit an observed result. This can be further expanded to situations where the rule order is not a factor in the error, but presents a variation in the processing of intermediate events that may complicate the process of bug localization. MTLs (particularly those using graph transformation rules) also support non-deterministic selection of model elements when executing a transformation rule (Czarnecki and Helsen 2006). The LHS of a graph transformation may match many distinct sets of elements, but the order in which these sets are chosen is often not deterministic. Thus, a developer may re-execute a system to find the elements are changed in a new way. In fact, the bug may even be due to the choice of elements made by this non-deterministic system. Thus, while non-deterministic systems are central to many MTLs, the use of non-determinism presents an obstacle to bug localization that is not adequately managed by stepwise execution debuggers. However, an omniscient debugger provides an ideal solution to this concern. The non-deterministic behavior is captured by an omniscient debugger. Thus, within a given debugging session, the developer can revisit past states exactly as they occurred during initial execution. This enables developers to track an error to the initial failure without concern for the non-deterministic decisions made by the underlying execution engine.

2.3 Bidirectional transformations

An interesting feature of MT is the direction of the transformation flow. Typically, transformations proceed from source(s) to target(s), but transformations may also be bidirectional to accommodate translation both from source(s) to target(s) and target(s) to source(s) (Stevens 2010). Bidirectional transformation rules provide an alternative to storing trace information. For systems with bidirectional transformations, the assumption that the bidirectional nature of the transformation is implemented properly may not hold unless a strict bijective approach (which requires a single reversible operator) is used, and a strict bijective approach is not always possible (Stevens 2010). Additionally, transformation rules are not always defined as bidirectional. Thus, we would require deriving an inverse for each rule. However, an inverse rule is not always possible. Consider a model transformation rule that deletes a model element. Since any information in the deleted element is lost once deleted, an inverse rule would not be possible. Similarly, updates can be ambiguous; e.g., setting an element to a specific value does not provide any clue to the prior value. More complex scenarios also exist. An element might be updated based on the value of a second element (e.g., `e1.value += e2.value`). If the second element is then deleted, the inverse rule is no longer applicable because the value of `e2` has been lost. Thus, a key goal of our omniscient technique is to store a minimal complete trace of execution history that will manage the ambiguities not possible with a bidirectional transformation solution.

2.4 Stepwise execution

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 (AToM3 2015) allows developers to manually control rule scheduling, which would normally be scheduled using a non-deterministic method). A stepwise execution environment generally possesses the following features: `play`, `pause`, `stop`, and step features. `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. Three step features (`stepOver`, `stepIn`, and `stepOut`) allow developers to incrementally progress the execution environment in distinct ways. Numerous MDE tools (e.g., TROPIC (Schönböck et al. 2009), GReAT (Agrawal et al. 2006), ATL (Jouault and Kurtev 2006), TefKat (Steel and Lawley 2004), AToM3 (AToM3 2015), VIATRA2 (Varró and Balogh 2007), AGG (Taentzer 2003), and Fujaba (Henkler et al. 2010)) provide basic debugging support in the form of stepwise execution facilities.

Mannadiar and Vangheluwe introduced a variety of basic techniques to aid debugging domain-specific models including MTs (Mannadiar and Vangheluwe 2011). Their contribution includes using language features such as stack traces, exceptions, assertions, as well as the more formal technique of stepwise execution.

2.5 Omniscient debugging for GPLs

Omniscient debugging is not a new technique in the realm of GPLs. Zelkowitz published on the concept of reversible execution in the early 1970s (Zelkowitz 1973). Since this time, significant work has been undertaken in the context of GPLs including several commercial products (Engblom 2012). However, these techniques have focused on either utilizing low-level machine implementations to support reverse and replay or utilizing traces designed to capture information for a given GPL. In this work, we define a trace-based omniscient debugger supporting hybrid MTLs. The debugger is defined at the level of the transformation engine. Thus, we support omniscient debugging at the level of CRUD (create, read, update, and delete) operations in a modeling environment. The term originated from the database community, but has been adopted by the modeling community to define the most atomic set of operations on a model.

Omniscient debugging can be viewed as 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. and Lewis discussed a strategy similar to garbage collection (Lewis 2003; Lienhard et al. 2008, 2009) removing any elements from history that are no longer referenced. This technique seeks to minimize data collected over time, but in some scenarios these elements may need to be regenerated, thus reducing execution time efficiency. Lewis discussed limiting the portion of history that can be navigated (Lewis 2003), 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 (Lewis (2003) 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 (2009) also explored a dynamic variant (e.g., select elements no longer of interest during runtime). This technique 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.

2.6 Omniscient debugging for MDE

Recently, there has been some work in the area of MDE toward the application of omniscient debugging. Van Mierlo presented a proposal toward the debugging of executable models defining simulation semantics (Van Mierlo 2014). A particular focus of the work addressed handling simulated real time. The scope of our new contribution of this paper concerns applying omniscient debugging to MTs. Our work does not concern handling simulated real time or relating the model entities with generated code. Our prior work has investigated applying omniscient debugging to model transformations toward a scalable and performant omniscient technique (Corley 2014; Corley et al. 2014). Furthermore, in collaboration with researchers from IRISA/INRIA and University of Rennes, we have explored applying omniscient debugging to an executable domain-specific modeling (xDSML) environment (Bousse et al. 2015). The collaborative work utilized generated domain-specific trace metamodels along with a generated domain-specific trace manager to enable developers to utilize a generic omniscient debugger implementation with xDSMLs.

This collaborative work also investigated multi-dimensional omniscient debugging traversal features. These features enable a user to explore history through only steps relevant to a given model element. Thus, the developer can minimize the time spent reviewing steps not of interest. In this paper, we focus on evaluating the changes to the underlying model transformation engine to support omniscient debugging. Because the multi-dimensional features are defined using a subset of omniscient debugging features (primarily `jump`), we do not include those features in this paper. We are not aware of any other literature concerning omniscient debugging in the context of MDE.

2.7 AToMPM

Our omniscient debugger prototype is implemented within the context of AToMPM (Syriani et al. 2013), which is a cloud-based modeling solution with an associated graphical, browser-based user interface. The back-end structure of AToMPM is intended to provide a scalable solution to modern modeling concerns. 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 (Syriani et al. 2015).

3 An omniscient debugging scenario

This section describes an illustrative scenario of a developer using our technique to locate a defect in a MT. For this scenario, we will describe the efforts of a developer (who we will refer to as James) attempting to find a defect using omniscient features. For the purposes of this illustrative scenario, we use a model transformation solution for the 2014 Transformation Tool Contest (TTC) Movie Database Case (Movie DB Case) (Horn et al. 2014) originally presented in (Ergin and Syriani 2014). The transformation pairs actors with movies and records the ratings for those movies in which they appear. The main task of the model transformation is to identify all actor couples that appear in at least three movies together and to compute the average rating of those movies. This task can be broken into three subtasks: generating the data, identifying couples, and averaging the ratings of the movies. We will focus on the second subtask, identifying couples. The solution was developed in MoTif (Syriani and Vangheluwe 2011) and executed in AToMPM (Syriani et al. 2013). For the sake of simplicity, this scenario focuses primarily on a specific rule, but the transformation contains many rules. See Horn et al. (2014) for the full details of this transformation, and (Ergin and Syriani 2014) for a solution created in AToMPM. Here, focusing on a single transformation rule can be likened to focusing on a single method of a Java program.

3.1 Transformation details

The transformation presented in Fig. 2 identifies pairs of actors/actresses who have at least three movies in common, links the two actors/actresses to a shared couple node, and then links the couple node to each movie shared by the two actors/actresses comprising the couple. The transformation contains two graph transformation rules, `findStarsAndCreateCouple` and `referenceToCoupleMovies`. The first, `findStarsAndCreateCouple`,

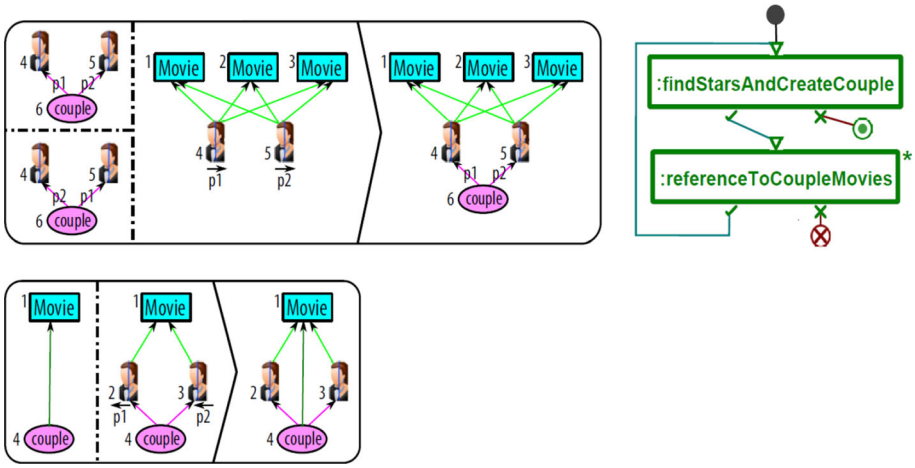

Fig. 2 Solution to Task 2 of the 2014 TTC Movie DB Case as presented by Ergin and Syriani (2014)

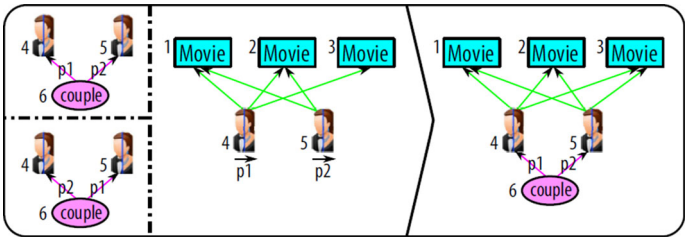

Fig. 3 Defective variant of `findStarsAndCreateCouple`

identifies a pair of actors/actresses which are both linked to at least three movies with each other and creates a new couple node that is connected to both actors/actresses. The first rule also uses two NACs to ensure the two actors/actresses are not already attached to a shared couple node. The second, `referenceToCoupleMovies`, identifies a couple transitively linked to a movie through both actors/actresses comprising the couple, and then `referenceToCoupleMovies` links the couple directly to the movie. A NAC ensures that couples are not linked to the same movie more than once. Finally, a MoTif transformation seen to the right of `findStarsAndCreateCouple` defines the control flow for the transformation. The transformation starts by executing `findStarsAndCreateCouple`. After creating a new couple, the transformation repeatedly executes `referenceToCoupleMovies` to link the new couple to each movie linked to both actors/actresses comprising the couple. The transformation exits successfully if `findStarsAndCreateCouple` fails, indicating no further actor/actress couples exist. The transformation exits with failure status if `referenceToCoupleMovies` fails to apply at least once.

Consider the scenario where our developer must fix a defective implementation of `findStarsAndCreateCouple`, as presented in Fig. 3. The defective rule may identify a pair of actors/actresses that only share two movies. The defective rule then creates a new link such that the pair of actors now appears to be linked to three movies. Alternatively, the

defective rule might identify a correct couple, but then link one of the actors/actresses to a new movie. After the rule has been executed, the model appears to be in a correct state. However, the model has been subtly corrupted and the transformation will not produce correct results.

3.2 An omniscient debugging scenario

Our developer, James, might execute a set of test cases where the resulting couples and couple averages (average of all shared movie ratings for a given couple) are known. In the process of executing the tests, James notices that in a specific test case the average for a certain couple has been computed incorrectly. He executes the transformation using an omniscient debugger and initially traverses to Task 3 to observe the couple's average being computed. After stepping through and locating the step which computed the couple's average, James notices that the couple's average has been correctly computed based on the existing links. Further investigation identifies that the couple has been computed with an additional movie incorrectly included. He jumps back to Task 1 to observe the actors, movies, and linking edges being generated. However, he immediately finds that the two actors have been correctly generated with the expected movie links. James now continues re-executing the system to see the couple being created and the movies linked to the couple. He then steps through the defective `findStarsAndCreateCouple` rule. From this navigation sequence, he is able to observe that the rule has incorrectly matched a movie only connected to a single actor, and he can even back up the system and re-execute to confirm his initial observation. He observes the extra link being created and understands that this rule is the defect causing the error. James investigates the rule definition, identifies the missing edge, and is able to correct the defect. Further testing verifies the change is correct and the issue is resolved.

In the scenario, James is able to freely traverse the execution history of the system enabling him to directly follow the trail of clues to eventually identify the defective rule. When James first observes the defective behavior, he is even able to immediately revert and re-execute the rule to verify the defective behavior. James uses the jump feature to quickly move through the system's execution history to an interesting point (where the actors, movies, and linking edges are created). The omniscient features enabled a simple, intuitive exploration of the system's execution to identify the defect. James made use of numerous basic features (e.g., jump, back, step) to explore the system. If James had been using a stepwise debugger, he would have needed to restart the system at least twice. The first time, he would need to restart when moving from Task 3 (where the couple average is computed) to Task 1 (where the actors, movies, and linking edges are generated). The second time, he would restart when he re-executed the defective rule to verify the rule was producing incorrect results.

In this scenario, re-executing costs time. Depending on the transformation, the time to re-execute can be significant. We have observed rules involving complex searches of large models that can take 5–10 min. In particular, a transformation that generates a model(s) (e.g., generating test cases for mutation testing) or simulates a complex model(s) (e.g., a physics simulation of the interaction of stellar objects) can take significant time to execute. Additionally, the size of input for a transformation directly impacts the execution time (especially concerning rules that must search the model). Kolovos et al. (2013) describe large models as containing on the order of millions of elements. However, in some cases, re-executing may also cause the developer to lose the context where the defect occurs. The defect may be lost because some defects do not appear in all executions

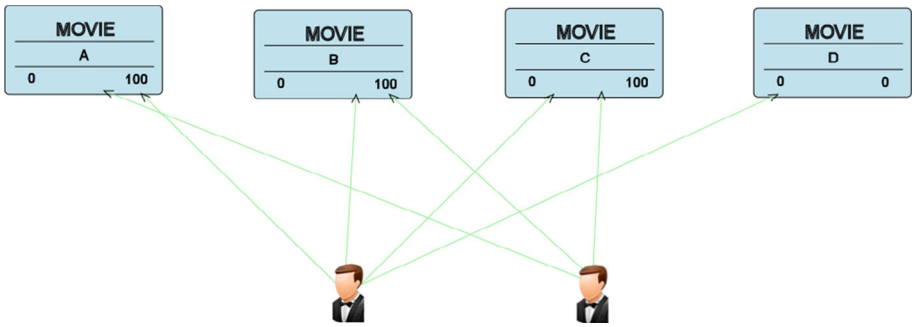

Fig. 4 Sample model for Movie DB Case

given the same input conditions. This is due to the non-deterministic behavior of MTs as discussed in Sect. 2.2. Consider the sample model presented in Fig. 4. When processing this sample model, the transformation should couple the two actors and provide a couple average of 100. However, due to the non-deterministic selection of model elements when multiple matches are present for a rule's LHS, the defective rule may match either movies A, B, and C or some triple containing movie D. In the case where the rule matches movies A, B, and C, the result will be calculated correctly, but if the movie D is matched the result will be calculated incorrectly to be 75. This is because the defective rule would create a link between both the right actor and movie D, increasing the number of movies associated with the couple, and decreasing the couple's average movie score. Omniscient debugging preserves the context in which the defect occurs (such as connecting the couple to movies A, B, and D in Fig. 3). Thus, James may fully explore the context where the defect is presented initially, and avoid applying to a different set of elements which may not present the defective behavior. Thus, the omniscient debugger prevents James from re-executing the system, which may result in not identifying the observed error (due to a different set of elements being selected by the rule).

4 Omniscient debugging for model transformations

Omniscient debugging is a natural extension of stepwise execution that enables reverse execution. The AToMPM Omniscient Debugger (AODB) was developed as a prototype omniscient debugger within AToMPM (Corley 2014; Corley et al. 2014). The AODB prototype continues evolving to better support omniscient debugging for MTs. AODB implements our technique to support omniscient debugging for MTs. Our technique provides the common features of stepwise execution (i.e., `play`, `pause`, `stepIn`, `stepOut`, `stepOver`, and `stop`) (Corley et al. 2014). The stepwise features have been modified to leverage an execution trace history supporting omniscient traversal that avoids the need to re-execute transformation rules in many cases. 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 technique also provides a set of features that mimic stepwise execution, but revert execution. The omniscient features are `playBack`, `backIn`, `backOver`, and `backOut`. In this section, we first define the supported traversal features (both stepwise execution and

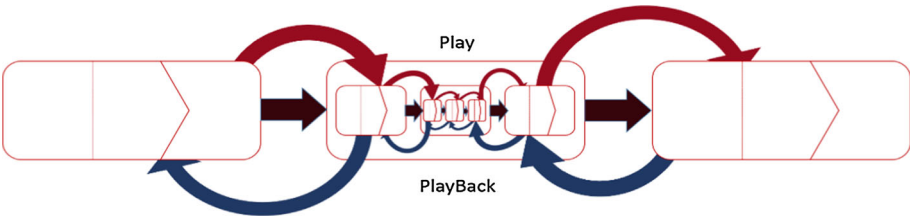

Fig. 5 Stepwise and omniscient continuous execution features

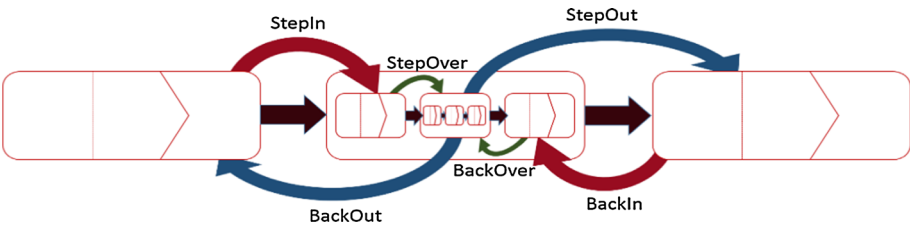

Fig. 6 Stepwise and omniscient step features

omniscient) as illustrated in Figs. 5 and 6 and then discuss how history is collected and stored. This section also presents an algorithm for more efficient traversal of history.

4.1 Execution traversal features for omniscient debugging

The following are the definitions that we will use for standard stepwise execution traversal features that are found in a typical debugging environment:

- `Play`: Continuously execute the system.
- `Pause`: Suspend execution until restarted using another traversal feature.
- `Stop`: Cease execution and clear any intermediate data stored during execution.
- `StepIn`: Execute a single atomic step of the system, entering into any contained scopes.
- `StepOut`: Execute the system until the first atomic step outside of the current step is reached.
- `StepOver`: Execute the system until the next atomic step in the current scope is reached.

In an omniscient environment, we must consider executing the same stepwise execution environments in two contexts. The debugging session may be at the most current step of history, in which case the execution engine will execute the next step as a typical case of a stepwise execution debugger. However, if the user is not at the most current step, then the omniscient portion of the debugger will replay the rule from history. Consider a user executing a transformation rule that scans the model to identify a specific pattern of model elements to be modified by a subsequent series of transformation rules. This exact scenario can be seen in the Sierpinski Triangles (described in Sect. 5.2) where all sets of triangles currently existing are found, and then, a subsequent set of rules operates on these triangles. In this scenario, replaying from history may save substantial time in only a single rule application. Additionally, in scenarios where we can reduce multiple rule applications to a

single set of changes, we can also reduce traversal time. In each of these scenarios, we utilize the stored history of execution rather than the execution engine as typical for stepwise execution features. Thus, our implementation considers both of these scenarios as summarized in the following definitions.

4.1.1 Stepwise execution traversal features modified for omniscient traversal

- `Play`: If at the most current step of history, continuously execute the system. If not at the most current step of history, continuously replay the system.
- `StepIn`: If at the most current step of history, execute a single atomic step of the system execution, entering into any contained scopes. If not at the most current step of history, replay a single atomic step of the system execution, entering into any contained scopes.
- `StepOut`: If at the most current step of history, execute the system until the first atomic step outside of the current step is reached. If not at the most current step of history, replay the system until the first atomic step outside of the current step is reached.
- `StepOver`: If at the most current step of history, execute the system until the next atomic step in the current scope is reached. If not at the most current step of history, replay the system until the next atomic step in the current scope is reached.

Finally, we provide a set of additional features to enable traversal back through the history of execution. These features are designed to mirror the traditional stepwise execution environment to provide an intuitive extension to the most common debugging environment (stepwise execution).

4.1.2 Omniscient execution traversal features

- `PlayBack`: Continuously revert the system.
- `BackIn`: Revert a single atomic step of the system execution, entering into any contained scopes.
- `BackOut`: Revert the system until the first atomic step outside of the current step is reached.
- `BackOver`: Revert the system until the next atomic step in current scope is reached.
- `Jump`: If the target step of the `jump` is located in history before the current step, revert the system until the target step is reached. If the target step of the `jump` is located after the current step, replay the system until the target step is reached.

Further advanced navigation facilities (as seen in Bousse et al. (2015)) could be introduced to the environment, but these features would be defined using a set of the features defined above. The goal of this paper is to evaluate the efficiency and scalability of the model transformation engine modified to handle omniscient debugging. Thus, here we do not define advanced traversal features, such as traversing history by navigating through only the steps relevant to a specified element or transformation rule.

4.2 Collecting a history of execution

Our technique to support omniscient debugging collects a history of execution to enable traversal without re-executing rules. Figure 7 presents the structure of our trace of

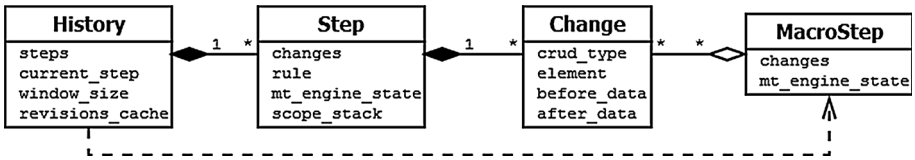

Fig. 7 Structure of history

execution, hereafter referred to as history. We define history using the following terminology and structures.

Change—A single atomic change operation.

- **CRUD type**—Type of change made. Changes can be create, update, or delete. Reads are not stored, because they are not necessary to recreate the model state at a given state, but we recognize that this information might be useful to developers (e.g., identify elements matched by LHS, but not altered by the transformation rule). Future studies could evaluate the impact of recording reads, but this paper is focused on efficiency of execution time and memory consumption.
- **Element**—ID of the element that was changed. An element is defined at the most atomic level. If two attributes of an object were changed, we would store two changes for distinct elements because each attribute is treated as an atomic element.
- **Before data**—Value of the element before the change.
- **After data**—Value of the element after the change.

Step—A step stores the full history of changes related to a single invocation of an atomic transformation rule; i.e., a rule that does not contain any other rules. In practice, this means that we must maintain placeholder steps for any rule that is defined using contained rules. The placeholder is used to maintain a proper history of scope transitions.

- **Changes**—A set of all changes that occurred during this step.
- **Rule**—ID of the model transformation rule related to this step.
- **MT engine state**—A general storage bucket for any auxiliary storage necessary for the MT engine (e.g., T-Core introduced by Syriani et al. (2015) maintains a packet that is passed between all transformation rules and is altered during each step).
- **Scope stack**—Maintains any scoping information. The scope stack stores the ID of the last step at the containing scope (nil if there is no containing scope). The transformation rule referenced by the last step of the previous scope contains the current step's transformation rule. Thus, each step only stores a single reference, but has access to the full scope stack at every step.

History—The complete record of all changes that have occurred during the transformation.

- **Steps**—Sequentially ordered series of all step entities in history.
- **Current step**—Index indicating the current step being observed.
- **Window size**—Size of the active window of history. History stores up to this limit in memory, and the remainder of history is serialized to permanent storage. This provides an upper limit to the memory consumption of history. By default, the window size is set to infinite (i.e., memory size of history is not limited).

- **Revisions cache**—A cache that stores a mapping of each element that has been changed to the set of steps where that element has been changed. This is used to quickly identify where a given element has been changed in history.

4.2.1 Evaluating the memory consumption of history

The space complexity upper bound of history, `O(As + Bc)`, is influenced by two key factors, the number of steps s and the number of changes stored in history `c`. `A` is a constant referring to the transformation state information, 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 vary minimally. Thus, for transformations affecting a large number of elements and containing a large number of steps, the structure performs poorly. However, the scaling concerns are due to the need for a complete trace of execution as assumed by our technique.

The current space complexity, `O(As + Bc)`, ignores the impact of the revisions cache stored in history, because we can amortize the cost of the lookup table across the set of changes stored in history. For each change in history, there will be a single entry in the lookup table. For each change in history, we add a constant amount of increase to the overall size of history. Therefore, we can redefine `B` to be the sum of the average memory consumption of a single change and the overage memory consumption of a single reference added to the lookup table.

Despite storing minimal information, history may eventually exceed the bounds of memory if the system is very large or the transformation involves enough changes. To address this concern, our technique maintains a window of active history. As mentioned in Sect. 2, this technique has been explored previously by Lewis (2003). However, as opposed to prior work, history outside of the current window is stored in 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 impacts the execution time 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.

4.3 Traversing a history of execution

The goal of the majority of existing literature in the area of omniscient debugging is to provide a scalable technique, in terms of memory usage, that enables reversing the execution of a software system. However, we have also explored a technique to efficiently, in terms of execution time performance, revert execution by identifying and executing a minimal set of changes. Our technique utilizes the execution history to create a macrostep that avoids unnecessary CRUD operations. A macrostep 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 macrostep 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. This technique is designed for a jump feature, where the user could provide a target step and then move to the target step by executing a minimal set of changes. However, `backOut`, `backOver`, `stepOut`, and `stepOver` can also utilize the

technique when reverting/replaying previously executed portions of the transformation. For these steps, the target step for the jump is implied by the type of step and the scope. When executing a `stepOver`, the target step is the next step in the same scope or a containing scope. Thus, these features each have the potential for iterating over an indeterminately large number of steps and changes.

4.4 Recognizing patterns of change

Our technique increases the efficiency of traversing history by identifying a minimal set of changes to execute. The set of changes executed avoids redundant incremental updates and executes direct state changes. To identify a minimal set of changes, we have developed an algorithm that recognizes a set of special cases where it can ignore changes. We handle five patterns to discern required versus redundant changes. All patterns consider only changes between the current step (i.e., the step being observed before the traversal) and a target step (i.e., the step being observed after the traversal). An individual change may be either create (C), update (U), or delete (D). The five patterns are defined as sequences of these three change types. The five patterns, as illustrated in Fig. 8, are as follows:

1. If only a single change (create, update, or delete) is identified for a given element, then the change is considered required and included in the minimal set of changes.
2. If multiple updates are identified for a given element, then only the update most local to the target step is included in the minimal set of changes. This pattern is particularly significant, because it may occur during the three remaining patterns. We only include the update most local to the target step and ignore all other updates.

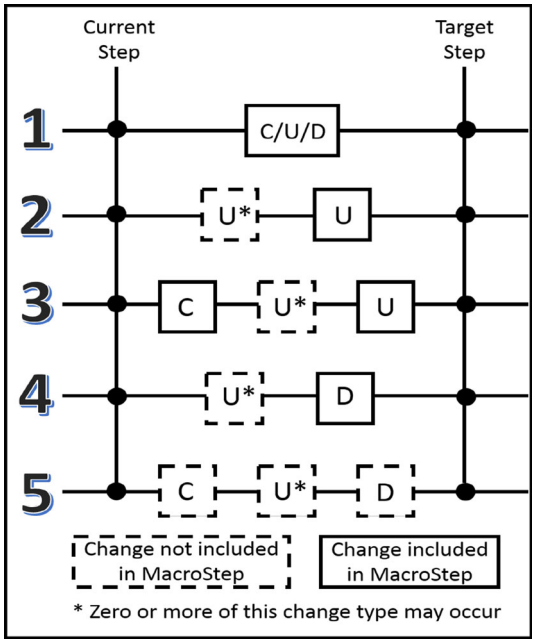

Fig. 8 Patterns to identify required changes between the current step and a target step

3. If we identify a create and an update for a given element, both the create and update are included in the minimal set of changes. We need the create operation to recreate the element and the update to reset the element to the appropriate state.
4. If we identify an update and delete for a given element, only the delete is included in the set of changes. We can ignore any updates, because the element will not exist after the set of operations.
5. If we identify a create and delete for a given element, no changes are included for this element. We can ignore all changes, because the element does not currently exist and will not exist afterward. Thus, taking no action will result in the model being in the correct state.

These patterns are described assuming the target step is after the current step in history (i.e., forward traversal). However, the patterns can still be applied when the target step is before the current step (i.e., backward traversal). When reverting execution, a create change is treated as a delete, and a delete change is treated as a create with an associated update to revert the element to its state before the recorded delete change.

4.5 Efficient omniscient traversal using MacroSteps

Thus far, we have defined traversal of execution using Steps, where a Step relates to executing a single transformation rule. However, when we traverse through history, it is not necessary to re-execute every CRUD operation. We take advantage of this fact by constructing and using MacroSteps to traverse history. As illustrated in Fig. 7, a MacroStep is similar to a Step in that it contains a set of changes, but a MacroStep contains a subset of the changes contained by a sequence of Steps. Furthermore, history uses, but does not store, MacroSteps.

Consider the following scenario: a developer is debugging a model transformation. Over the course of the transformation, a given element might be updated numerous times incrementally reducing the value of the element (e.g., a timer or resource indicator). However, if the developer wanted to jump back to the beginning of the transformation, the transformation engine could ignore most of these changes and revert the place directly to the appropriate state. To accomplish this, the debugger builds a MacroStep by identifying the change that will revert the place to the correct state (ignoring all other changes). Then, the MacroStep is used in place of a Step to revert the system. The changes contained by the MacroStep represent the minimal set of CRUD operations necessary to traverse from the current step to a given target step. When building a MacroStep, we use the patterns described in Sect. 4.4 to identify unnecessary changes that are then omitted from the MacroStep.

4.5.1 Algorithms to construct a MacroStep

In our technique, the debugger stores history using a structure that provides efficient access to the most recent change. Furthermore, the debugger maintains a revisions cache for each element containing a record of every step where the associated element was altered. The history stores all steps in increasing order within a dynamic array structure, and each step provides similar facilities for storing changes. Therefore, once the appropriate change is identified using the revisions cache, we can guarantee constant time access to the associated change in history. The `IterateElements` algorithm, provided in Listing 1, uses

these facilities to construct a minimal set of changes for a MacroStep to traverse from the current step to the target step.

```
for element from history in topologically sorted order
  if (element is not changed between current and target step)
    move on to next element
  else
    find the firstChange and lastChange between current and target steps
    #pattern 1
    if (firstChange and lastChange are the same)
      store the change
    #pattern 2
    else if (firstChange and lastChange are updates)
      store the change closest to target step
    #pattern 3
    else if (firstChange is a create and lastChange is an update
             and current is before target)
      store both changes
    else if 9lastChange is a delete and firstChange is an update
             and current is after target)
      store both changes
    #pattern 4
    else if (lastChange is a delete and firstChange is an update
             and current is before target)
      store lastChange
    else if (firstChange is a create and lastChange is an update
             and current is after target)
      store firstChange
    #pattern 5
    else if (firstChange is a create and lastChange is a delete)
      do not store either change
```

Listing 1 IterateElements Algorithm: iterate over all elements in history to build a MacroStep.

Assuming an element is changed between the current and target steps, the `IterateElements` algorithm finds the first change and last change within the interval and then applies each of the five patterns discussed in Sect. 4.4. The first pattern and second patterns are fairly straightforward. First, if there is only one change for the element, then we must keep that change. Second, if there are only updates, then we keep the update closest to the target step. Here, we can be sure there are no creates or deletes, because any create must be the first change, and any delete must be the last change. The third and fourth patterns rely on the direction of traversal. If we are moving backward, create and delete changes are treated as their opposite. Thus, if we recognize a create and an update when moving forward, we apply pattern 3. If we recognize a delete and an update when moving backward, we also apply pattern 3. Similarly, an update and a delete moving forward applies pattern 4, and a create and an update moving backward applies pattern 4. Finally, if we identify a create and a delete, then (as pattern 5 states) the element is both created and destroyed during the intervening steps and the related changes can be ignored.

The algorithm in Listing 1 is designed assuming that when building the macrostep, iterating over the changes contained in the sequence of steps from current to target 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 enough, this assertion does hold true. However, if the size and number of steps are relatively small or the size of the model is relatively large, the cost of iterating over all model elements may exceed the cost of iterating over all the steps. Thus, the debugger compares the number of

changes that must be iterated over to the number of elements in the model and then decides if iterating over the changes is more or less costly. However, we still ensure that a minimal set of changes is identified for the MacroStep. The upper bound of overall execution time remains the same, because we ensure that iterating over the changes will have similar lower bound or we iterate over the model elements. History maintains a count of the total number of previous changes in history at each step. The debugger then uses these counts to determine whether to iterate over the model elements or the full set of changes. The algorithm in Listing 2 provides the details of how a macrostep is generated when iterating over the intervening steps (from current step to target step).

```
for each step from current to target in order of execution
  for each change in the step
    #pattern 1
    if (we have not seen a change for this element)
      add change to createCache, updateCache, or deleteCache
    #pattern 2
    else if (the change is an update and we have only seen updates)
      keep the change closest to the target step in updateCache
    #pattern 3
    else if (the change is an update and we have seen a create
         and target is after current)
        replace any current change in updateCache
    else if (the change is a delete and we have not seen a create
         and target is before current)
         add chance to deleteCache
    #pattern 4
    else if (the change is a delete and we have not seen a create
         and target is after current)
      add the change to deleteCache
      remove any changes for this element from updateCache
    else if (the change is an update and we have seen a create
         and target is before current)
         add change to updateCache if no update has been seen yet
    #pattern 5
    else if (the change is a delete and we have seen a create)
      remove changes for this element from createCache and updateCache
```

Listing 2 IterateSteps Algorithm: iterating over steps from current to target to build a MacroStep.

The `IterateSteps` algorithm also identifies each of the five patterns discussed in Sect. 4.4 to provide a minimal set of changes. Unlike the `IterateElements` algorithm, the `IterateSteps` algorithm does not have access to all of the changes for a given element at a time. Therefore, we store the changes in a set of caches that can be referenced later to identify cumulative effects. We will recognize each pattern bit by bit until we have identified the full pattern. Whenever we identify the first change for a given element, we assume pattern 1 and store the change in the relevant cache. If we do not identify any further changes, then we were correct to assume pattern 1. If we identify multiple updates, then we recognize pattern 2 and keep only the update closest to the target step. There are two conditions where we can identify pattern 3 (one for each direction of traversal): first, if we are traversing forward and identify an update after having identified a create; second, if we are traversing backward and identify an update after having identified a delete. Similarly, there are two conditions where the debugger identifies pattern 4: first, if we are traversing forward and identify a delete after having identified an update; second, if we are traversing backward and identify a create after

having identified an update. Finally, if we identify a delete and have already identified a create, then we recognize pattern 5.

4.5.2 Evaluating the MacroStep construction algorithms

The `IterateElements` algorithm (Listing 1) has `O(n*lg(n) + n*lg(m))` execution time complexity, which can be simplified to `O(n*lg(n))` for large-scale models (i.e., cases where the scaling becomes notable). Here, `n` is the number of elements in the model that have been altered (only elements that have been altered are stored in history) and m is the number of steps where a given element is altered. The 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. Listing 1 displays the basic algorithm used to build a macrostep. To make the simplification, we recognize that to maintain `O(n*lg(m))`, the history must have as many or more changes per element as there are elements stored in history (previously stated to be only those elements that are changed). As n reaches levels where scale is a concern, and even for small-scale scenarios with thousands of elements in history, the number of changes required becomes unreasonable for most transformations. We expect the number of changes for a given element to be small, and the execution time complexity upper bound to approach `O(n*lg(n))` in practice.

The upper bound of execution time complexity for the `IterateSteps` algorithm, when iterating over the steps from current step to target step, is determined by the total number of changes that must be evaluated for inclusion in the macrostep. Thus, the algorithm (Listing 2) has an `O(abs(mc -- mt))`, where `mc` is the sum total number of changes for all steps up to and including the current step, and `mt` is the sum total number of changes for all steps up to and including the target step. Therefore, the theoretical tipping point for choosing iterating over changes (Listing 2) rather than iterating over all elements (Listing 1) is when the number of changes, `abs(mc -- mt)`, is less than `n * lg(n)`.

4.6 Maintaining scope in history

As mentioned previously, our omniscient technique is an extension of stepwise execution. As such, we provide scope-based operations (i.e., `stepIn`, `stepOver`, `stepOut`, `backIn`, `backOver`, and `backOut`). To support these operations while traversing through history, scope information must be provided. The ideal solution is to provide a full scope stack at any given step. By providing the full stack, we may provide stack traces similar to those provided by exceptions in GPLs (e.g., Python, C++, or Java). These stack traces provide developers with additional information regarding the current state of the system. However, providing a full copy of the stack trace for each element can create a state space explosion by replicating scope information between multiple steps. To address this concern, we developed a technique similar to a cactus stack (or spaghetti stack) (Clinger and Ost 1988). Every step stores a pointer to a step where the current (relative to the step) top of the stack is stored. We store a given scope's information only a single time for each traversal through the scope. When a scope is entered, the current step has the scope information added, and each subsequent step which is directly contained in the same scope stores a link to the initial step for the scope. Each scope node then stores a link to its containing (or parent) scope. Thus, we store a single node for each time a scope is encountered. A flyleaf pattern could be applied to our technique. The flyleaf variant would present a minimal trace of scope stack information through eliminating redundancy of

changes and values. Consider if we had a string value that is repeated through the program that could be replaced with a single object referenced in each location rather than repeating the string value.

4.7 Supporting omniscient debugging in other modeling platforms

Although our context for exploration of omniscient debugging is AToMPM, we believe that the algorithms and general technique of our omniscient debugger can be ported to other modeling tools. Our technique at the most primitive level is based upon capturing and replaying CRUD-level operations resulting from applying model transformation rules. As such, the history structure presented is generic and does not rely on any specific transformation features. The technique requires only the ability to capture and replay CRUD-level operations during execution. However, some transformation environments may need to be modified to emit the CRUD operations during execution to enable the collection of history. We expect this modification will not cause a significant difference in transformation execution, but the details may vary with implementation specifics. The omniscient traversal methods do not require modifying the transformation engine because they entail only executing CRUD operations on the model. Furthermore, supporting the algorithms as presented here requires only the ability to support constant time access array-like structures and support for a cache structure that maps elements to a listing of the steps where the element is changed. As these describe basic data structures, we expect the implementation environment of any model execution engine should be able to meet these requirements.

Beyond the mechanical requirements, the environment would need to be attuned to any differences between the languages. The most significant concern here is regarding the selection of the granularity at which to define a Step. We define a Step in MoTif as a single non-composite MoTif transformation rule and in T-Core as a single T-Core primitive. However, in other languages the rules may be defined using a set of lower-level operations, similar to a method in Java. Here the implementation will need to decide on the precise level of granularity that defines a Step. Additionally, the implementation will need to precisely define scope to employ the scope stack (assuming the concept of scope is relevant to the target transformation language). Our structure for storing the scope stack is generic in assuming a layered scoping mechanism common throughout many GPLs and MTLs (e.g., helper functions in ATL[1]), but the recognition of scoping will need to be tailored to the specific MTL. We expect this recognition to be a constant time addition to the initial execution time as it is in our implementation within the AToMPM environment. Thus, we do not anticipate any significant differences in runtime, but the implementation in other modeling tools must be customized to match the relevant MTL structure and semantics.

5 Empirical evaluation study design

We conducted an empirical study to evaluate the performance and scalability of our omniscient debugging technique on two model transformations. In this section, we describe the design of the empirical study as well as a discussion of the threats to the validity of the results.

[1] http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#ATL_Helpers.

5.1 Research questions

The primary goal of the study is to understand the execution time performance and scalability (in terms of memory consumption) of our omniscient debugging technique on two model transformations designed in two different MTLs.

The focus of the case study is to address the following questions:

RQ1. Is there a significant difference in execution time between executing a model transformation with omniscient debugging versus stepwise execution?
RQ2. Is there a significant difference in execution time between executing a model transformation with or without macrosteps?
RQ3. Is there a significant difference in execution time between the `iterateSteps` and `iterateElements` algorithms?
RQ4. At what point does omniscient debugging outperform restarting a model transformation in terms of total execution time?
RQ5. What is the effect of the changes and steps on memory consumption in history?
RQ6. What is the impact of history on total memory consumption?

We perform Wilcoxon signed-rank tests for *RQ 1*, *RQ 2*, and *RQ 3*. For each hypothesis test, we do not presuppose the directionality of the difference during testing. Therefore, each hypothesis test is two-tailed. For each test, we formulate a null hypothesis to evaluate whether there is a significant difference between the two sets under comparison. If, after testing the null hypothesis, we find we can reject it with a high confidence ($p = 0.05$), we accept an alternative hypothesis. Accepting the alternative hypothesis corresponds to there being a significant difference between the two sets. Below we provide an example null hypothesis (H0) followed by the corresponded alternative hypothesis (HA).

`H0: timewithomniscience = timewithoutomniscience`
`HA: timewithomniscience ≠ timewithoutomniscience`

The null hypothesis asserts that omniscient debugging *does not* significantly affect the execution time of the model transformation, and the alternative hypothesis states the opposing view that omniscient debugging *does* significantly affect the execution time of the model transformation.

Omniscient debugging does significantly affect the execution time of the model transformation.

5.2 Debuggers and model transformations used in evaluation

In this study, we ran two model transformations using both AODB and the standard stepwise debugger provided in AToMPM v0.5.4. AODB is provided within an extended version of AToMPM v0.5.4 where the transformation engine has been extended to support omniscient traversal features. Thus, the omniscient debugger and stepwise debugger within AToMPM can provide a direct comparison. For the AODB omniscient debugger, we executed `play` until the end of the transformation was reached. We then called `BackIn` repeatedly until we returned to the initial step. After returning to the beginning, we re-executed each step of the transformation using `StepIn` until we reached the end again. By using `BackIn` and `StepIn` to proceed backwards and forwards through the transformation, we were able to collect execution time and memory usage statistics for the history

of the transformation. After finishing the `StepIn` task, we divided the collected history into ten partitions resulting in eleven boundary steps. We used the boundary steps as jump points and proceeded to jump from every jump point to every other jump point.

For the stepwise execution, we performed `play` until the end. Because history does not exist and the omniscient traversal features are not supported for stepwise execution, this was the only task we performed. However, we captured incremental step times throughout the `play` operation enabling us to compare directly with the data collected from the omniscient debugger.

The two model transformations selected for this study were acquired from the 2014 TTC. The first of these transformations is the Movie DB Case described in Sect. 3. For this study, we chose to use element sizes of 116, 580, and 1160 elements. For details of the Movie DB Case transformation implementation, please refer to the solution from the 2014 TTC by Ergin and Syriani (2014). In this study, we modified the transformation only to connect the three primary tasks within a single transformation.

The second transformation constructs a Sierpinski Triangle (metamodel provided in Fig. 9a), which is a fractal where each generation creates an additional level of depth. The triangle is created by starting with an initial equilateral triangle with a horizontal base. The next step shrinks the triangle in half, makes two copies, and positions the three smaller triangles so that each triangle touches the two other triangles at a corner, effectively splitting each triangle into three smaller triangles (Fig. 9b). The second step is then repeated to create each new generation. Table 1 lists the generations used in this study and the number of model elements created by AToMPM at each generation. The number of model elements in Table 1 is higher than in the conceptual problem. This variance is due to the underlying model representation of AToMPM where an edge is represented as a specially typed node with two edges connecting the edge node to the two traditional nodes connected by the edge. The additional complexity is necessary to

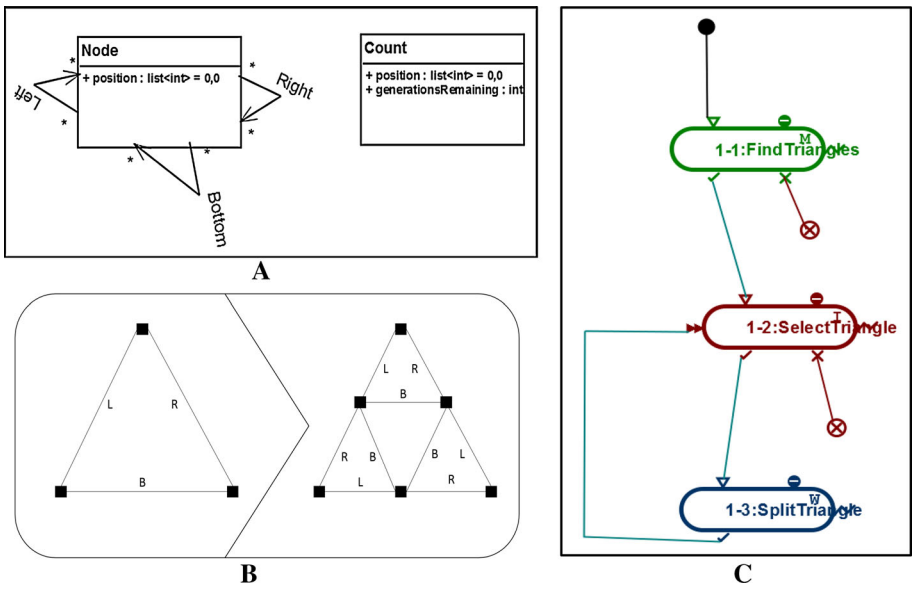

Fig. 9 Sierpinski Triangles TTC details. **a** Metamodel, **b** graph transformation rule, **c** MoTif transformation

Table 1 Number of model elements for each generation of Sierpinski Triangle

Generation	Model elements
0	12
1	33
2	96
3	285
4	852
5	2553
6	7656
7	22,956
8	68,892

represent typed edges, because the underlying low-level representations allow only generic edges.

The two transformations possess distinct properties that impact the results of various tests for the analysis tasks described in Sect. 5.6. The Sierpinski Triangle is able to perform a single search for all triangles that must be split to create the next generation. Figure 9c displays the T-Core transformation that generates the next generation of a Sierpinski Triangle (splits all triangles) by executing the pattern matching and graph rewriting defined in a graph transformation rule. The Matcher (`FindTriangles`) identifies all triangles in the current generation, and then, the Iterator (`SelectTriangle`) repeatedly selects a triangle for the Rewriter (`SplitTriangle`) to split until no matches from the previous generation remain. Thus, the Sierpinski Triangle transformation is able to perform a single search over the entire graph for each generation. However, the Movie DB Case transformation must search again after identifying each couple. This is due to the update statements making previous matches invalid for the rule. Thus, the Movie DB Case transformation spends a significant amount of additional time searching over the model than the Sierpinski Triangle. This difference is especially interesting given that executing in history does not require repeating these computationally expensive searches. Additionally, both transformations generate their own sample models with the Sierpinski Triangle always starting with the same initial model for all target generations (i.e., a single triangle representing generation 0).

5.3 Measures used in the evaluation

The goal of omniscient debugging is to reduce the time it takes for a developer to return to a previous state in execution. Omniscient debugging makes this easier by allowing for bidirectional execution through the history of an execution. Without omniscient debugging, the developer must re-execute the transformation starting from the beginning. To understand how time can be saved using our omniscient debugger, we recorded the execution time for each step and jumped through history in milliseconds and compared it to the time required for a normal execution. Additionally, we record which rule is executed at each step. This added information allows us to refer back to the transformation when we need to determine the types of operations that were applied during the step.

In addition to the execution time required for moving through history, we collect the added memory usage required to store the changes that occur at each step. Because changes during the model transformation can be ambiguous (as discussed in Sect. 2.3),

this additional memory usage is required to recreate the exact changes. For this reason, we would like to know the impact that history has on memory usage. To understand this impact, we collect the memory usage in bytes at each step for the overall system, the revisions cache, state information for the most recent step, and the changes in the most recent step. This information is also recorded for each macrostep built during a jump. Each of these measures is used to give an overall understanding of the impact on the system.

5.4 Configuration of experimental platform

To conduct our study, we used a 64-bit Windows 8 machine with an Intel dual-core 3.33 GHz processor with 4 GB of RAM. The tool used for the study was AToMPM which had a model transformation engine written in 32-bit Python. Due to using the 32-bit version of Python, AToMPM is limited in the amount of memory it can address. AToMPM uses Python-igraph for holding and manipulating the low-level graph representation of the model. The tool is capable of executing model transformations in both the T-Core and MoTif hybrid MTLs. In this context, hybrid refers to the combination of graph rewriting rules, which are purely declarative and imperative control flow.

AToMPM is a cloud-based modeling tool that sends communications between the model transformation engine and the front-end client across a network. The client is written in HTML5 and JavaScript with the Chrome browser as the main development environment. For the purposes of this study, we disabled communications to the client. This allowed us to isolate execution time on the model transformation engine and remove the additional overhead of any added message passing.

5.5 Data collection and analysis

We instrumented the model transformation engine to collect information during the execution of the model transformation. For each step, we collected the time it took to execute the step, the number of changes in the step, the number of elements in the step, and the number of creates, deletes, and updates involved in the step. We then executed the model transformation in the forward direction, backward through history, forward through history, and jumping through history after splitting the history into ten partitions. Afterward, we re-executed each model transformation while forcing the macrostep to be built using either the `iterateSteps` or `iterateElements` portions of the algorithm. We then used this for comparison of the two techniques.

For *RQ 1*, *RQ 2*, and *RQ 3*, we conducted Wilcoxon signed-rank tests to determine whether a significant difference is found. The Wilcoxon signed-rank test is the nonparametric analog of the *t* test. We did not assume directionality of the difference.

For *RQ 1*, we compared different step types and levels of the model transformations, as well as all steps across all types and all levels. The types of steps that we considered were search, scope, and change. The levels we used were the 1 iteration, 5 iterations and 10 iterations of the Movie DB Case model transformation, as well as the combination of all iterations. We also included 8 iterations of the Sierpinski Triangle. For *RQ 2* and *RQ 3,* we used 7 different levels across the two model transformations. We used the same iterations of the Movie DB Case model transformation as the *RQ 1* and then included 3, 5, 6, and 8 iterations of the Sierpinski Triangle. We then combined all iterations of both model

transformations to form a combined Movie DB Case and Sierpinski Triangle. Finally, we also combined all levels of both model transformations.

For *RQ 4,* we identify cases that highlight the different circumstances that can make traversing through history either faster or slower than re-executing the code. We use these cases to give a high-level view of how characteristics of the model transformation can impact the effectiveness of history.

Finally, in *RQ 5* and *RQ 6*, we captured the memory consumption of the system as we progress through the model transformation. We compared this information to the amount of information contained in history, as well as to the number of changes and number of steps currently in history. Through these questions, we have obtained a better understanding of the impact of omniscient debugging on the memory usage of the model transformation engine.

5.6 Threats to validity

The study has limitations that may affect the validity of our findings. In this section, we describe some of the limitations as well as our attempts to mitigate them.

Threats to conclusion validity concern the degree to which the conclusions we reach about the relationships in our data are reasonable. In order to mitigate these concerns, we limited the comparisons we made to execution time on the model transformation system and attempted to limit the effects of other variables, such as communication time to the client. In addition, we used nonparametric statistical tests and did not make any assumptions about the distributions of the data.

Threats to construct validity concern how well the measurements used in the study describe the concept being studied. Possible threats to construct validity include the effects other processes on the host machine have on the time required to execute the steps in the transformation. In order to limit these issues, we ensured that a minimal set of processes were running on the host.

Threats to internal validity include possible errors in executing the study procedure or defects in the tools used. To mitigate these issues, the model transformation engine was instrumented to automatically log execution time performance and memory usage along with which debugger generated the log, which transformation was run, what size model was used during the run, and which traversal feature (e.g., stepIn or stepOut) was used to generate the log. The instrumentation of the system may have affected the execution time performance. In order to account for this possibility, we instrumented logging outside of the measured tasks to limit the total impact whenever possible. When it was not possible to instrument logging outside the measured tasks, we attempted to keep all runs similar and to record any impact of the instrumentation in order to remove the impact from recorded observations.

Threats to external validity concern the extent to which we can generalize the results. We chose two model transformations with varying factors to gain a more complete understanding of how the omniscient debugger will work on other model transformations written in these languages. We used model transformations written in two MTLs. To understand how our technique would be affected by other transformation languages, we would need to rerun our experiments in those languages. However, our results should be similar for other model transformations written in T-Core and MoTif, and languages with a similar set of features.

6 Results obtained from performance and scalability study

In this section, we present quantitative data from statistical tests and descriptive statistics. This section is organized by research question. Section 7 will discuss the implications and provide qualitative analysis.

6.1 Is there a significant difference in execution time between executing a model transformation with omniscient debugging versus stepwise execution? (*RQ 1*)

To answer this question, we recorded the elapsed time while running each model transformation. Between the two transformations, we recorded the elapsed time for four different levels: 1, 5, and 10 iterations of the Movie DB Case and 8 generations of the Sierpinski Triangle. We can limit the evaluation to only a single case of the Sierpinski Triangle transformation, because each subsequent generation executes the same steps as the previous step plus an additional generation. We also recorded the elapsed time for three different types of step: change, scope, and search. Change steps are only concerned with executing some change (e.g., creating new nodes). Scope steps refer to a subprocess and do not make any direct changes. Search steps may include some changes, but also include a significant find operation. An example of a search operation is the `findTriangles` step of the Sierpinski Triangles transformation, which searches over the entire model and identifies all triangles.

The results of all steps for each of the four levels are represented by the boxplots in Fig. 10. From these boxplots, it can be seen that the times for each step are similar. The biggest difference is visible for the Sierpinski Triangles. To determine whether a significant difference exists between omniscience and stepwise debugging for any level, any step type, for any step of any level, or for all steps of all levels and step types combined, we conducted a series of Wilcoxon signed-rank tests.

The results of the Wilcoxon tests failed to show any significant difference with the exception of the change step type for 10 iterations of the Movie DB Case model transformation. For this level, building the changes in history for omniscient debugging showed a slight increase in execution time over the stepwise debugging. All 23 other cases failed to display a statistically significant difference in execution time.

6.2 Is there a significant difference in execution time between executing a model transformation with or without macrosteps? (*RQ 2*)

The intent of *RQ 2* is to determine whether there was a significant difference between traversing history via executing all intermediate changes (i.e., using `back` or `stepForward`) versus jumping through history. When jumping, we compute MacroSteps to define the precise set of changes that will be executed to traverse to the target location in history. Jumping through history uses MacroSteps, and stepping through history does not. For each model transformation, we divided the history into ten partitions of steps. This resulted in eleven distinct endpoints for moving through history. We then recorded the elapsed execution time for jumping from each endpoint to each other endpoint in both the forward and backward directions. We replicated this experiment for seven distinct levels across the two model transformations. We looked at 1, 5, and 10 iterations for the Movie DB Case transformation. We looked at 3, 5, 7, and 8 generations for the Sierpinski

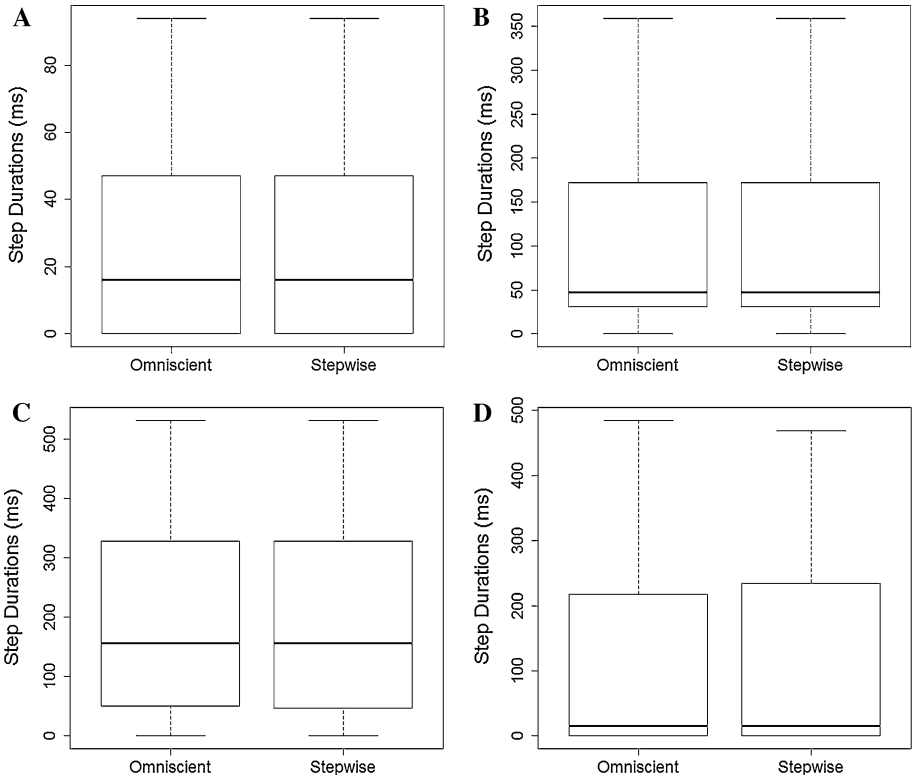

Fig. 10 Measuring differences in execution time: omniscient versus stepwise. **a** Movie DB Case 1 iteration, **b** Movie DB Case 5 iterations, **c** Movie DB Case 10 iterations, **d** Sierpinski Triangles

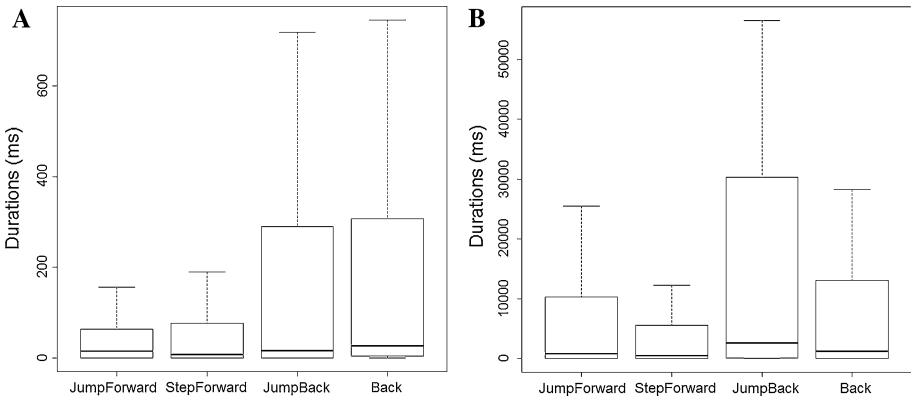

Fig. 11 Measuring differences in execution time: jump versus stepping

Triangles transformation. We also analyzed the combined result sets for each model transformation at all levels and the combination of all levels across both transformations.

Figure 11a contains boxplots representing the steps for all iteration levels (1, 5, and 10) of the Movie DB Case model transformation. We see that for moving forward through

history, the spread increases for stepping forward over jumping forward, while the median for stepping forward goes down. While moving backward, there is an increase in the entire spread for moving back through history versus jumping backward. Also worth noting, we observe that moving backward through history results in higher execution times than stepping forward through history.

Figure 11b contains boxplots for the Sierpinski Triangle. Unlike the results in Fig. 11a, where we observed an increase in the spreads for moving (i.e., either stepping or jumping) forward and backward through history, in Fig. 11b we also observe a decrease in the spread for both stepping forward and stepping backward in history versus jumping.

We conducted Wilcoxon signed-rank tests for each of the different levels and combinations. We also investigated whether there exists a significant difference between jumping of any type and the combination of stepping forward and moving back. For the Movie DB Case, no significant difference was found between jumping through history and stepping through history. However, for the Sierpinski Triangle, a significant difference was found for all comparisons except for stepping forward at the 3 iteration level. No significant differences were found when looking at the combination of the two model transformations.

6.3 Is there a significant difference in execution time between the `iterateSteps` and `iterateElements` algorithms? (*RQ 3*)

There are two possible paths for building a MacroStep during the execution of the model transformation. The first is `iterateSteps`, which in our MacroStep building algorithm occurs when the number of steps is less than the number of elements in the system. The second is `iterateElements`, which occurs in the opposing situation. To better understand the impact of the differences in how these two methods build the MacroStep, this research question focuses on whether there is a significant difference in the execution time between the two. For this question, we looked at what would happen if we forced MacroSteps to be built in one of the two paths and then compared the outcome. We used the same levels as *RQ 2*.

Figure 12a shows the spread of building each step for all iterations of the Movie DB Case model transformation. From this graph, there appears to be a slight decrease in `iterateElements` versus `iterateSteps`. Figure 12b shows the spreads for all iterations of the Sierpinski Triangle. In the case of the Sierpinski Triangle transformation,

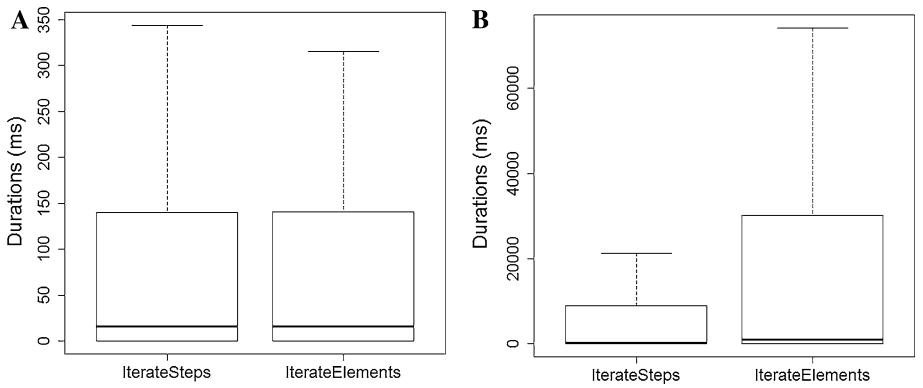

Fig. 12 Measuring differences in execution time: IterateSteps algorithm versus IterateElements algorithm. **a** Movie DB Case all levels, **b** Sierpinski Triangles all levels

the `iterateElements` was observed to have a large increase in spread versus `iterateSteps`.

Again, we conducted Wilcoxon signed-rank tests to determine whether there was a significant difference between `iterateSteps` and `iterateElements` at any level. For the Movie DB Case, we did not identify a significant difference at any level. However, for the Sierpinski Triangle and for the combination of the two systems, we identified a significant difference for every level.

6.4 At what point does omniscient debugging outperform restarting a model transformation in terms of total execution time? (*RQ 4*)

We believe that there exists a point for which re-executing a given model transformation is faster than executing a jump through the history for a given transformation. To gain a better understanding of this situation, we recorded the elapsed time from beginning to end of the model transformation and the elapsed time of jumping back to different points in history. We then took the case of jumping from the end of the model transformation to a previous point.

Figure 13 shows the point in which re-execution becomes better than omniscient. This graph shows the percentage of the model transformation that may be re-executed before it would be better to jump back from the end. For all levels of the Movie DB Case, and for 3 iterations of the Sierpinski Triangle, it was always better to jump back from the end than to re-execute the transformation. For 5 iterations of the Sierpinski Triangle, 70 % of the model transformation may be re-executed before jumping back becomes better. For 7 iterations, this value is 80 %. The performance focus of this discussion also does not consider the importance of non-determinism as described in Sect. 2.2

Multiple factors and characteristics influence these results (discussed further in Sect. 7).

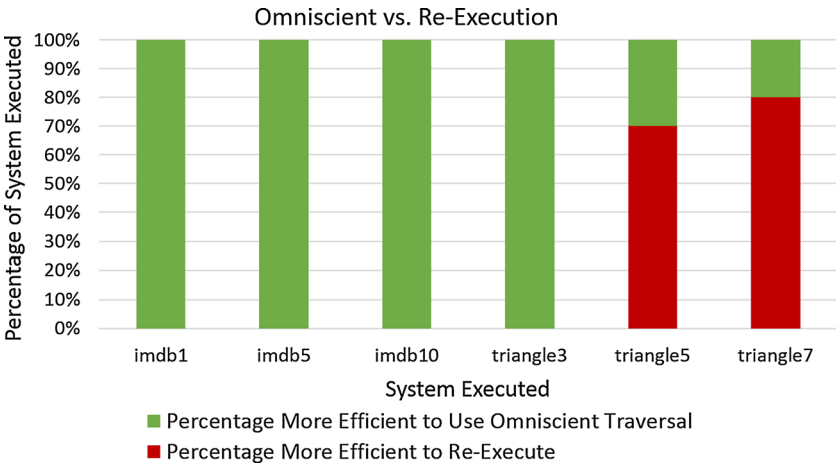

Fig. 13 From the end of execution, what percentage of the system can be re-executed before omniscient traversal is more efficient

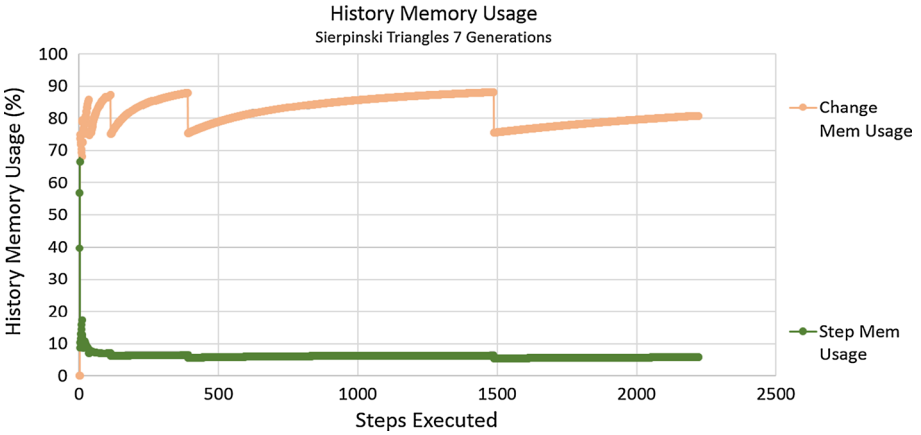

Fig. 14 History memory usage Sierpinski Triangles 7 generations

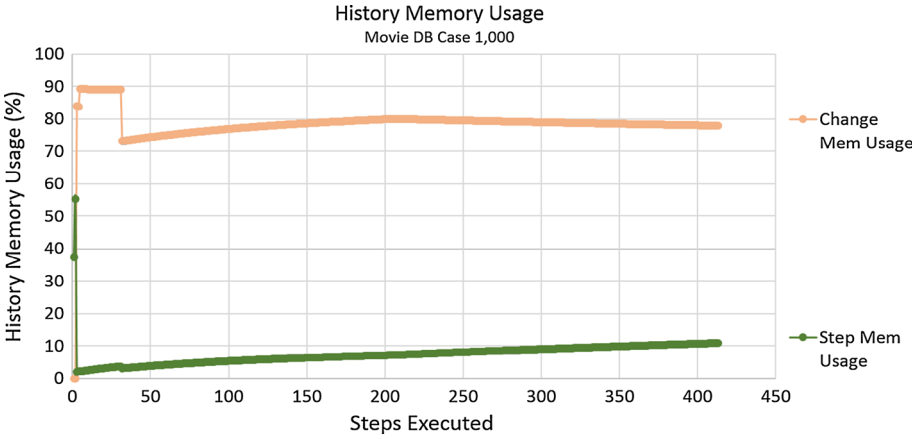

Fig. 15 History memory usage Movie DB Case 10 iterations

6.5 What is the effect of changes and steps on memory consumption in history? (*RQ 5*)

To determine whether the changes or the steps have a greater overall influence on the memory consumption of our history structure (see Sect. 4.2), we recorded the total amount of memory used by the history as well as the amount of memory used by the changes and the amount of memory used by the steps. When recording the memory used by a given step, we omit the changes included in the step and focus on the other features of a step (e.g., scope information and transformation rule information). We then calculated after the system executed each subsequent step of the transformation, the percentage of memory composed of changes and the percentage composed of steps. The results of this analysis for 7 generations for the Sierpinski Triangle are shown in Fig. 14. The results of this analysis on the Movie DB Case with 10 iterations are shown in Fig. 15.

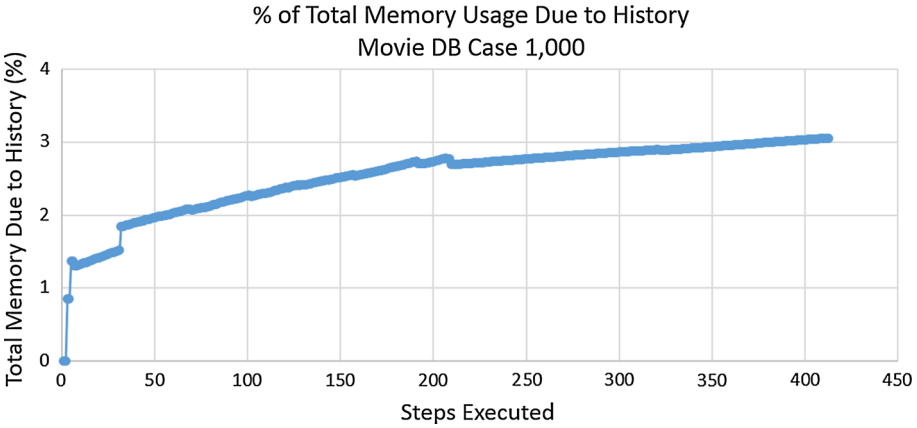

Fig. 16 Movie DB—percentage of total memory usage for the transformation engine due to history

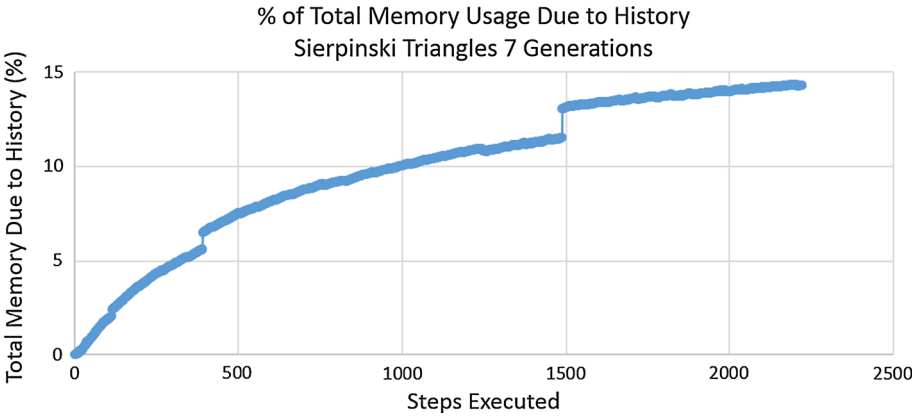

Fig. 17 Sierpinski—percentage of total memory usage for the transformation engine due to history

For both model transformations, the percentage of memory usage by steps is significantly lower than the percentage of memory usage by changes. For the Sierpinski Triangle, sudden drops are observed in the memory usage of changes with each new generation of the transformation.

6.6 What is the impact of history on total memory consumption? (*RQ 6*)

In addition to the amount of memory used by changes and steps in history, we sought to understand the percentage of memory used by history for the entire model transformation engine. To calculate this value, we recorded the total amount of memory usage by history as well as the total memory usage of the model transformation engine. Figure 16 presents the results of this analysis for 7 generations of the Sierpinski Triangle. Figure 17 presents the results of this analysis for the Movie DB Case with 10 iterations.

It is worth noting that these figures use a *y*-axis with a maximum value of 25 %. The reduced scale on the y-axis (25 % rather than 100 %) is because the total usage of history in the most extreme case observed is less than 15 % of the total amount of memory usage of the entire model transformation engine. For the Movie DB Case model transformation, the total memory usage for history is always below 5 %, while the total memory usage for the history of the Sierpinski Triangle is always below 15 %.

7 Discussion of empirical study and lessons learned

In this section, we address each research question providing a discussion about the implication of the results presented in Sect. 6. This section is organized according to the research question being addressed.

7.1 Is there a significant difference in execution time between executing a model transformation with omniscient debugging versus stepwise execution? (*RQ 1*)

The goal of this question was to show that there is not a significant difference between running the model transformation using normal stepwise execution and omniscient debugging. Because adding changes to history requires time, it may affect the performance of the technique if the debugger results in a significant change of the execution time performance. Therefore, when studying this question, the desired result would be to find no significant differences.

We chose three different step types to investigate as we conducted this study: scope, search, and change. Each of these steps has a different effect on what the model transformation engine does at that step. Search steps are meant to identify elements in the model based on some pattern. This type of step does not typically generate any changes, and the time spent for the searching operation typically vastly outweighs time spent changing the model. Thus, the omniscient debugger performs a minimal set of operations (i.e., only creating a new empty step) in addition to basic execution concerns common to both debugging approaches. Therefore, we did not expect to find a significant difference in observed execution times except for those produced from noise. Similarly, the scope steps are used to enter and exit different levels of scope in the transformation and do not result in changes to the model. We also did not expect to see a significant difference in this case. However, for the change step types, changes must be added to the trace in history. We expect that with enough changes, there might be a significant difference in the execution time of steps of this type. This assumption was supported by the results of the study. For smaller cases, no significant difference was found for the change type. However, for the largest level of 10 iterations of the Movie DB Case model transformation, a significant difference was identified.

Though a significant difference was found for the change type of the 10 iteration Movie DB Case level, no significant differences were identified for any level when considering all step types together. This is an important result as it indicates that building the history should not significantly decrease execution time performance of a model transformation engine.

7.2 Is there a significant difference in execution time between executing a model transformation with or without macrosteps? (*RQ 2*)

One of the benefits of having history is that it allows the developer to jump to any step that has already been executed. In order to facilitate such a traversal, we implemented an algorithm that builds a macrostep with all changes from the current step to the target step (see Sect. 4.4). Additionally, while building the macrostep we may exclude redundant changes from execution. However, a consequence of building the macrostep is that it requires additional computation that is not required in other steps. If this additional computation is more costly than moving through history, the benefits of the macrostep are diminished.

The purpose of *RQ 2* is to understand how building the macrostep compares to moving through execution (either forward or backward) via a more traditional stepping algorithm in which we execute all changes for each step iteratively until the traversal is complete. We partitioned the steps in history into ten different sets. The partitioning resulted in eleven endpoints. For each of the eleven endpoints, we jumped from each point to every other point in both the forward and the backward directions. For each direction, we also computed the elapsed time for traversing using back and `stepForward` through history. We then compared the times for each of these moves. We found that for the Movie DB Case model transformation, there was a decrease in the overall time required to move through history by using steps; however, no significant difference was detected. For the Sierpinski Triangle, there was a significant difference detected between the jumps and `stepForward` and back. The main cause of this difference is due to the difference in performance between `iterateSteps` and `iterateElements`. We believe that we can increase the performance of jump by changing the cutoff point between `iterateSteps` and `iterateElements`.

An additional finding during this question is that for both model transformations, there is a significant difference between moving forward and moving backward in history. There are two reasons for this. The first is that we found a large increase in changes for moving backward versus moving forward resulting in added computation in the backward direction. The second reason is due to the need to reverse the changes that were added in the macrostep. In the current implementation of the debugger, changes must be added to the step in the forward direction and then the resulting set is reversed to move in the backward direction. This added to the time required for both stepping backward and jumping backward.

7.3 Is there a significant difference in execution time between the `iterateSteps` and `iterateElements` algorithms? (*RQ 3*)

Depending on the number of changes and elements in the system, macrosteps will either be built by iterating over all elements in history or by iterating over all steps. The purpose of this question is to gain a better understanding of the efficiency of the two techniques and whether one significantly outperforms the other.

The results of this study found that for the Movie DB Case model transformation the two techniques did not have a significant difference and are roughly the same in terms of execution time performance. However, for the Sierpinski Triangle, there was a significant difference at all levels. The `iterateElements` algorithm has a few shortcomings that make it a bad choice to use when there are a lot of elements and few changes per element. One concern is the need to identify the `mostLocal` and `mostRecent` changes for each element in the system. If the number of steps is large and the element is changed frequently,

then identifying `mostLocal` and `mostRecent` changes becomes an issue because the process must search through a cache of revisions to identify the `mostLocal` and `mostRecent` changes. In the case of the Sierpinski Triangle, there exist a large number of elements in history, but few changes per element. Another potential concern is the need to reorder the changes for a MacroStep. In the `iterateElements` algorithm, the changes are not identified based on when the change occurred during the initial execution (as they are in the `iterateSteps` algorithm). However, as the changes must be completed in a set order to prevent conflicts (attempting to update an attribute before an element has been created), we must ensure the resulting set has the appropriate order. This results in a need to reorder elements. This difference in performance also affects the results of the jumps in RQ 2. Because of the difference in performance in certain cases, we believe that changing when the algorithm determines whether to use `iterateSteps` versus `iterateElements` will result in a significant increase in performance of the MacroStep building algorithm.

7.4 At what point does omniscient debugging outperform restarting a model transformation in terms of total execution time? (*RQ 4*)

One of the arguments presented in favor of omniscient debugging was to reduce the cost of reaching a desired state by removing the need to re-execute the system. However, depending on the transformation and the amount of changes in the system, there are times when using omniscient traversal may be more expensive than re-executing the system from the beginning. There are a number of factors that affect the boundary points for when re-executing should be selected over reverting execution through omniscient features. If the number of steps between moving from the current point in history to the target point is greater than the number of steps that would need to be executed from the beginning of the transformation, then it may be better to re-execute the transformation. However, if these steps are computationally expensive to execute and trivial to revert using logged information, the larger number of steps may take less time to revert than executing the smaller number of computationally expensive steps.

We studied these boundary points for all levels of the two model transformations to identify when it would be better to select re-execution over omniscient traversal. To study this, we divided the execution histories into 10 partitions and compared re-executing from the first step with reverting from the last step to reach each intermediate partition boundary. For all levels of the Movie DB Case transformation, omniscient traversal was observed to be more efficient. This is due to the costly search steps that are a part of this transformation. The amount of time required for identifying couples that match the search criteria is costly, and this cost is incurred repeatedly during the transformation. Thus, avoiding the search process with omniscient traversals after the initial execution is preferable.

For the Sierpinski Triangle transformation, it is better to use omniscient traversal when the number of iterations is low. The omniscient traversal was only preferable for small jumps back through the system. We note that improving the MacroStep building algorithm as discussed in Sect. 7.3 may alter these results. The Sierpinski Triangle model transformation does have a costly search step, but the search step (`findTriangles`) is only executed at the beginning of each generation. As the generations increase, the number of change steps compared to search steps grows significantly in favor of the change steps. Therefore, traversing through the latter generations of the model transformation is less efficient than in the earlier generations of the transformation. If the current execution has reached the end of the transformation, omniscience traversal only provides a benefit when moving back to the point where 70 % of the execution has been executed for generation 5.

Re-executing is faster at any earlier point. For generation 7, the boundary shifts to 80 %. Again, this is due to the high growth rate and number of changes at the end of the model transformation.

7.5 What is the effect of changes and steps on memory consumption in history? (*RQ 5*)

In addition to the added processing required to perform omniscient debugging, AODB also requires a history of previous changes that have occurred within the transformation. The history is required for the debugger to return to any previous state of the transformation, but comes at the cost of additional memory consumption. Both *RQ 5* and *RQ 6* are designed to assist in understanding the overall impact that history has on memory consumption.

History is composed of three main components. The first is the changes that occur during the transformation. The second is the steps or the discrete units of execution that contain the changes. Finally, the revisions cache contains a link between the elements in the model and the changes in history. Additional minor elements are also included in history (i.e., a pointer to the current step), but these elements provide only static memory usage and are trivial compared to the three main components. To investigate the impact of these three components, we mapped out the percentage of history that is composed of the changes and the percentage that is composed of the steps as we progress through the model transformation and the number of executed steps increases. The revisions cache stores an entry for each change that has occurred. We can safely consider only these two concerns because the cost of the revisions cache can be amortized among all changes to produce a constant increase per change. Thus, the primary component is the number of changes, because the memory usage of the revisions cache is dependent upon the number of changes. Furthermore, as a result of the study, we observed that the main contributing component to the memory consumption of history for the observed systems (Sierpinski Triangles and Movie DB Case) is due to the changes. This was expected because steps are a containing unit of the changes and each step has an almost constant memory consumption.

There are several points during execution when we observed notable reductions in the percentage of history's memory usage due to the changes (as seen in both Figs. 14 and 15). These reductions are observed in both model transformations. For the Sierpinski Triangles, the model transformation is written in T-Core which maintains an internal store of dynamic information which includes a set of matched sets (i.e., subgraphs matched by the matcher rule, `findTriangles`). As the transformation progresses through the remainder of the generation, the iterator removes elements from the matched set, and the rewriter uses the removed match set to process a triangle. Over the course of the generation, the matched sets slowly decrease and the total number of changes steadily increases. At the beginning of the next generation, a new (significantly larger) matched set is generated. Because history stores step information such as the matched sets, the percentage of history will vary based on the current point in the generation. The Movie DB Case, however, appears to only have a single transition rather than repeated transitions from continuous searches, but the reduction in this case is due to the same concern. Internally, MoTif rules possess a similar structure to the matched sets in T-Core that are used for the same purpose. However, the Movie DB Case has a constant series of searches after the initial building phase of Task 1. Thus, when Task 1 completes, the transformation will then continuously have a matched set component that does not slowly reduce in size as seen in the Sierpinski Triangles transformation. Thus, the result is a single reduction at the point where Task 1 ends, but the reduction is due to the same cause as observed for the Sierpinski Triangles.

7.6 What is the impact of history on total memory consumption? (*RQ 6*)

RQ 5 focused on the composition of history to determine the main components impacting memory consumption. However, this question did not address the impact of history on overall memory usage for the transformation engine. *RQ 6* addresses the effect of history on the memory consumption of the model transformation engine as a whole.

For each model transformation, we mapped the percentage of the model transformation engine's memory that is accounted for by the history as the number of executed steps increases. For the Movie DB Case transformation, this value never surpassed 5 % of memory consumption and displayed a slow growth rate. In contrast, the Sierpinski Triangles transformation grew at a roughly logarithmic rate as the number of executed steps in the transformation increased. Additionally, the Sierpinski Triangles transformation has several observable sharp increases in memory usage which are due to the matched sets generated at the beginning of generation (as discussed in the previous section). However, even with this growth rate and the sharp increases, the memory usage never surpassed 15 % of the overall memory usage for the model transformation engine.

This would seem to indicate that there is still considerable room for the history to grow, allowing for a larger number of model elements and changes. Furthermore, we can observe that the size of a model and other factors seem to be contributing more significantly to memory consumption than history.

7.7 Evaluating the efficiency and scalability of our technique

Each of the previous questions addressed a different point regarding the efficiency and the scalability of the technique. The answers to these questions help address when the algorithms proposed in this paper can be used by a developer during MT debugging. Due to the non-deterministic nature of model transformations, omniscient debugging allows for the developer to debug an execution that may have been difficult to recreate without these features. However, it is important to ensure that the techniques used to provide these features avoid adding significant overhead affecting either runtime performance or memory consumption.

From the results of our study, we found that the omniscient debugging technique does not significantly affect the execution time of a model transformation when compared with stepwise execution. This is an important finding as it indicates that the developer should not notice a difference when utilizing the omniscient features. In addition, we looked at memory usage of the history structure that is required to make omniscient debugging possible. We found that history only had a minor impact on the system. The major contributors to memory usage include the size of the model and other standard features of the transformation engine. With this in mind, the dominating factor for determining memory resources to allocate is still the size of the model and not the additional storage of history.

In terms of the effectiveness of our MacroStep building algorithms, we found that there was a significant difference between iterating through the steps (`IterateSteps`) and iterating through the elements (`IterateElements`). The results of our analysis have indicated we could improve the performance by altering the cutoff points for using one algorithm over the other. We believe this will lead to significant gains when jumping through history and improve the overall omniscient debugging process.

Finally, we found that in cases where non-determinism was not a factor and a model transformation can be re-executed by the developer, it is often superior with regard to execution time to use the omniscient features. This is especially true when the

transformation incorporates costly search steps or when the number of changes grows large. As we increase the effectiveness of building the macrosteps, this difference should grow more prominent for all cases.

8 Conclusions and future work

Like all software systems, evolution also occurs in software models. In MDE, the evolution of models is commonly defined using 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. Traditional approaches to bug localization have also been applied to assist in locating errors in model transformations (Schönböck et al. 2009). Debugging is a fundamental software engineering task. However, despite the common need for debugging in software development, tool support for debugging has changed little over the past half century (Seifert and Katscher 2008).

We presented an omniscient debugging technique and associated algorithms for model transformations. The technique provides an intuitive extension to stepwise execution to enable free traversal of execution history through a set of omniscient features that control the execution of the system in a stepwise manner. We provide the ability to continuously execute or revert the system (i.e., `play` and `playback`), progress stepwise through the execution of the transformation (i.e., `stepIn`, `stepOver`, or `stepOut`), revert the system in a mirror of the stepwise features (i.e., `backIn`, `backOver`, and `backOut`), and jump directly to a target step of execution. We discussed the use of a trace of execution, referred to as history, to enable the various omniscient traversals. Finally, we introduced an algorithm to efficiently identify a minimal set of changes, MacroStep, which must be executed to complete a traversal through history (either forward or backward) along with a set of patterns used to determine which changes must be included.

We also provide discussion of the theoretical execution time performance scaling of the MacroStep building algorithm and the theoretical memory consumption scaling of history. We followed this theoretical discussion with an empirical evaluation of these concerns guided by a set of 6 research questions. The evaluation indicates execution time performance was not significantly different than a stepwise debugger when considering initial execution (where omniscient debugging must provide additional processing to manage history), and the memory scaling of the overall system was not observed to have a primary effect on the memory usage of the model transformation execution engine as a whole. Additionally, numerous features and components were explored such as comparing two approaches to building a MacroStep.

As future work, we are preparing to conduct a human-based study to evaluate the usability of AODB in practice. Thus far, AODB has been applied as a prototype designed to explore our technique with a particular concern for the efficiency of the algorithms and storage structures. We would like to extend AODB with a visualization of history and perform a formal user study to assess the usability of the tool. The study will provide a comparative analysis using the stepwise execution debugger provided in AToMPM as a baseline. The results will provide insight and feedback to further drive the development of our technique supporting omniscient debugging for model transformations as well as evaluation of the usability of AODB.

References

Agrawal, A., Karsai, G., Neema, S., Shi, F., & Vizhanyo, A. (2006). The design of a language for model transformations. *Journal of Software and Systems Modeling, 5*(3), 261–288.

Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., & Tratt, L. (2013). State-based model slicing: A survey. *ACM Computing Surveys, 45*(4), 53:1–53:36.

AToM3. (2015). AToM3 Project Page. University of McGill Modelling, Simulation, and Design Lab. http://atom3.cs.mcgill.ca. Accessed 9 Sept 2015.

Basciani, F., Di Rocco, J., Di Ruscio, D., Di Salle, A., Iovino, L., & Pierantonio, A.(2014) MDEForge: an extensible web-based modeling platform. In *Proceedings of the 2nd international workshop on model-driven engineering on and for the cloud co-located with the 17th international conference on model-driven engineering languages and systems* (pp. 66–75).

Bousse, E., Corley, J., Combemale, B., Gray, J., & Baudry, B. (2015) Supporting efficient and advanced omniscient debugging for xDSMLs. In *Proceedings of the 8th international conference on software language engineering* (accepted for publication).

Burgueño, L., Troya, J., Wimmer, M., & Vallecillo, A. (2015). Parallel in-place model transformations with LinTra. In *Proceedings of the 3rd workshop on scalable model driven engineering part of the software technologies: applications and foundations federation of conferences* (pp. 52–62).

Clinger, H., & Ost, E. (1988). Implementation strategies for continuations. In *Proceedings of the ACM conference on LISP and functional programming* (pp. 124–131).

Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J. M., & Gray, J. (2014). Globalizing modeling languages. *IEEE Computer*, 10–13.

Corley, J. (2014). Exploring omniscient debugging for model transformations. In *ACM student research competition at the 17th international conference on model-driven engineering, languages, and systems* (pp. 63–68).

Corley, J., Eddy, B., & Gray, J. (2014). Towards efficient and scalable omniscient debugging for model transformations. In *Proceedings of the 14th workshop on domain-specific modeling* (pp. 13–18).

Corley, J., Syriani, E., Ergin, H., & Van Mierlo, S. (2016). Cloud-based multi-view modeling environments. In A. M. Cruz & S. Paiva (Eds.), *Modern software engineering methodologies for mobile and cloud environments*. Pennsylvania: IGI Global.

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal, 45*(3), 621–645.

Di Ruscio, D., Kolovos, D., & Matragkas, N. (2013). Scalability in model driven engineering: BigMDE'13 workshop summary. In *Proceedings of the first workshop on scalability in model driven engineering* (pp. 1–2).

Engblom, J. (2012). A review of reverse debugging. In *System, software, SoC and silicon debug conference* (pp. 1–6).

Ergin, H., & Syriani, E. (2014). Atompm solution for the IMDB case study. In *Proceedings of the 7th transformation tool contest part of the software technologies: Applications and foundations* (pp. 134–138).

Gray, J., Tolvanen, J-P, Kelly, S., Gokhale, A., Neema, S., & Sprinkle, J. (2007). Domain-specific modeling. *Handbook of Dynamic System Modeling*, 7/1-7/20.

Henkler, S., Meyer, J., Schafer, W., von Detten, M., & Nickel, U. (2010). Legacy component integration by the Fujaba real-time tool suite. In *ACM/IEEE 32nd international conference on software engineering* (pp. 267–270).

Horn, T., Krause, C., & Tichy, M. (2014). The ttc 2014 movie database case. In *Proceedings of the 7th transformation tool contest part of the software technologies: Applications and foundations* (pp. 93–97).

IEEE. (2002). IEEE 610-12.1990 IEEE standard glossary of software engineering terminology. IEEE. https://standards.ieee.org/findstds/standard/610.12-1990.html. Accessed 19 Sept 2015.

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming, 72*(1–2), 31–39.

Jouault, F., & Kurtev, I. (2006). Transforming models with ATL. In *Satellite events at the MoDELS 2005 conference* (pp. 128–138).

Kolovos, D. S., Rose, L. M., Matragkas, N., Paige, R. F., Guerra, E., Cuadrado, J. S., et al. (2013). A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the workshop on scalability in model-driven engineering* (pp. 1–10).

Kühne, T. (2006). Matters of (meta-) modeling. *Journal of Software and Systems Modeling, 5*(4), 369–385.

Lewis, B. (2003). Debugging backwards in time. In *Proceedings of the fifth international workshop on automated debugging.*

Lienhard, A., Fierz, J., & Nierstrasz, O. (2009) Flow-centric, back-in-time debugging. In *Proceedings of objects, components, models and patterns* (pp. 272–288).
Lienhard, A., Gîrba, T., & Nierstrasz, O. (2008). Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on object-oriented programming* (pp. 592–615).
Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., et al. (2016). Model transformation intents and their properties. *Journal of Software and Systems Modeling*. doi:10.1007/s10270-014-0429-x.
Mannadiar, R., & Vangheluwe, H. (2011). Debugging in domain-specific modelling. In *Software language engineering* (pp. 276–285).
Pothier, G., & Tanter, E. (2009). Back to the future: Omniscient debugging. *IEEE Software, 26*(6), 78–85.
QVT. (2015). Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) version 1.2. http://www.omg.org/spec/QVT/1.2. Accessed 19 Sept 2015.
Schönböck, J. (2012). Testing and debugging of model transformations. Dissertation. The Vienna University of Technology. http://publik.tuwien.ac.at/showentry.php?ID=209018&lang=2. Accessed 19 Sept 2015.
Schönböck, J., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W., & Wimmer, M. (2009). Catch me if you can—debugging support for model transformations. In *Proceedings of the 12th international conference on model-driven engineering, languages, and systems* (pp. 5–20).
Seifert, M., & Katscher, S. (2008). Debugging triple graph grammar-based model transformations. In *Proceedings of 6th international Fujaba days* (pp. 19–22).
Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software, 20*(5), 19–25.
Steel, J., & Lawley, M. (2004). Model-based test driven development of the Tefkat model transformation engine. In *Proceedings of the 15th IEEE international symposium on software reliability engineering* (pp. 151–160).
Stevens, P. (2010). Bidirectional model transformations in QVT: Semantic issues and open questions. *Journal of Software and Systems Modeling, 9*(1), 7–20.
Syriani, E., & Vangheluwe, H. (2011). A modular timed model transformation language. *Journal of Software and Systems Modeling, 12*(2), 387–414.
Syriani, E., Vangheluwe, H., & LaShomb, B. (2015). T-Core: A framework for custom-built model transformation engines. *Journal of Software and Systems Modeling, 14*(3), 1215–1243.
Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., & Ergin, H. (2013). AtomPM: A web-based modeling environment. In *Joint Proceedings of MODELS invited talks, demonstration session, poster session, and ACM student research competition co-located with the 16th international conference on model driven engineering languages and systems.*
Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., & Varró, D. (2014). IncQuery-D: A distributed incremental model query framework in the cloud. In *Proceedings of the ACM/IEEE 17th international conference model-driven engineering languages and systems* (pp. 653–669).
Taentzer, G. (2003). AGG: A graph transformation environment for modeling and validation of software. In *Proceedings of the second international workshop on applications of graph transformations with industrial relevance* (pp. 446–453).
Ujhelyi, Z., Horvath, A., & Varro, D. (2012). Dynamic backward slicing of model transformations. In *IEEE fifth international conference on software testing, verification and validation* (pp. 1–10).
Van Mierlo, S. (2014). Explicit modelling of model debugging and experimentation. In *Proceedings of doctoral symposium co-located with the 17th international conference on model driven engineering languages and systems.*
Varró, D., & Balogh, A. (2007). The model transformation language of the VIATRA2 framework. *Science of Computer Programming, 68*(3), 214–234.
Zelkowitz, M. V. (1973). Reversible execution. *Communications of the ACM, 16*(9), 566.

Jonathan Corley is currently a Ph.D. student in Computer Science at the University of Alabama working in the area of model-driven engineering. He graduated with a M.S. in Computer Science from the University of Alabama in May of 2012. While working toward his M.S. and Ph.D., he has been a member of the Software Engineering Group at the University of Alabama. In addition to his research, Jonathan has taught several undergraduate courses and has also been involved with numerous outreach programs in Tuscaloosa (e.g., assisting with coordinating an after-school program at local elementary schools).

Brian P. Eddy is an Assistant Professor at the University of West Florida in Department of Computer Science. He graduated with his Ph.D. from the Department of Computer Science at the University of Alabama in 2015. He graduated from Armstrong Atlantic State University in 2009 with B.S. degrees in both Computer Science and Applied Mathematics. He then went on to receive his M.S. in Computer Science from the University of Alabama. Mr. Eddy's research interests include program comprehension, software and maintenance, and computer science education. His main areas of research include improving the state of the art in maintenance and debugging tools for large-scale software systems. In addition to his research interests, Brian has a passion for teaching. He was worked with a wide range of students in the areas of Computer Science and Mathematics.

Eugene Syriani is currently an Assistant Professor in Computer Science at the University of Montreal where he is a member of the GEODES Software Engineering Research Group. He teaches bachelor, masters, and doctoral-level courses in software engineering. Eugene's main research interests are in model-based design, in particular model transformation design and verification, model-driven methodology, simulation-based design, and application of MDE in non-computer science domains. In addition to his research and teaching, he serves on the program committee, organizes several international conferences and workshops, and is a reviewer for journals in modeling and simulation. Eugene was formerly an assistant professor at the University of Alabama until 2014. He received a Ph.D. in Computer Science in 2011 and holds a B.Sc. in Mathematics and Computer Science since 2006, both at McGill University.

Jeff Gray is a Professor in the Department of Computer Science at the University of Alabama. His research interests include software engineering, model-driven engineering, mobile computing, and computer science education, More about Jeff's research can be found at http://gray.cs.ua.edu.