Supporting Debugging in a Heterogeneous, Globally Distributed Environment

Jonathan Corley and Jeff Gray

The University of Alabama, U.S.A.
corle001@ua.edu, gray@cs.ua.edu

Abstract. Model-Driven Engineering has emerged as a software development paradigm that can assist in separating the issues of the problem space of a software system from the complexities of implementation in the solution space. As software systems have become more complex, a need for multiple abstractions to describe a single system has emerged. The development teams of these massive systems are also often geographically distributed. These emerging concerns for MDE systems have led to a need for a heterogeneous, and potentially globally distributed, modeling environment. As these modeling environments are being explored, new challenges are being uncovered. In this paper, we discuss the need for debugging support in heterogeneous, globally distributed modeling systems and identify a number of challenges related to debugging that must be overcome to support this evolving paradigm for software development.

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. MDE approaches often use customized domain-specific modeling languages (DSMLs) that capture the intent of a particular group of users through abstractions and notations that fit a specific domain of interest. Through the application of DSMLs, various stakeholders in a project are enabled to view and edit the system using an abstraction most appropriate to their needs and expertise. However, the disparate abstractions introduced can create barriers between components in the same project by separating these concerns into distinct DSMLs without the ability to describe interactions between components [1]. An electrical engineer may produce a wiring diagram while a software engineer produces a component diagram. If the two engineers are both working on a shared project then the implementation of these designs will impact each other. This scenario indicates a need for shared reasoning, analysis, and communication between these two groups to enhance the cohesiveness of the final design and resulting implementation.

As the development of support for heterogeneous, globally distributed modeling environments progresses, a key concern is what support is expected in these new environments. Debugging is a common task that all software developers encounter across different software artifacts [2]. Though debugging has been a consistent aspect of the software development process, debugging tool support has changed little over the past half century [2]. Several novel approaches to debugging have been introduced in the research literature, such as omniscient debugging [3] and query-based debugging [4,5,6], each claiming to improve efficiency and effectiveness of developers. However, commercial tool support available to programmers focuses primarily on stepwise execution of code, typically with break-points [5]. MDE tools are typically less mature than tools available for traditional general-purpose programming languages (GPLs). With respect to MDE research, Mannadier and Vangheluwe observed there has been little concern for the state of debugging support [7]. However, over a decade ago, Bran Selic commented that if developers are not satisfied with the "day-to-day" application of MDE, then MDE will be rejected by developers [8]. One of the most common tasks undertaken by software developers is debugging. Therefore, we believe improved debugging support will aid adoption of MDE, especially for complex systems described by multiple DSMLs.

2 Background and Related Work

Like all software systems, evolution also occurs in software models. In MDE, the evolution of models is commonly defined using model transformation languages (MTLs), which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level. Model transformations are also a type of software abstraction that can be subject to human error. Thus, traditional approaches to bug localization may also be applied to assist in locating errors in model transformations.

2.1 Stepwise Execution with Breakpoints

The most commonly implemented debugging approach is stepwise execution, which enables the developer to observe hidden state information dynamically during execution. A stepwise execution environment generally possesses the following features: play, pause, stop, and step. Play allows for continuous execution; pause suspends execution at the current state; stop terminates execution. Most tools support three types of step (*i.e., stepOver*, *stepIn*, and *stepOut*) enabling developers to incrementally progress the execution. Numerous MDE tools (including TROPIC [4], GReAT [9], ATL [10], and more) provide support for stepwise execution.

2.2 Query-based Debugging

Query-based debugging (QBD) promotes the use of queries to aid a developer in locating the source of a bug. These queries aid a developer in gaining a better understanding of the underlying system. Most QBD techniques have promoted the use of a rich, expressive query language [4,6]. However, some approaches focus on a smaller range of query options. The Whyline [5] focuses on queries designed to lead from a point of error to the fault that generated the observable error. These queries are termed "why did" queries (*e.g.,* "why did attribute x have value y?") [5]. QBD has been empirically evaluated and demonstrated to improve developer efficiency and effectiveness during debugging tasks [5,6].

We are only aware of one MDE tool, TROPIC [4], that supports query-based debugging. TROPIC provides an interactive query console that enables developers to specify OCL queries. TROPIC supports debugging by converting transformations to a Transformation Net (TN), a specialized colored Petri-net. TROPIC supports many modeling languages, but converts all models to a common TN representation. TROPIC is intended for a single developer and does not support collaborative distributed development.

2.3 Omniscient Debugging

We are not aware of any MDE tool supporting omniscient debugging, which can be considered an extension of stepwise execution that provides the ability to traverse back through the execution of a debugging session dynamically at run-time. Current work in the area of omniscient debugging is focused on GPLs. However, the technique would also be beneficial in the MDE context. MTs are a software abstraction subject to error similar to GPLs [3]. An error may manifest at a point later in execution than the source of the defect, the fault. A concern common to both an MDE and GPL context is the time and effort required to reach a portion of the system's execution that exercises the defect. The developer may target the location of an error and thereby miss the location of the fault. In a traditional stepwise execution environment, the developer would need to restart the system and target a new location. This process of restarting to inspect new target locations may even be repeated multiple times. Restarting may require a nontrivial amount of time to reach the desired location, and may require significant manual input from the developer. Omniscient debugging enables full traversal of the execution history thereby eliminating this concern.

3 Challenges and Potential for Global Debugging Tool Support

Though the various debugging techniques can be applied to an MDE context, the application of these techniques to a globally distributed, heterogeneous modeling system brings new challenges that must be overcome. In this section, we discuss challenges unique to this paradigm.

3.1 Supporting an Extensible Debugging Environment

In a heterogeneous modeling environment, the underlying system natively supports a variety of DSMLs. This requirement is not a consideration in GPL design where a single language is supported. The more versatile MDE system must be able to represent information in the most appropriate formalism. However, the developers of these systems cannot always anticipate the unique concerns and features of future developers. The issue of supporting debugging for the variety of DSMLs available encounters similar concerns. To address these concerns, a heterogeneous modeling system must be extensible, and may require developers to provide the debugging support appropriate to their formalism with no support from the underlying tool. However, this creates a scenario where future developers will often duplicate effort for common concerns. An alternative is to provide an extensible base of support that future developers may extend.

This base of support should handle simple concerns including the ability to manually control execution of the system, visualize and explore state information, and collect state information as the system progresses. Providing these three basic features enables stepwise execution, omniscient debugging, and query-based debugging. An extension to these that would enrich debugging tool support is the ability to reload transformations and alter model elements at runtime. This extension would enable a developer to freely explore and modify the system in order to identify faults and test alterations.

A heterogeneous modeling environment may also enable the execution of multiple MTLs providing a range of potential features. Consider a system that includes a distinct MTL for both inplace and outplace transformations. These scenarios each create a unique concern for debugging support. The inplace transformation displays a single model (or set of models), but the outplace transformation must provide facilities for identifying input and output models. If providing traceability features, the inplace transformation would link elements from a previous version of the same model(s), whereas the outplace transformation would link elements from input model(s) to output model(s). These differences can lead to a varied implementation of the same concerns, but a heterogeneous environment may need to consider either or both. Such an environment must also enable defining the basic characteristics of stepwise execution (*e.g.,* what constitutes a step or scope). A step in a GPL is a statement. However, MTLs are typically defined by rules. A rule may be a simple graph transformation or a more complex component that can contain other rules (*e.g.,* ATL pre and post conditions [10]). The definition of step may therefore occur at various levels of granularity. Future designers should be able to define precise semantics for this concern that match most closely with the intended MTL environment. Similarly, defining what constitutes a scope is vital to a stepwise execution environment.

3.2 Supporting Many Formalisms in a Consistent Debugging Interface

Query-based debugging (QBD) is a promising debugging technique that provides developers with facilities to ask questions directly to the system. Query languages are typically strongly coupled with the target language. The target language may have concepts of classes and inheritance or functions and return types. However, in a heterogeneous modeling environment the target language is not fixed. The system allows (and encourages) developers to use a wide array of DSMLs to facilitate the goal of using the most appropriate abstraction. This design results in a system that may or may not possess a vast and varying set of concepts and structures. Applying QBD in a heterogeneous environment must therefore provide some design to handle the variability of DSMLs without the designers of the system making any assumptions regarding the specific terminology and structures available to future developers. This concern is further exacerbated by the inclusion of graphical query languages.

A modeling system is not always designed with the intention to limit the system to a single view. The system may include multiple metamodels and even multiple concrete syntax for the same metamodel. For example, a vehicle may contain many varied subsystems such as brakes, steering, power windows, blinkers, and many more. A vehicle is also a single unit where many subsystems have a direct impact on others. A prime example is how electrical wiring directly impacts the functioning of the power window subsystem. If the power windows exceed the capacity for the electrical wiring system then the windows will fail to open and close properly. However, these two systems may be designed using different DSMLs and in a typical modeling environment would be in separate models. However, a heterogeneous modeling environment would enable developers to view these systems either together or separately as needed. This leads to further concerns when applying QBD. If the developer of the car system were to pose a query such as “why did the power window not rise?” the system may need to search through models defined using several DSMLs to provide the answer. However, current work in the area of QBD has always assumed a single language and there is no existing technique that is concerned with searching across multiple languages.

A primary concern for omniscient debuggers is the collection of trace information required to revert the system to previous states of execution. This collection of trace information forms a history of execution for the system. In a heterogeneous modeling environment, the collection of trace information is complicated by the varying structures. An omniscient technique must collect the smallest units of information for each modification in order to minimize the space consumption of the history structure. However, an omniscient debugger must also collect information relevant to the structure of the model in order to ensure proper application of any change. For example, assume a model element ‘a’ relies on model element ‘b’ and both are deleted. When reverting the delete operation, the system must ensure there is never a state where ‘a’ exists without ‘b’ to avoid violating constraints of the modeling environment. Similarly, if element ‘a’ is always altered to match any modification in ‘b’, the underlying execution environment may capture these modifications independently, but upon replaying these modifications may incur an additional modification to element ‘a’ (both when ‘b’ is reverted causing ‘a’ to be automatically altered and when ‘a’ is directly altered to revert the recorded modification). However, these structural concerns may vary depending on the specific formalisms used.

3.3 Supporting Debugging in a Globally Distributed Environment

A global software engineering system is concerned with geographically distributed development teams. A natural implication of this environment is that developers will utilize separate physical machines even while actively collaborating. However, the implications of separate machines accessing a common environment can be more subtle. The typical debugging environment includes a single machine and therefore assumes a single point of control for the debugging environment. The introduction of more points of access to control the environment creates a more complex scenario. Consider the following scenario: James is inspecting the state of execution at a specific point, and Elizabeth (unaware of James’s intent) progresses the transformation to a state she is interested in investigating. In this scenario, James and Elizabeth are each intent on exploring distinct points in history. A simple solution is to provide facilities for James to express his intent to explore the current state and even possibly lock the system to the current state. However, this solution restricts Elizabeth from following her own thread of investigation. A global omniscient debugging system could also offer facilities for both parties to independently explore the system while simultaneously enabling the two developers to share an environment. Thus, Elizabeth and James could work both collaboratively on a single issue and in parallel on separate related issues within the same environment.

4 Conclusion and Future Work

We have discussed the current state of debugging for MDE. We then summarized the challenges presented when applying these techniques in a globally distributed, heterogeneous modeling environment. Several challenges were discussed for each technique. Some challenges were caused by the application of many formalisms to describe a single system, contrasting with the existing work that typically focuses on a single formalism. Other challenges were the result of applying these techniques to a distributed environment. In these environments, the system must support both collaborative processes where models and control is shared for a common goal, and parallel processes where both models are shared and control is independent simultaneously. These concerns were presented and discussed, but left as future work to solve. We believe debugging support is a requirement in a modern software development environment and these issues are therefore of vital concern for the future of this growing and evolving paradigm.

References

1. Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J.M., Gray, J.: Globalizing Modeling Languages. Computer (June 2014) 10–13
2. Mirko Seifert, Stefan Katscher: Debugging triple graph grammar-based model transformations. In: Proceedings of 6th International Fujaba Days, Dresden, Germany (2008)
3. Adrian Lienhard, Julien Fierz, Oscar Nierstrasz: Flow-centric, back-in-time debugging. In: Proc. of Objects, Components, Models and Patterns, Zurich, Switzerland (2009) 272–288
4. Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, Manuel Wimmer: Catch me if you can – debugging support for model transformations. In: Proc. of 12th Int'l Conf. on Model-Driven Engineering, Languages, and Systems, Denver, CO, USA (2009) 5–20
5. Andrew J. Ko, Brad Myers: Debugging Reinvented: Asking and answering why and why not questions about program behavior. In: Proceedings of the 30th International Conference on Software Engineering (ICSE '08), Leipzig, Germany (2008) 301–310
6. Lencevicius, R., Hölzle, U., Singh, A.: Dynamic Query-Based Debugging of Object-Oriented Programs. Automated Software Engineering **10** (2003) 39–74
7. Mannadiar, R., Vangheluwe, H.: Debugging in domain-specific modelling. In: Software Language Engineering. Volume 6563 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 276–285
8. Selic, B.: The pragmatics of model-driven development. IEEE Software **20**(5) (2003) 19–25
9. Agrawal, A.: Graph rewriting and transformation (great): a solution for the model integrated computing (mic) bottleneck. In: Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. (Oct 2003) 364–368
10. Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 128–138