EXPLORING EFFICIENT AND SCALABLE OMNISCIENT DEBUGGING FOR MDE

by

JONATHAN CORLEY

JEFF GRAY, COMMITTEE CHAIR
JEFFREY C. CARVER
RANDY SMITH
SUSAN VRBSKY
EUGENE SYRIANI

A Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2016

ABSTRACT

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 a particular solution space of implementation. 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. In MDE, the execution and evolution of models is commonly defined using model transformation languages, which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level, or defining directly executable models. Like more traditional software development artifacts, both model transformations and executable models are subject to human error, and common software engineering practices (*e.g.,* debugging) are still prevalent in MDE.

The primary thrust of the work described in this dissertation concerns investigating the application of omniscient debugging in an MDE context. The work also explores how developers form and use queries during debugging tasks, thereby promoting a better understanding of the types of data developers seek during debugging tasks and how tool support can assist developers during these activities. Furthermore, as distributed development becomes ever more prevalent, I explored supporting collaborative development processes (*e.g.,* paired debugging) in a distributed modeling environment. The work presented in this dissertation has impacted the MDE community by developing new applied techniques and provided an improved understanding of the process used by developers during debugging tasks.

DEDICATION

To my best friend and wife, Stephanie Corley. You have pushed me to be a better man, and to never compromise my dreams. I look forward to all that the future has in store for us and our family.

To my brilliant, exasperating, and perfectly imperfect children; Elizabeth, James, and Anastasia. I strive daily to be a man that you are proud to know as your father.

LIST OF ABBREVIATIONS

API	Application Program Interface
CIM	Computation Independent Model
CoV	Coefficient of Variation
CPU	Central Processing Unit
CRUD	Create, Read, Update, and Delete
DSML	Domain-Specific Modeling Language
EMF	Eclipse Modeling Framework
FIFO	First In, First Out
fUML	Functional Unified Modeling Language
GB	GigaByte
GHz	Giga Hertz
GPL	General-Purpose Language
GUI	Graphical User Interface
H0	Null Hypothesis
HA	Alternative Hypothesis
HTML5	HyperText Markup Language version 5
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LAN	Local Area Network

LHS	Left-Hand Side
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
MoTif	Modular Timed Graph Transformation Language
MT	Model Transformation
MTL	Model Transformation Language
MvK	Modelverse Kernel
NAC	Negative Application Condition
OCL	Object Constraint Language
OO	Object-Oriented
PIM	Platform Independent Model
QBD	Query-based Debugging
QVT-O	Operational Query/View/Transformation Language
RAM	Random Access Memory
REST	Representational State Transfer
RHS	Right Hand Side
SE	Software Engineering
UML	Unified Modeling Language
V&V	Verification and Validation
xDSML	Executable Domain-Specific Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

ACKNOWLEDGMENTS

First, I would like to acknowledge my wife, Stephanie Corley, who has motivated me to achieve more than I could have dreamed possible. Her love and support over the past 12 years, and especially as I undertook to earn a Ph.D., has been invaluable.

I would like to thank my parents for all of the support and guidance they have provided me. I remember many conversations with both my father and mother concerning their experiences, and the wisdom gained from their honest discussions with me of both their successes and failures has shaped me into the man that I am today.

I would like to thank Dr. Jeff Gray for his guidance and advise throughout my dissertation work. Dr. Gray has been an incredible mentor throughout my dissertation work. His mentorship has guided me in this research, but also he has afforded me the opportunity to explore teaching and outreach activities that have been the highlights of my time at the University of Alabama.

I would also like to thank my friends and colleagues, Dr. Brian Eddy and Dr. Dustin Heaton. They have been my sounding board, first round reviewers, collaborators, and party members. The time spent with both of them collaborating on research and academic projects or sitting around a table slaying dragons has been, and will continue to be, some of the best times of my life.

I would also like to thank Dr. Nicholas Kraft for his mentorship and support during my initial years in graduate school pursuing my Masters. It was the experiences during these years that led me to pursue a Ph.D. and a career as a professor. The time spent working with Dr. Kraft

assisting with courses, research, and outreach activities was a critical time of my life that has set me on the path that I now pursue fervently.

I would also like to thank Dr. Eugene Syriani for his mentorship and the opportunities he has afforded me over the past several years. Dr. Syriani has been a great mentor, teacher, and collaborator for me during my graduate studies, and much of the work presented in this dissertation could not have been completed without the opportunites that he has made available to me over the past few years.

I would also like to thank my Committee members: Dr. Jeffrey Carver, Dr. Susan Vrbsky, and Dr. Randy Smith, who have provided me with invaluable advise and guidance in my research and truly memorable and engaging learning experiences in their courses. All that I have learned from each of you over the past few years, I will carry forward with me into my career.

The work presented in this dissertation has been supported by the NSF CE21 grant, and previously by a Department of Education GAANN fellowship.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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 a particular solution space of implementation. MDE approaches often use customized domain-specific modeling languages (DSMLs) that capture the intent of a particular group of end users through abstractions and notations that fit a specific domain of interest [3]. In MDE, the execution and evolution of models is commonly defined using model transformation languages (MTL), which can be used to specify the distinct needs of a requirements or engineering change at the software modeling level, or defining directly executable models. Like more traditional software development artifacts, both model transformations (MTs) and executable models are subject to human error, and common software engineering (SE) practices (*e.g.,* debugging) are still prevalent in MDE.

The majority of the existing work concerning debugging (*i.e.,* the process of locating and correcting bugs) has focused on the application of debugging approaches to General-Purpose Languages (GPLs), such as C and Java. A wide variety of debuggers, tools and techniques that aid developers in the process of debugging have been created, studied, and evolved. A variety of distinct approaches have been introduced including the traditional step-through debugging approach, as well as more advanced approaches, such as Omniscient Debugging (also referred to as Back-In-Time or Reverse Debugging) and Query-Based Debugging (QBD). Over the course of conducting the research described in this dissertation, I have investigated the application of omniscient de-

bugging to both a model transformation environment and an executable modeling environment, explored how developers debug with special emphasis on the impact and formation of queries, and investigated how to support collaborative processes in a distributed modeling environment. Through this work, I hope both to impact the MDE community by developing a new applied techniques and to generate a better understanding of the debugging process used during debugging tasks by developers.

1.1 Tool Support and Debugging for MDE

The primary thrust of my research has been related to debugging support for MDE. Debugging is a common task that all software developers encounter across different software artifacts. Seifert and Katscher [4] state that, "the search for defects in programs has become a common activity of every software developer's life." Despite the longstanding need for debugging support, the state of tool support for debugging tasks has changed little over the past half century compared to other advances in development methodologies and techniques [4, 5]. There have been several novel research approaches to debugging, such as omniscient (or back-in-time debugging) [6–10] and QBD [5, 11–15]. However, commercial debugging tools available to mainstream developers are focused primarily on stepwise execution of code and utilization of breakpoints [5]. The tools available for MDE are not exceptional in this regard, and are in fact less mature than tools available for traditional GPLs.

Mannadier and Vangheluwe [16] have observed, "very little attention has been paid to debugging" in the context of MDE. An investigation into the state of debugging support for MTs revealed only one (experimental) MDE tool called TROPIC [17–19] that offers capabilities be-

yond stepwise execution and breakpoints. Several MDE tools (*e.g.,* GReAT [1] [20], ATL [2] [21], TEFKAT [3] [22], UML Model Debugger [4], AToM3 [5], AToMPM [23–25], VIATRA2 [6] [26], AGG [27], and Fujaba [7] [28]) provide basic debugging support in the form of stepwise execution facilities; GReAT, ATL, TEFKAT, and UML Model Debugger also provide support for breakpoints.

A central goal of MDE is to improve developer productivity. This goal is obtained primarily through raising the level of abstraction away from the solution domain by focusing on models and model transformations that focus on the problem domain. Despite the focus on models and model transformations as opposed to GPLs, traditional development concerns such as debugging must still be undertaken by developers adopting MDE practices. Bran Selic commented that if developers are not satisfied with the "day-to-day" application of MDE then MDE will be rejected [29]. Because one of the most common tasks undertaken by software developers is debugging, improved debugging approaches for model transformations have the potential to improve the acceptability of MDE from an industrial perspective and aids in attaining the goal of improved developer productivity.

1.2 Supporting Debugging in a Heterogeneous, Distributed Environment

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 DSMLs that capture the intent

[1] www.isis.vanderbilt.edu/tools/GReAt
[2] www.eclipse.org/atl/
[3] tefkat.sourceforge.net
[4] http://www.research.ibm.com/haifa/projects/services/uml/index.shtml
[5] atom3.cs.mcgill.ca
[6] wiki.eclipse.org/VIATRA2
[7] www.fujaba.de

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 [30]. 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, 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 [4].

1.2.1 Challenges and Potential Impact for Distributed Debugging Tool Support

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

1.2.1.1 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 vari-

ety 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 features 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 [31]). 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.

1.2.1.2 Supporting Many Formalisms in a Consistent Debugging Interface

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 `X` relies on model element `Y` and both are deleted. When reverting the delete operation, the system must ensure there is never a state where `X` exists without `Y` to avoid violating constraints of the modeling environment. Similarly, if element `X` is always altered to match any modification in `Y`, the underlying execution environment may capture these modifications independently, but upon replaying these modifications may incur an additional modification to element `X` (both when `Y` is reverted causing `X` to be automatically altered and when `X` is directly altered to revert the recorded modification). However, these structural concerns may vary depending on the specific formalisms used.

1.2.1.3 Supporting Debugging in a Distributed Environment

Geographically distributed development teams necessitate some manner of distributed SE system. This system could be managed manually, but would incur significant cost of effort and risk of error. A preferable solution would be a software system designed to manage SE projects with a distributed (or cloud) environment. 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 additional 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 distributed omniscient debugging system could also offer facilities for both parties to explore the system independently 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.

1.2.2 Supporting Distributed Debugging

As distributed development becomes increasingly prevalent, need is increasing for modeling environments that support collaborative processes. The distributed environment must support sharing a consistent model view and editing/transformation control among potentially many devel-

opers. Furthermore, because modelers often work within projects including many discrete domains of interest, the supporting environment must provide a consistent environment across the various domains of interest for a project that can also be extended to support new domains. Chapter 3 presents an investigation into supporting such a modeling environment. The contributions of the chapter begin with exploring a set of collaboration scenarios detailing how modelers could share and interact within the modeling environment. Next, the architecture of AToMPM is presented which supports multi-view, multi-paradigm modeling within a cloud-based architecture. Finally, Chapter 3 provides an initial evaluation of the architecture of AToMPM with a series of experiments designed to reflect the collaboration scenarios. Thus, the contributions address the challenges of supporting an extensible modeling environment and many formalisms in a consistent environment. Chapter 4 then presents the results of an investigation into omniscient debugging of MTs within AToMPM. Thus, the combined contributions of these chapters provide support for debugging in an extensible, consistent, distributed modeling environment.

1.3 Structure of the Dissertation

The overall theme for the dissertation has been introduced, and the current lack of research in this area and need for debugging support and support for a distributed, heterogeneous modeling environment has been motivated citing key literature from the MDE community. Chapter 2 will survey existing work and introduce relevant concepts in MDE and debugging. Chapters 3-6 will present the research projects comprising the core dissertation research. Finally, Chapter 7 will provide concluding remarks and outline further avenues for research.

The remainder of this section briefly overviews the projects comprising my dissertation.

1.3.1 Collaborative Modeling

In this project, I worked as part of a team of researchers spanning the University of Alabama, the University of Montreal, and the University of Antwerp to investigate supporting a cloud-based collaborative modeling architecture [24, 25]. This work has explored how to support modeling for projects distributed across many artifacts and with team members geographically distributed. This work addresses many of the debugging concerns discussed in Section 1.2.1.

1.3.2 Omniscient Debugging for MDE

This project investigated omniscient debugging, explored previously for GPLs, within the new context of MDE. This work explores efficient trace-based techniques to support omniscient traversal (*i.e.,* either forward or backward in execution history). The technique has been applied in a graphical modeling environment supporting MT [32–34], AToMPM[8], and a textual/graphical modeling environment supporting executable domain-specific modeling [35], GEMOC[9]. The investigation of omniscient debugging for MTs within AToMPM is presented in Chapter 4. The investigation of omniscient debugging for xDMSLs was part of a collaborative research project with researchers from INRIA/IRISA and the University of Rennes. The collaborative work presented in Chapter 5 extends the traversal techniques introduced in Chapter 4 and introduced a relevant visualization technique supporting omniscient debugging.

1.3.3 How Developers Debug

The final contribution of this dissertation is an exploratory study investigating the formation and use of queries during debugging tasks in an object-oriented system [36]. This study follows from existing work (as discussed further in Chapter 6). The study provides insight into the type of

[8] http://www-ens.iro.umontreal.ca/ syriani/atompm/atompm.htm
[9] http://gemoc.org/ins/

queries used by developers and how queries impact the debugging process. Thus, a key outcome of the study is empirical evidence informing future debugging tools and techniques. Furthermore, the study establishes a base study of GPL environments enabling comparisons to a future related study to be performed using a model transformation system.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter introduces MDE including models, metamodels, model transformations, and executable models. Then, the chapter discusses several debugging techniques and explores the concerns and challenges to providing debugging for MDE.

2.1 Model-Driven Engineering

MDE is a subfield of SE focused on the development of software utilizing modeling techniques. Mens and Gorp describe MDE as "a discipline in software engineering that relies on models as first-class entities and that aims to develop, maintain and evolve software by performing model transformations" [37]. This description provides two basic components of MDE: models and model transformations.

2.1.1 Model: The Object of MDE

To understand MDE, a proper definition of the notion of what is meant by *model* in this context is required. Bézivin [38] illustrates the importance of models in MDE by comparing the principle "Everything is an object" to the comparable concept in MDE that "Everything is a model." Bézivin and Gerbé state that a model is a simplification of a system with a specific intention and that the model should be usable in place of the fully realized system for the given intention [39]. Kühne [40] provides a similar definition for model, "A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made." Czarnecki and Helsen [41] describe an example of a civil engineer creating a model of a bridge to test the bridge

before generating the physical bridge. The model does not possess the full complexity and expense of the physical bridge, but rather provides an abstraction that can be used to test certain properties of the bridge. The model of the bridge could be utilized to predict the structural capacity of the bridge, but does not contain the full complexity and cost of the physical bridge.

Just as high-level languages provide abstractions from the accidental complexity of 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, and bring development closer to the problem domain [3]. MDE achieves this by considering model abstractions that are focused purely on the essential characteristics of the system. The Model-Driven Architecture (MDA) concepts of a platform-independent model (PIM) and computation-independent model (CIM) illustrate this process. A PIM is developed independent of any platform-specific implementation details (*e.g.,* details of a Java library or Windows specific threading implementation). A CIM is similarly independent of any computation-specific details (*e.g.,* details of how a random item from a collection is selected) [42]. The platform and computation-specific details are incorporated into the model through the use of forward engineering model transformations that incorporate the concepts defined across the two models [42]. The incremental incorporation of the solution domain concerns enables developers to focus on the problem domain during design and early implementation stages. In addition, forward engineering model transformations also enable a PIM to be deployed to many platforms by varying the platform-specific details to be incorporated.

2.1.2 Introducing Meta-Models and Meta-Meta-Models

Models are an abstraction of a system, but the model is also a system that may be modeled. The model of a model is a metamodel, and the model of a metamodel is a meta-metamodels [42]. The Object Management Group (OMG) (an international organization focused on defining stan-

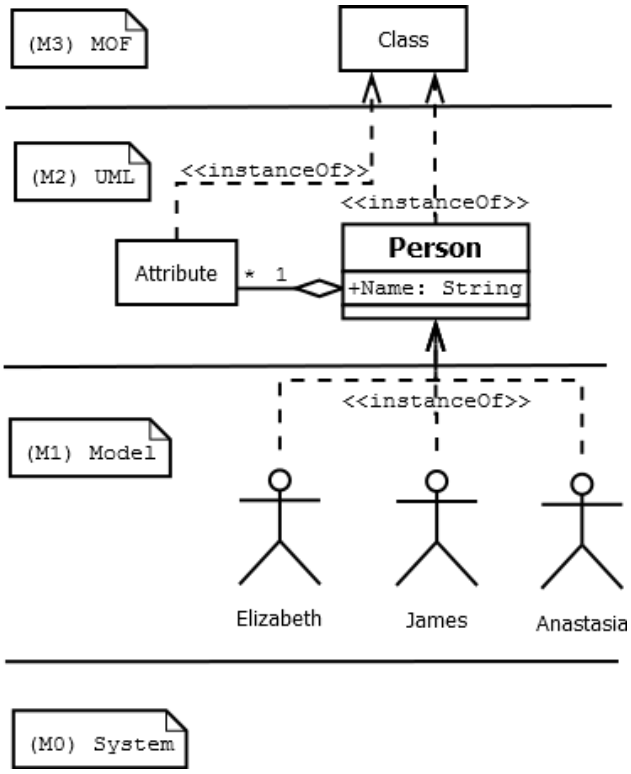


Figure 2.1: Example of four layer meta-model hierarchy

dards for software development practices and protocols) advocates a four layer metamodel hierarchy [42] (as illustrated in Figure 2.1) in the MDA [42]. In the four layer metamodel hierarchy, the top level (M3) contains the meta-metamodel. The M2 level contains any metamodels that are an instance of the meta-metamodel defined at the M3 level. The M1 level contains any user-defined models that are an instance of one of the metamodels at the M2 level. The M0 level represents a system that is modeled by user-defined models at the M1 level.

The common definition of the correspondence between model, metamodel, and meta-metamodel is the *instanceOf* relationship[1]. A model is an *instanceOf* a metamodel, which is then an *instanceOf* a meta-metamodel [40]. A metamodel may be the definitive source for an infinite number of models (in the same sense that a grammar is the definition for an infinite number

[1] The conformantTo or conformsTo relationship is also discussed in existing literature [40]. To simplify discussion, only the *instanceOf* relationship utilized by the OMG standards [42–45] is defined here.

of programs), but a model is an *instanceOf* exactly one metamodel. Similarly, a meta-metamodel can be used to define any number of metamodels, but a metamodel is an *instanceOf* exactly one meta-metamodel [46]. One of the most common metamodels is the Unified Modeling Language (UML) [43, 44], which can be used to model software and systems using object-oriented (OO) abstractions. The modeling concepts defined in UML are instantiated from UML's meta-metamodel, the Meta Object Facility (MOF) [45], which is the OMG standard meta-metamodel intended to provide a common core for tool support. The concepts within the MOF meta-metamodel can be used to define itself, which is why the reflective tower stops at the level of meta-metamodel. However, other architectures might extend beyond four levels to have higher level metamodels.

2.1.3 A Brief Introduction to MTs

MTs are core operations that drive evolution and maintenance within the MDE paradigm. An MT converts source model(s) to target model(s) by following a set of rules. MTs may either be outplace (source model(s) and target model(s) are distinct sets) or inplace (source and target model(s) are the same set). The rules may be purely imperative, such as in QVT-O [47]; purely declarative, such as in a graph transformation [41]; or combine imperative and declarative elements, such as in ATL [48]. 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 [47].

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 [41]. The combination of

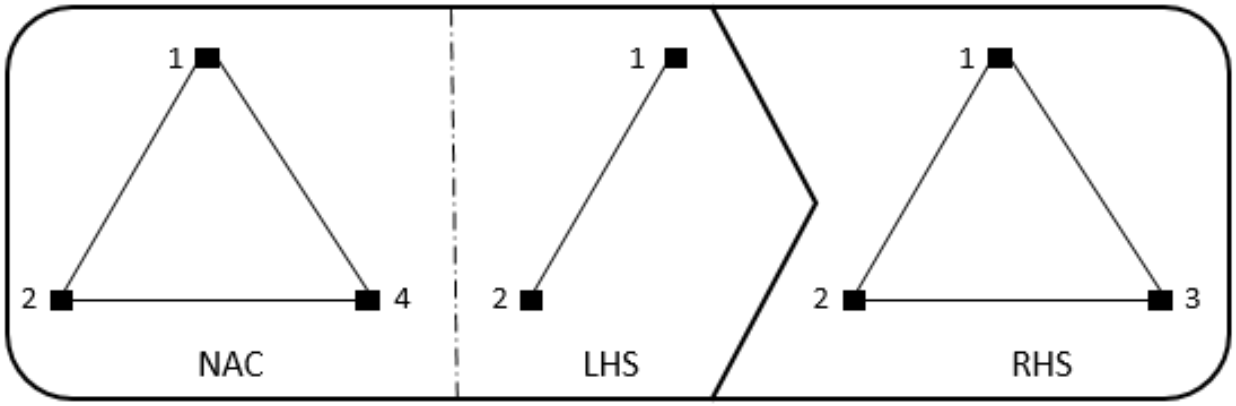


Figure 2.2: Sample Graph Transformation Rule

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), which specifies a constraint that, if true, prevents execution. Figure 2.2 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 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 [49]) 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 is potentially nondeterministic.

2.1.3.1 Nondeterminism in Model Transformations

Nondeterminism is a key feature of hybrid and declarative model transformation languages (MTLs), but not found commonly in GPLs. Scheduling of rules (*i.e.,* how the order of rule application is decided at runtime) may be nondeterministic [41]. Nondeterministic 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 varying the order of rules can produce incorrect or even invalid 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 nondeterministic 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 nondeterministic selection of model elements when executing a transformation rule [41]. The LHS of a graph transformation may match many distinct sets of elements, but the order in which these sets are chosen is often nondeterministic. 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 nondeterministic system. Thus, while nondeterministic 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 that can only progress forward. However, an omniscient debugger that can progress forward or back through execution provides an ideal solution to this concern. The nondeterministic behavior is captured by the 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 nondeterministic decisions made by the underlying execution engine.

2.1.3.2 Bidirectional Transformations

An interesting feature of MTs 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) [50]. 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 [50]. Additionally, transformation rules are not always defined as bidirectional. Thus, a system (*e.g.,* a debugger) built assuming bidirectional transformations would be required to derive an inverse for each rule. However, an inverse rule is not always possible. Consider a model transformation rule that deletes a model element. Because 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, while bidirectional transformation is possible, a general solution supporting model transformation should not rely on bidirectional transformation.

2.1.4 Defining Executable Models

The purpose of metamodeling is to define languages. *Executable* metamodeling also includes defining execution semantics within the language definition. This is typically practiced in

a domain-specific environment with xDSMLs, languages that include the definitions of the *execution state* of a model conforming to the language and *execution semantics* that operate on this state. Given an xDSML, executing a model conforming to the xDSML consists of the application, on demand, of the transformation rules that define its operational semantics. An *xDSML* is defined by:

- An *abstract syntax* is a metamodel. An *immutable property* (*i.e.,* property defining a value that will not vary during execution) can be introduced in this metamodel. At the model level, an *immutable field* (*i.e.,* field with a value that does not vary during execution) is an object's field based on an immutable property.

- An *execution metamodel* extends the abstract syntax by package merge [2]. A *mutable property* (*i.e.,* property defining a value that will vary during execution) can be introduced in this metamodel. In a model, a *mutable field* (*i.e.,* field with a value that does vary during execution) is an object's field based on a *mutable property*.

- *Operational semantics* are a set of rules that modify a model conforming to the execution metamodel by changing values of mutable fields and creating/destroying instances of classes introduced in the execution metamodel.

- An *initialization function* that, when given a model conforming to the abstract syntax, returns a model conforming to the execution metamodel.

[2] Note that in practice, existing tools and approaches use different, but similar, extension mechanisms; *e.g.,* Kermeta [51] uses aspect weaving and xMOF [52] uses generalization.

2.2 Debugging Techniques

Hibberd, Lawley, and Raymond [53] define the debugging process as a combination of the following three steps:

1. Spotting the Bug: Identifying the existence of a bug.

2. Finding the Bug: Localizing the fault(s) causing the observable failure(s).

3. Fixing the bug: Making any corrections necessary to remove the bug from the system.

Hibberd, Lawley, and Raymond [53] state "traditionally, the majority of effort is spent on bug localization and the case in model transformations is no different." Over the years, many approaches to debugging have been identified (usually focused on the more mature tool environments for GPLs), including traditional stepwise debugging with breakpoints, omniscient debugging [54], and query-based debugging [13]. These approaches focus primarily on supporting the task of Finding the Bug. Seifert and Katscher [4] state "the common goal of all debugging methods is to ease the localization of errors in programs." This section briefly reviews five approaches to debugging in a general SE context:

1. Utilization of Language and Basic Tool Features to Support Debugging

2. Stepwise Execution and Breakpoint based Debugging

3. Omniscient Debugging

4. Query-based Debugging

2.2.1 Language and Basic Tool Support

As discussed in [16] and [15], a number of language features enable what Mannadier and Vangheluwe [16] term a "poor man's debugger." The following features enable developers to perform a debugging process, but do not provide the level of support of a dedicated debugging approach such as stepwise execution with breakpoints. These features focus on exact points in code, but a dedicated debugging approach supports features that enable exploration of code either dynamically (*i.e.,* at runtime) or statically (*e.g.,* during post-mortem analysis).

2.2.1.1 Print Statement

A print statement is a common language feature available in all major GPLs and can be used to output state information or track control flow at runtime [16]. This technique is very basic and requires additions to the system that must be maintained and evolved based on a need for particular debugging information not relevant to normal functioning of the system.

2.2.1.2 Assertions

Assertions enable a developer to check a specified condition (often expressed as some relational expression over the current program state) at runtime. Assertions cause the program to abort if the condition fails, where failure is determined by the result of checking a specified condition [15, 16]. Assertions are managed more easily than print statements due to common implementation of release and debug modes in languages and associated tool support [16]. In "debug" mode, assertions work as described, but in "release" mode assertions are ignored at runtime. This allows a developer to manage the addition and removal of assertions without concern for the assertions causing unexpected or unwanted behavior when released.

2.2.1.3 Exceptions

Exceptions are the most advanced language feature for debugging support. Syriani, Kienzle, and Vangheluwe [1] state "debugging a model transformation is not trivial and exceptions are very helpful for debugging, namely to detect logical errors in the design of a transformation." An exception is thrown at runtime and signifies an error during execution. Exceptions contain information concerning the state of the program at the time the error was encountered. An exception may be caught and handled by custom exception handlers [1, 16]. A developer may associate specific code in the body of an exception handler to attempt correction of the error and continue execution at the same or different point in the control flow. A developer may also abort execution and provide the state information as output to aid in a post mortem analysis of the error. If the exception is not caught by an appropriate handler, then execution will be aborted with a stack dump provided as typical output. In addition to being able to use predefined exceptions, many languages provide support for defining customized exceptions [1, 16].

2.2.2 Tracing

Traditional GPL tools offer support for stack traces that provide the developer with information concerning the current scope at a particular time during program execution [16]. Stack traces are commonly provided during stepwise debugging processes (discussed later) and as a part of the information included by exceptions. In addition, traceability links have been utilized to aid in a variety of tasks including verification and validation (V&V) such as requirements traceability [55]. The concept of tracing follows the trail of associations between two software elements that can be either design or implementation elements. The application of tracing to debugging is fairly common in usage and a powerful technique for understanding the relationships across artifacts in the development lifecycle. Tracing provides the developer with an accurate record of the

history of execution, which can be used during live or forensic [53] debugging to better understand how an error state has been reached and what portions of the system were involved.

2.2.3 Stepwise Execution and Breakpoints

The stepwise execution of a program is the most common form of tool support provided for debugging and available in the majority of integrated development environments (IDEs) for most languages (*e.g.,* Eclipse provides an open debug tool extension interface to which new languages may extend). Stepwise execution allows a developer to examine the state of a program during a dynamic execution while offering access to state information that would otherwise be hidden from the developer. The goal of this debugging approach is to allow the developer to examine a segment of a program more closely during execution to identify erroneous code. Stepwise Execution debuggers typically provide two key features: stepwise execution and breakpoints [15, 16]. Tools providing Stepwise Execution also support viewing dynamic state information while the program is paused at a certain point, and some tools allow the alteration of state information during runtime [16]. Thus, developers may execute the system in a stepwise manner and pause at predictable points in execution to view dynamic state data to better understand the behavior of the system under study.

2.2.3.1 Stepwise Execution

Most development tools support three basic stepwise execution features: `stepOver`, `stepInto`, and `stepOut` [16]. `StepOver` executes a single unit before pausing execution again. The unit stepped over may be a composite unit such as a function, but will be executed completely before pausing execution. `StepOver` enables developers to maintain focus within a certain scope during stepwise execution. `StepInto` moves to the first executable unit within a composite unit of code. `StepOut` moves execution to the first unit of executable code in the containing scope, and executes

any remaining code in the current scope. Two additional commands are `play` (which causes the program to execute until either a breakpoint is reached, or the pause command is activated) and `pause` (which halts execution at the current point). The combination of these commands enables developers to traverse the dynamic execution of their program and, with careful navigation, view any state of interest. It is possible for a developer to mistakenly navigate beyond a point of interest, such that the only method of returning to past points in the execution history using traditional stepwise execution requires restarting the program execution. However, several approaches exist that can address this issue. Omniscient debugging (discussed in Section 2.2.4) allows the developer to return to past points in execution history during a debugging session. Alternatively, techniques such as program slicing and QBD (discussed in Section 2.2.5) can be used to identify points of interest before execution and prevent unnecessary repeat executions.

2.2.3.2 Breakpoints

A key feature of most implementations of stepwise execution is the concept of a breakpoint [15, 16], which represents a point in the code where the developer wants to pause the execution of a program. Breakpoints enable developers to run a program and only pause at areas of interest within the code. Without breakpoints a developer would either need to carefully time the invocation of the pause command during stepwise execution or manually step through many uninteresting areas to reach an area of interest. Breakpoints may be further enhanced in the form of conditional breakpoints that only pause execution if a certain condition is found true when the breakpoint is reached [15, 56]. Conditional breakpoints are particularly useful regarding looping structures or commonly executed code where a traditional breakpoint would cause the debugger to manually examine many uninteresting states before reaching a state of interest [16].

2.2.4 Omniscient Debugging

The IEEE *Systems and Software Engineering – Vocabulary* provides the following three definitions:

- Error: "The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." [57]
- Fault: "An incorrect step, process, or data definition." [57]
- Failure: "Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits." [57]

Omniscient debugging enables a developer to trace backward in time through a program's execution history from the location an error was identified and trace to the location of the fault that caused the failure. This terminology is important to clarify that the underlying cause of an error is not always located at the point in execution where the error is identified. Lienhard, Gîrba, and Nierstrasz [8] state, "the hardest task is to find the actual root cause of the failure as this can be far from where the bug manifests itself." Omniscient debugging implementations combine the advantages of tracing techniques with those of stepwise execution and breakpoint based debugging techniques [9, 10].

Omniscient Debugging is not a new technique in the realm of GPLs. Zelkowitz published on the concept of reversible execution in the early 1970s [54]. Since this time, significant work has been undertaken in the context of GPLs including several commercial products [58]. However, these techniques have been focused on either utilizing low-level machine implementations to support reverse and replay, or utilizing traces designed to capture information for a given GPL.

The work described in this dissertation has focused on a trace-based omniscient debugging approach initially supporting hybrid model transformation languages, as discussed in Chapter 4, and later expanding to xDSMLs in a collaborative project with researchers from IRISA/INRIA and the University of Rennes, discussed in Chapter 5. Both debuggers are defined at the level of the transformation engine. Thus, omniscient debugging is supported at the level of CRUD (Create, Read, Update, and Delete) operations in a modeling environment.

The primary challenges of omniscient debugging are scalability and usability. Scalability refers to the challenge of maintaining enough state information to enable debugging backwards in time to any previous point in the execution history of a system. This issue has been explored by many researchers and several potential solutions have been presented. These solutions are outlined in the following:

- Utilizing garbage collection similar to the facilities available in modern GPLs such as Java [6–8]. The garbage collector removes references to elements that are no longer referenced and not relevant to the execution history. This approach attempts to minimize data collected over time, but in some scenarios these elements may need to be regenerated, thus reducing runtime efficiency.

- Limit the portion of execution history that can be navigated [6], which would provide a window effect (statically located at an area of interest or dynamically located around the current point of execution). The effect, advantages, and disadvantages of this solution are similar to utilizing garbage collection, but whereas garbage collection would maintain some history of elements and try to identify only irrelevant elements for destruction, the window solution removes all information outside of the current window.

- Identify a subset of the program as being of interest to the debugging process [6] and only record information concerning these elements. This solution can be applied in a static manner (*e.g.,* select elements of interest before playback begins) or in a dynamic manner (*e.g.,* select elements no longer of interest during runtime) [10]. This approach creates the challenge of discerning which elements will be of interest. This is particularly a concern for the static approach, which requires foreknowledge of all interesting elements. Several analysis techniques could be used to inform this decision (*e.g.,* impact analysis or program slicing), but the potential exists for false positives that would decrease the efficiency, and false negatives that can cause important information to be ignored.

In addition to the scalability challenge, omniscient debugging introduces the usability challenge of navigating to meaningful locations in the execution history. Some existing tools provide a means to utilize query-based debugging to identify elements of interest and locations in the execution history of interest. The query-based debugging approach provides useful information, but represents a cost to runtime efficiency because of the need to evaluate queries, in addition to providing omniscient debugging facilities. TOD (trace-oriented debugger) [9, 10] incorporates this approach, but implements a minimal query language compared to focused query-based debuggers in order to preserve runtime efficiency.

2.2.5 Query-based Debugging

QBD reduces the burden of bug localization on developers utilizing debugging approaches such as stepwise execution and breakpoints or, as discussed in Section 2.2.4, omniscient debugging [14]. QBD implementations provide query languages that enable developers to ask questions about a program. The developer-provided queries analyze the program for portions of the code

that are relevant to the query [14]. The query-based approach provides a semi-automated solution for bug localization. Both static [15, 59–61] and dynamic [5, 11–15, 61] approaches to QBD have been investigated. Static approaches focus on utilization of trace information and analysis of static documents including any input, output, and executable elements. The static approaches do not necessitate the inclusion of any trace information or additional features added to runtime environments or code in order to utilize the technique [59]. However, trace information, which may require some of these alterations, is helpful during the debugging process. The dynamic approach requires additional support for query evaluation.

One noteworthy query-based debugger is Whyline [5, 11, 12], originally introduced for debugging Alice code [5, 11] and later evolved to debug Java code [12]. The Whyline focuses on providing a guided debugging session by offering suggested queries to the user. The queries presented by the Whyline focus on questions such as "why does property x of object y have this value?" and "why is property x of object y not set to this value?" The suggested queries focus on exploring the reasons behind a failure [3]. This differs from typical query-based debuggers that offer an expressive query language and require the developer to provide custom queries.

2.3 The State of Debugging in MDE

This section briefly reviews the five approaches to debugging (as described in Section 2.2) within the context of MDE.

[3] Failure was previously defined to be "termination of the ability of a product to perform a required function or its inability to perform within previously specified limits" [57]

2.3.1 Language and Basic Tool Support

2.3.1.1 Print Statement

In many model transformation languages the equivalent of a print statement is either not available or is complicated to implement, thus limiting the usefulness of this debugging approach [16]. Mannadier and Vangheluwe described several potential methods for implementing a print statement in model transformations [16]. The simplest technique uses action code in a transformation to add printing features such as calling the print function provided by Python action code in AToM3 [4] [16]. Alternatively, entire rules focused on providing print statements could be created, but these rules would need to be constructed carefully to avoid altering the models and to ensure proper termination [16]. The final option presented by Mannadier and Vangheluwe is to provide language support by adding special print rules to MTLs. Mannadier and Vangheluwe argue in favor of adding print rules and comment that every GPL incorporates print rules despite the function of print rules being largely limited to debugging in modern practice [16]. It is worth mentioning that GPLs are also used in the creation of command line programs that require some form of print command for output purposes, but modern development is primarily focused on the creation of graphical interfaces that typically do not require the use of print statements. A similar example in favor of adding print rules to provide debugging support is the inclusion of assertions in many modern GPLs.

2.3.1.2 Assertions

Mannadier and Vangheluwe briefly discuss the possible application of assertions to MTs [16]. Two methods for implementing assertions for model transformations are proposed: 1) encoding assertions as rules or patterns for the MTL, and 2) adding support for assertions as a feature of the

[4] `atom3.cs.mcgill.ca`

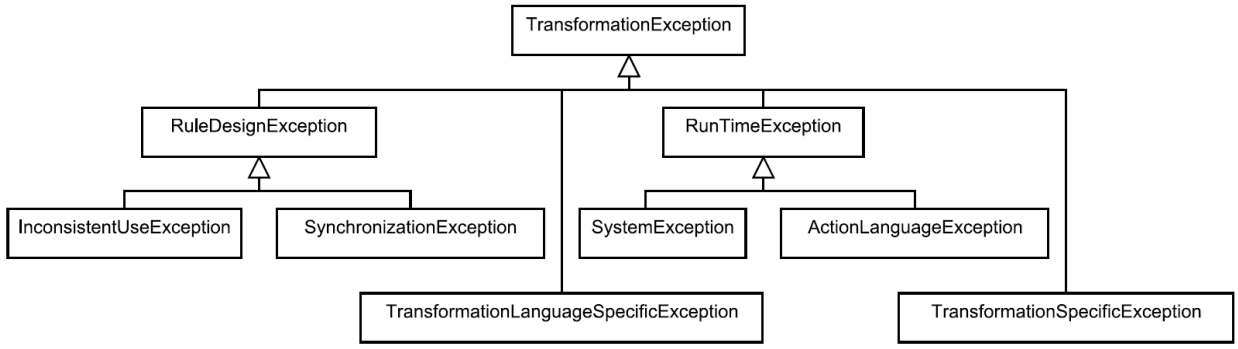


Figure 2.3: An Exception Hierarchy for Model Transformation [1]

MTL [16]. The first option (encoding assertion) has the same issues as were mentioned for print statements. The developer must ensure the assertion is side-effect free and properly terminates. The second option (adding language support for assertions) is more promising, but requires support from the developers of the MTL to implement the necessary features.

2.3.1.3 Exceptions

Applying the concept of exceptions to model transformations has been explored by Syriani, Kienzle, and Vangheluwe [1]. They introduce a set of proposed exception types for model transformations and an experimental version of the exceptions is implemented in MoTif [1]. The proposed exception types are illustrated in Figure 2.3 and briefly described as follows:

- Rule Design Exception occurs when there is an error in the transformation rule declaration.
  - Inconsistent Use Exception occurs when a rule or set of rules may be applied in several legal orders, but produce different results depending on order of execution.
  - Synchronization Exception occurs when a set of parallel transformations affect each other and therefore cannot be applied in parallel.
- Runtime Exception occurs when an exception is encountered at runtime in the transformation engine.

  - System Exception occurs when the virtual machine executing the transformation encounters a runtime error, such as running out of memory.
  - Action Language Exception occurs when an exception occurs in the action language.

- Transformation Language-Specific Exception is a class of exceptions specific to a particular transformation language, such as an unbound parameter exception in languages that allow parameterized rules.

- Transformation-Specific Exception is a class of exceptions that can be defined by the developer and encompasses exceptions specific to a particular transformation set.

A model for handling exceptions in model transformations is also proposed that includes adding an additional output for exceptions originating from a transformation [1]. The exception output may be connected to a handler that resolves the particular exception. The handler may resume execution normally for the transformation, restart execution to return to the beginning point of execution for the transformation, or terminate execution to cancel execution of the transformation [1]. The three options may all be undertaken without any portion of the system outside of the current scope being aware that an exception was encountered. Alternatively, the handler can propagate the exception to any enclosing scope. If no further enclosing scope exists, the execution is halted and the state information encapsulated in the exception is displayed. If an exception is not caught, then the exception is propagated as described previously [1].

2.3.2 Tracing

Amstel, van den Brand, and Serebrenik [62] state, "traceability plays an essential role in a number of typical model development scenarios such as debugging." Mannadier and Vangheluwe point out that the application of traditional stack traces is very dependent on specific details of the

Name	Stepwise Execution	Breakpoints
TROPIC	X	X
UML Model Debugger	X	X
ATL	X	X
Tefkat	X	X
GReAT	X	
AToM3	X	
VIATRA2	X	
Graphical Model Debugger Framework	X	

Table 2.1: Common Debugging Features Implemented in Model Transformation Tools

MTL [16]. An MTL such as ATL that supports invoking a rule from within another rule would benefit from stack traces similarly to GPLs for debugging. However, if all rules are atomic, then stack traces would only inform the developer of the current rule, which is not a significant aid in the debugging process. However, the traceability links are significant for any MTL because it can provide a detailed execution history including the order of transformation execution [63] (especially useful in an environment with nondeterministic scheduling) and links between source and target models [64] that can be used to reason about the runtime behavior of a transformation [65].

2.3.3 Stepwise Execution and Breakpoints

Several model transformation tools support the application of stepwise execution along with observing and altering the state during debugging. A few of those tools also support breakpoints for debugging. Table 2.1 summarizes the model transformation tools that have been identified within the literature as offering these debugging features.

2.3.3.1 Stepwise Execution

All of the identified tools that support debugging concerns implemented some form of stepwise execution. The typical step-over, step-into, and step-out functions are supported by all of the environments identified. All tools except the Graphical Model Debugger Framework (GMDBF) [66] implement the play and pause commands of stepwise execution discussed in Sec-

tion 2.2.3.1. Some of the tools, such as AToM3 [5], additionally include the ability to control the nondeterministic scheduling of rule execution (*i.e.,* manually choose the rule to be executed from a set of applicable rules) or to allow the scheduling engine to determine the next rule application. Generally, all tools possess the same level of implementation and only differed in features derived from differences in the environments, such as allowing rules to contain other rules, imperative vs. declarative vs. hybrid style rules, and deterministic vs. nondeterministic scheduling. One exception was GMDBF [66], which was focused on enabling the developer to view live runs of an embedded system at the model level. GMDBF may result in updates that occur too quickly for a human to monitor and does not allow controlled execution during live debugging sessions. However, GMDBF includes a replay engine that allows a user to replay system execution using trace information gathered during a live run. The live run does not allow the developer to control the underlying system execution, but the offline simulation allows such control [66]. Most tools focus on providing debugging support within the constraints of the chosen implementation languages (source model(s), target model(s), and model transformation), but TROPIC [17] converts the entire system to a custom colored Petri net. The primary benefits of this approach are that the Petri net gives the transformation process an explicit definition and the Petri net provides access to potentially useful automatic verification. For example, a Petri net can be automatically checked to make sure it is possible for all transitions to fire. If a transition is found to be unable to fire at any point during execution, this indicates a portion of the model transformation will never be invoked (similar to unreachable code in a GPL) [17]. Several more examples are provided of automatic verification supported by the Petri net formalism being applied to indicate potential faults in the model transformations.

[5] `atom3.cs.mcgill.ca`

TROPIC allows stepwise execution where each step furthers the execution of the Petri net by invoking a transition and allowing the developer to observe the movement of tokens (which represent model elements and properties) through the system [17]. However, the conversion to a Petri net removes the developer from the context provided by the source model(s), target model(s), and model transformation and places the developer in the context of debugging a Petri net with places, tokens, and transitions derived from the original system. The Petri net labels for places are taken from the models to retain domain terms, but any carefully constructed formalism is traded for the generic, but powerful, Petri net formalism. The developer is able to debug in a Petri net to view the process that provides clear control flow and animations for the debugging process, but the developer must debug in a language other than the one chosen for implementing the system being debugged. This creates a language mismatch between how the developer described and developed the system, and how the debugger is executing and displaying the system. All other surveyed tools focused on providing facilities focused on debugging within the context of the source modeling language, but no other tool provides the facilities for the automatic verification properties available in TROPIC due to the use of the Petri net formalism.

2.3.3.2 Breakpoints

Several existing tools support the use of breakpoints during debugging tasks, including TROPIC [17], UML Model Debugger[6], ATL[7], and Tefkat[8]. The breakpoint implementations function similar to traditional breakpoints, as seen in GPLs, with a given point in the model transformation process tagged as a location of interest and execution halted when the breakpoint is reached. From a survey of supporting modeling tools, only TROPIC enables the use of conditional break-

[6] `http://www.research.ibm.com/haifa/projects/services/uml/index.shtml`
[7] `www.eclipse.org/atl/`
[8] `tefkat.sourceforge.net`

points that allow the breakpoint to halt execution if a specified condition is evaluated true. TROPIC implements conditional breakpoints using an OCL query language that can specify constraints and conditions to be checked when the breakpoint is reached.

2.3.4 Omniscient Debugging

Recently, there has been some work in the area of MDE toward the application of omniscient debugging. My prior work, discussed further in Chapter 4, has investigated applying omniscient debugging to model transformations toward a scalable and performant omniscient technique [32, 33]. Furthermore, in collaboration with researchers from IRISA/INRIA and the University of Rennes, I have explored applying omniscient debugging to an xDSML environment [35]. The collaborative work, discussed further in Chapter 5, describes a technique that utilizes generated domain-specific trace metamodels along with a generated domain-specific trace manager to support a generic omniscient debugger implementation for xDSMLs. This collaborative work also investigated multi-dimensional omniscient debugging traversal features. These features enable a user to explore history through only the steps relevant to a given model element. Thus, the developer can minimize the time spent reviewing steps not of interest. Van Mierlo presented a proposal toward the debugging of executable models defining simulation semantics [67]. A particular focus of the work addressed handling simulated real-time. The scope of my contributions presented in Chapter 4 and Chapter 5 concerns applying omniscient debugging to MTs and an environment supporting xDSMLs, respectively. My work does not concern handling simulated real-time or relating the model entities with generated code.

2.3.5 Query-based Debugging

Although some support for model querying has been explored in the existing literature [68–71], TROPIC [17–19] is the only tool that supports query-based debugging in the context of model

transformations. TROPIC includes a debugging console (based on the Interactive OCL Console provided by Eclipse) that can be used to invoke OCL functions to generate queries that can be evaluated in the runtime environment [17]. TROPIC provides an adequate query engine, but suffers from the same language mismatch described previously. The use of OCL for debugging has also been discussed briefly in the context of debugging for C++ by Hobatr and Malloy [72], who presented a UML metamodel and OCL compiler to compile the OCL queries into C++ code. Their approach utilized modeling techniques, but is focused on the application of debugging a GPL.

Important to the discussion of QBD is knowledge about the queries that developers use during the debugging process. Hibberd, Lawley, and Raymond [53] introduced three categories of questions that developers may ask during the debugging process for model transformations. The first category of questions is concerned with logical bugs, "violation of a relationship constraint between the source and target model" [53] (*e.g.,* "Why didn't source type, `t`, result in any target objects being created?" [53]). The second category of questions is concerned with well-formedness bugs, "violation of the constraints specified by the target metamodel(s)" [53] (*e.g.,* "Why was the single valued reference, `r`, assigned more than once?" [53]). The final category of questions is concerned with analysis questions and is composed of two sub-categories. Sub-category I concerns bug smells or static analysis questions that focus on identifying structures commonly associated with bugs (*e.g.,* "For all source types, which source objects of the selected type did not contribute to the creation of any target objects?"). Sub-category II concerns information discovery questions which, intuitively, gather information from the available sets of data such as trace information, source model(s), target model(s), and model transformation(s) (*e.g.,* "Given a target object, what source objects contributed to its creation?" [53]).

CHAPTER 3

SUPPORTING COLLABORATION IN A CLOUD-BASED MDE ENVIRONMENT

In MDE, stakeholders work on models in order to design, transform, simulate, and analyze systems. Complex systems typically involve many stakeholder groups working in a coordinated manner on different aspects of a system. Therefore, there is a need for collaborative platforms to allow modelers to work together. This chapter presents a collaborative project developing a multi-user, multi-view architecture for AToMPM and an initial evaluation of its performance and scalability.

3.1 Introduction

Complex systems engineering typically involves many stakeholder groups working in a coordinated manner on different aspects of a system. Each aspect addresses a specific set of system concerns and is associated with a domain space consisting of problem or solution concepts described using specialized terminology. Therefore, engineers express their models in different DSMLs to work with abstractions represented in domain-specific terms [30]. Recently, there has been a growing trend toward collaborative environments, especially those utilizing browser-based interfaces. Common tools include Google Docs[1], Trello[2], Asana[3], and more. Additionally, this trend can be seen in software development tools including WebGME [73], a web-based collab-

[1] `https://docs.google.com`
[2] `http://www.trello.com`
[3] `https://www.asana.com/`

orative modeling version of GME, and Eclipse webIDE[4]. These tools bring together developers, including geographically distributed teams, in a collaborative development environment to work on a shared set of software artifacts. However, the introduction of these collaborative environments brings new concerns.

My prior work has defined the requirements and challenges for a collaborative modeling environment [25]. In that work, Eugene Syriani and I refined a set of basic collaboration scenarios for multi-view modeling, discussed further in Section 3.2. Although there is a growing need for such collaborative environments (as discussed in Section 1.2), few modeling tools allow multiple stakeholders to work on their modeled system concurrently. The degree of collaboration between developers is often restricted to the version control repository used by the tool (*e.g.,* subversion, github). The lack of focus on the collaboration and integration of models and languages among teams and users has been a major impediment for industries who operate globally around the world [74]. To ensure consistency and synchronization among the artifacts produced by each stakeholder in a live environment, a cloud-based environment is preferrable. My prior work also presented the architecture of AToMPM [24, 25, 75], a cloud architecture designed for collaborative modeling, along with both a detailed description of how AToMPM addresses the challenges of the presented collaboration scenarios [25] and an initial evaluation of the architecture [75]. The work described in this chapter builds upon the prior work of my collaborators Eugene Syriani, Simon van Mierlo, and Hüseyin Ergin to define an in-browser multi-paradigm modeling environment [23] which is overviewed in Section 3.4.1.1; and the prior work of Simon van Mierlo to define a basic model storage and processing system (Modelverse Kernel or MvK) [76] which is overviewed in Section 3.4.2.

[4] `https://eclipse.org/ide/`

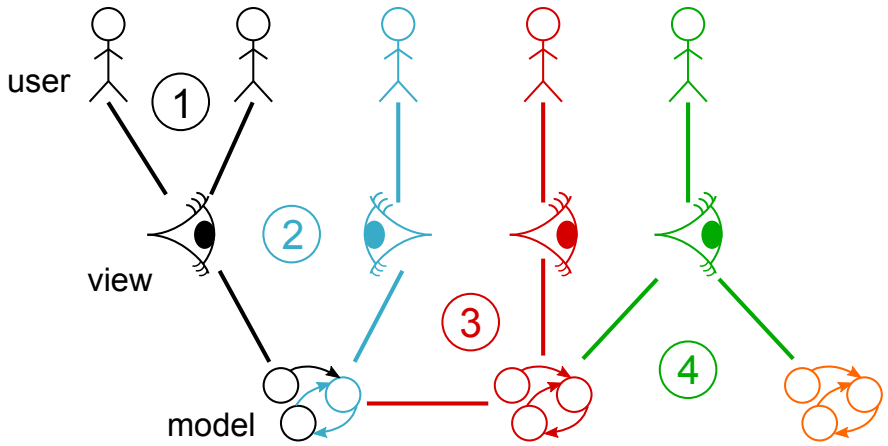


Figure 3.1: Scenarios in multi-view modeling

The chapter is structured as follows. Section 3.2 overviews the collaboration scenarios for modelers in a multi-view modeling environment. Then, Section 3.3 discusses further goals and concerns motivating the design of the architecture. Next, Section 3.4 presents the cloud architecture of AToMPM, designed to support collaborative modeling, and Section 3.5 empirically evaluates the performance and scalability of the architecture. Finally, Section 3.6 summarizes and discusses related work, and concluding remarks are provided in Section 3.7.

3.2 Collaboration Scenarios in Multi-View Modeling

In practice, teams of stakeholders work together in order to produce a coherent and complete system. Modeling tools, frameworks, and language workbenches typically consider all developed artifacts as models. In the context of a collaborative effort among individuals, such tools must separate views from models. A view is a projection of the model, showing only a part of the model in its own concrete syntax representation. Models are stored in a cloud-based repository and can only be accessed via their views. However, it is not necessary for a model to have a view to exist. For tools to support collaborative modeling activities, four possible collaboration scenarios are explored here for multi-view modeling. These are illustrated in Figure 3.1. For simplicity, the

discussion is restricted to two users/views/models, although generalizable to an arbitrary number of each component. For each scenario, a briefly discussion is provided of what conflicts can occur when users are connected live to the cloud and an idea is offered on how to resolve them. Note that, in practice, any combination of these scenarios is possible, but each one will be treated separately in the following discussion.

3.2.1 Multi-User Single-View (1 in Figure 3.1)

Stakeholders are working on the exact same artifact. This is equivalent to having both of them share the same screen to manually inspect a model together during training. In this case, two users are working on the same view of the model. They both see the exact same data in the same concrete syntax. Changes made by one user are reflected automatically to the other. In the case of simultaneous conflicting changes (*e.g.,* one user is deleting an element that the other user is updating), the conflict is resolved in a first come, first served fashion.

3.2.2 Multi-View Single-Model (2 in Figure 3.1)

Stakeholders are working on different parts of the same artifact. This situation is useful when artifacts can be designed incrementally and in parallel. In this case, two users are each working on a different view of the model. The two views may be presenting the same elements in the same or different concrete syntax. They may also share only some or no elements between them. In any case, the views represent a part of the same model and therefore the models in the views conform to the same modeling language. In the non-intersecting views cases, changes in one view are not reflected in the other view. Otherwise, if an element is present in both views, changes in its abstract syntax in one view are reflected in the other view. However, changes that only affect the concrete syntax of the element are not reflected in the other view. Note that some changes in the abstract syntax of an element may change its concrete representation. In the case of

simultaneous conflicting changes on a shared element, a conflict management strategy must be put in place to solve the conflict and maintain the two views consistent with each other.

3.2.3 Multi-View Multi-Model (3 in Figure 3.1)

Stakeholders with differing expertise are working on distinct artifacts that, together, compose the overall system. In this case, each artifact represents a concern of the overall system (*e.g.,* the electrical and software concerns of an automotive). Two users are each working on a different view and each view is a projection of a different model. The models define together the overall system. This means that elements in each view are part of models conforming to different modeling languages. However, the two models are subject to global consistency constraints (*e.g.,* a method call to an object in a UML sequence diagram can only be established if the UML class diagram has defined that method on the target class of the object). Furthermore, an element in one model may explicitly rely on elements in the other model, by having a reference to it. Only if that reference is part of the view of the second model, then changes to the referenced element in the first model may be reflected in that view. In particular, if the referenced element is deleted in the original model, then a decision must be made for whether the deletion should cascade to the reference or not. Simultaneous conflicting changes to that element are not existent between the two views, since one view passively references the element.

3.2.4 Single-View Multi-Model (4 in Figure 3.1)

This is a particular case of the previous scenario where one user is working on a view that projects two models, while another user is working on a view of one of these models. In this case, it is the view that makes the link between elements of the two models as opposed to the previous case where that link is defined in the model. Typically, the view represents an abstraction of elements of the two models (*e.g.,* a correspondence relation between the resistors of the electrical circuit and

the heat sensor of the engine). This view is generally the one that defines the consistency relation between elements of the models. A change that occurs on the abstract syntax of an element in one of the models may therefore affect the view. This scenario is outside the scope of this chapter.

3.3 Design Goals and Concerns

The primary guiding concern driving this contribution to the dissertation is to create a system that can support collaboration and modeling where it is most natural to the users. Thus far, the focus has been on the motivating scenarios for collaboration (Section 3.2). However, a number of additional factors also motivated the design. This section describes some of the goals and concerns that have guided the design of the AToMPM architecture.

3.3.1 Responsiveness

A primary concern for the architecture is to provide an acceptable level of responsiveness. Because of the focus on small scope CRUD operations, it is imperative to ensure these operations process in a minimal amount of time. A user across a session will make many individual CRUD operations (*e.g.,* updating the number of tokens in a Petri net place), and these operations should not present a notable delay in a user's process. Additionally, these CRUD operations form a basis for model transformations and larger processes.

3.3.2 Distinct View and Mode of Interaction

Modelers working within a shared artifact to collaborate on a larger project may not always share the same focus, working with subsets of the same model. Alternatively, they may desire to visualize a model for varying purposes. AToMPM utilizes views to provide each modeler with the capability to refine the scope and style of visualization when working on shared artifacts. A specific view may contain only a portion of a model or utilize a distinct visualization.

3.3.2.1 Models and Views

Within AToMPM, models are separated into two categories: abstract models and views. The distinction between views and abstract models allow modelers to work with portions of a model and use distinct representation, arrangement, and sizing for the elements of their view. Thus, stakeholders are able to share models, allowing modelers with varying skills to work collaboratively on the same model. An abstract model is the abstract syntax of a model conforming to the metamodel of a given DSL. Conceptually, a view is a projection of an abstract model onto a DSL that uses the most appropriate representation of a subset of the model's elements for the needs of the expert modeler working on that part of the model. A view references a set of elements from the abstract model. The view also maintains a list of other models necessary to use the specific view. The most common example of a model found in this list is a concrete syntax model. By varying the concrete syntax between views, different users may view the same model using varying visualizations (*e.g.,* textual or graphical). Other inclusions in the list could be executable models used for simulating the model through model transformation. Finally, the view includes a mapping of model elements to concrete syntax information, because these mappings are specific to a representation of the model. These mappings might include the absolute position, rotation angle, and size scaling of the element.

3.3.3 Distinct Mode of Interaction

Additionally, the architecture of AToMPM separates the processing for model and view actions from the specifics of the client. Thus, it is possible for developers to work within varying client systems. A modeler may choose to utilize a tablet or other touch-enabled device to manipulate a primarily graphical view, but the modeler could also utilize a purely textual command line environment to manipulate the model.

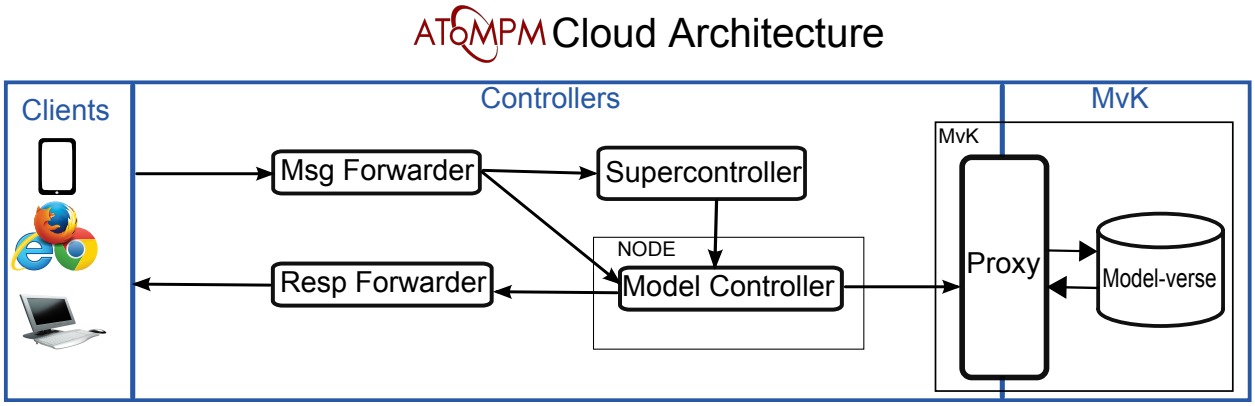


Figure 3.2: Overview of AToMPM architecture

3.3.4 Managing Conflicting Requests

As the number of concurrent users accessing a shared resource increases, conflicting requests become innevitable. Managing conflicting requests is a primary concern for systems enabling shared access to resources. However, due to the nature of request management within the Controllers portion of the architecture, conflicting requests are not truly processed concurrently. The system will always be able to process requests with conflicting requests being automatically resolved based on order of arrival. However, the client systems may still detect failures or even lost updates (*e.g.,* a scenario where a second update overwrites a prior update before the second user is made aware of the first update) due to the stream of updates from the Model Controller. Thus, the client system may provide further, more advanced and costly, methods of conflict management.

The core architecture focuses upon resolving all messages with minimal processing to preserve responsiveness. This focus on responsiveness also works to reduce the possibility of conflicting requests. Decreasing time spent processing and providing updates to the user more quickly, decreases the window in which users may provide conflicting requests. If the time to process a request and update all clients is 1 second, then a conflicting request must occur within 1 second of

the previous request being made. If the time is reduced to 0.1 seconds, then the conflict must occur within 0.1 seconds of the prior request.

3.4 Multi-View Modeling in AToMPM

This section presents the architecture of AToMPM. This architecture has been designed to resolve the challenges of providing a performant multi-user, multi-view modeling environment. Figure 3.2 illustrates the components of this architecture: client systems, Modelverse Kernel (MvK), and a set of controllers coordinating the other two components (referred to as the Controllers). The three components communicate by exchanging changelogs encoded in JSON that contain all the necessary information to perform a task. This design implements the classic Model-View-Controller pattern, but a distributed network of controllers is provided to manage the concerns of a multi-user, multi-view modeling system.

3.4.1 Client

AToMPM includes a web client that runs in-browser. However, the underlying architecture is designed to be client agnostic and supports a wide variety of clients. The distributed system that manages models and views exposes a simple API with connections and transmissions made using 0MQ [5], an open source socket messaging library with support for a large variety of languages and platforms. The initial target client for the architecture is the AToMPM Web Client that was demonstrated at MoDELS 2013 [23], but the architecture is available to any client that can maintain a live connection to the back-end system. Thus, it is possible (in the future) to have a developer working on a tablet application, another using the AToMPM Web Client, and yet another using a command line editor. Each of these disparate users would have live access to the same model and able to collaborate in live time. To accomplish this goal, multi-view modeling, wherein a model

[5] `http://zeromq.org/`

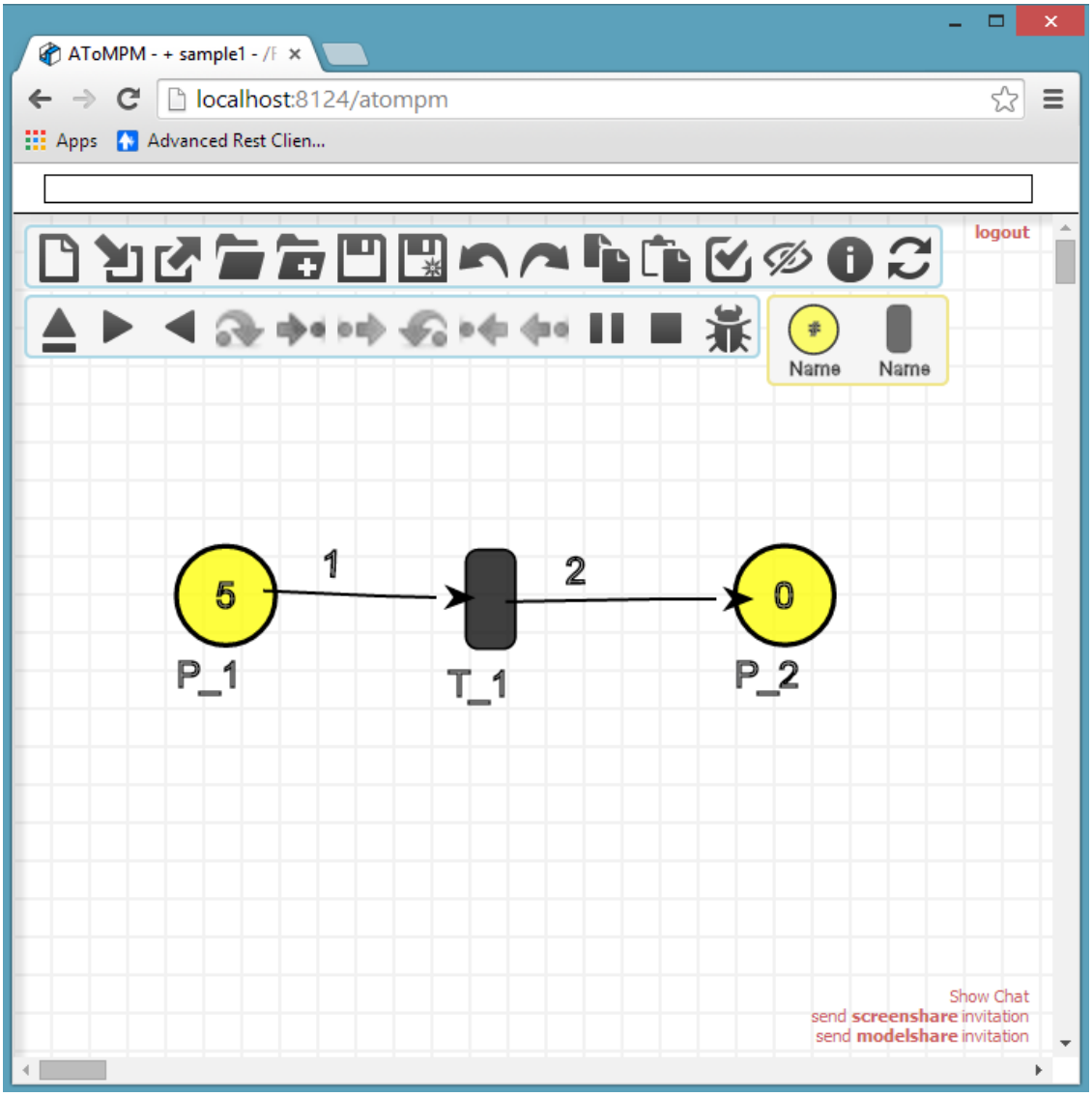


Figure 3.3: Overview of AToMPM architecture

is separated from its representation, is supported. The model may be represented through many views, which allows access to the same elements across such disparate environments that may not even support the same style of visualization and interaction.

3.4.1.1 AToMPM Web Client

AToMPM provides an in-browser GUI client for the user. Therefore, there is no installation required to perform modeling tasks. In the back-end, a node.js web server hosts the pages. The main features of the web-based GUI are:

- Creating DSMLs
  - Creating metamodels
  - Designing and assigning concrete syntax to metamodel elements
- Manipulating models through an HTML5 canvas
  - Creating, updating and deleting elements
  - Undoing, redoing changes
  - Copying and pasting elements
- Executing and debugging model transformations
- Collaborating with other users in real time

3.4.2 Modelverse Kernel

The Modelverse [76] is a centralized repository to store and manipulate models. All artifacts in the Modelverse are considered models, whether they describe a view, a model, a metamodel, or a model transformation. It provides an API, the MvK, to perform CRUD (create, read, update, delete) operations on model elements, check for linguistic conformance (with metamodels and constraints), and execute particular models of computations (built-in action language, transfor-

mations, and state machines). The MvK processes all modeling actions, such as primitive requests and execution, directly on models that are stored in the Modelverse.

3.4.3 Controllers

The Controllers portion of the architectures encompasses a series of distinct controllers that ensure consistency among clients, view(s), and the model(s). A user can perform CRUD operations, create a model, load a model, add or remove elements from a view, create new views, and execute a model or set of models. These requests are sent into the system through a message forwarder that routes messages to the proper controller for processing. Similarly, responses return to the clients via a response forwarder. Forwarders make use of the extended publish-subscribe pattern (provided by 0MQ). This pattern enables both clients and controllers to subscribe to messages regarding a specific view of an abstract model without needing to connect to the dynamically spawned controllers that manages the relevant views and abstract model. A client or controller can be subscribed to many models. The following describes how models are represented in the Controllers and presents three controller types.

3.4.3.1 Model Controller

The model controller provides a layer on top of the MvK that incorporates multi-user, multi-view modeling. A model controller manages a given model and all views of that model, and there exists one model controller per active abstract model. In this way, the processing of concurrent user requests is distributed based on the artifact being used. However, all requests for a given model are managed by a single controller ensuring consistency for the model and the associated views. Whenever a model operation alters a portion of the abstract model, the resulting changelog is published to all that have subscribed to a view that references the updated portion of the model. The view operations are guaranteed to have no side effect on the model and thus

do not need to update users of other views. Load and read requests are performed directly at the model controller, which are resolved locally using cached information. For all other operations, the model controller coordinates the processing of primitive operations within the MvK. The model controller ensures that the sequence of primitive operations is valid, and separates models from views, whereas they are treated uniformly within the MvK.

To enable automatic resolution of conflicting messages, the model controller guarantees one message must complete successfully (win) and other conflicting messages will fail. Incoming client changelogs are queued to be processed using a FIFO (first-in-first-out) strategy. Thus, operations that originate from clients sending a changelog may fail based on the results of a previous message. By allowing the earlier message to win, the system does not need to process any rollback operations due to conflicting messages. This scenario of conflicting messages is mitigated in practice by publishing update messages to ensure all users maintain a consistent copy of the model. Thus, even if multiple users send conflicting messages simultaneously, the users will be notified of the resolution for all messages. Furthermore, because users are notified of all updates immediately, the likelihood of generating conflicting messages is minimized.

3.4.3.2 Supercontroller and Node Controllers

Model controllers manage all model and view related operations. For each model, there exists exactly one model controller. However, the system may have any number of models. If the system maintained these model controllers at all times, a significant amount of resources would be wasted on model controllers that are not in use. To mitigate this issue, model controllers are spawned as needed on a distinct processor. Controlling which model controllers are active and assigning the model controllers to resources is the primary responsibility of the supercontroller. The supercontroller ensures there exists at most a single model controller for a given abstract model

and its related view(s), and it monitors all client changelogs entering the system. When a client changelog is encountered that requires a model controller not yet spawned, the supercontroller assigns the model controller to a node controller, collects all requests until the model controller is spawned, and forwards the requests when the model controller is ready to process requests. Additionally, the supercontroller ensures all client changelogs received during the spawning process are queued and forwarded after the model controller is sucessfully spawned. Any client changelog intended for an already active model controller is ignored by the supercontroller. When a model controller is spawned, it is assigned to a node controller. Node controllers are abstract representations of available hardware computing resources (*e.g.,* nodes in a cluster, CPUs, cores), and provide an interface to these resources for the supercontroller.

3.5 Evaluating the AToMPM Architecture in Collaboration Scenarios

This section presents an empirical evaluation of the architecture, focusing on the Controllers, targeting two key concerns: communication time for transmitting requests and processing time for handling requests transmitted. AToMPM is evaluated through a series of experiments where the size of models and number of concurrent users are varied independently. These experiments exercise the system in a wide variety of expected conditions including worst case scenarios to evaluate the performance of the system. Three experiments were conducted to answer the following research questions:

RQ 1 How does the architecture scale regarding the number of concurrent users?

RQ 2 How effective is the architecture in responding to a user request as the size of the model increases?

3.5.1 Experimental Setup

The supercontroller was running on a dual-core machine (3.33 GHz, 4GB RAM) and the MvK on a quad-core machine (3.1 GHz, 8GB RAM). Two dual-core machines (3.33 GHz, 4GB RAM) and a quad-core machine (2.4 GHz, 8Gb RAM) were simulating users to have true concurrency. The node controllers were running on three dual-core machines (3.33 GHz, 4GB RAM) and a quad-core machine (3.1 GHz, 8GB RAM).

This resulted in a load of 50 users and 50 model controllers per machine for each experiment. The experiment setup preloaded all users, model controllers, and models in the MvK before measuring. This warmup phase put the system in an average expected case; *i.e.,* how the system would perform after initial requests for each model controller. The startup time for a model controller is significant at approximately 2-3 seconds to spawn its process and some additional time spent building the cache of elements for the model and initial view. The time to load a view and the related model pair depends on the size of the models, but tends to be approximately 1 second per 2,000 elements in the abstract model, assuming the maximal case where the view references all elements (see startup time in Figure 3.4). The same Petri net models were used in all experiments with varying numbers of places and views. Note that all views referenced every element of the associated abstract model; *i.e.,* the experiments measured a worst case scenario for performance.

3.5.2 Experiments Evaluating Collaboration Scenarios

Each of the following experiments varied the number of concurrent users and model size independently. The variable for the number of concurrent users is $U = 1, 25, \ldots, 200$ with increments of 25. The variable for model size is $N = 1, 50, \ldots, 200$ with increments of 50. Here, N represents the number of places (*i.e.,* elements) in the Petri net model. Three experiments were conducted. Each experiment varied the resources shared by the users. However, all users send a

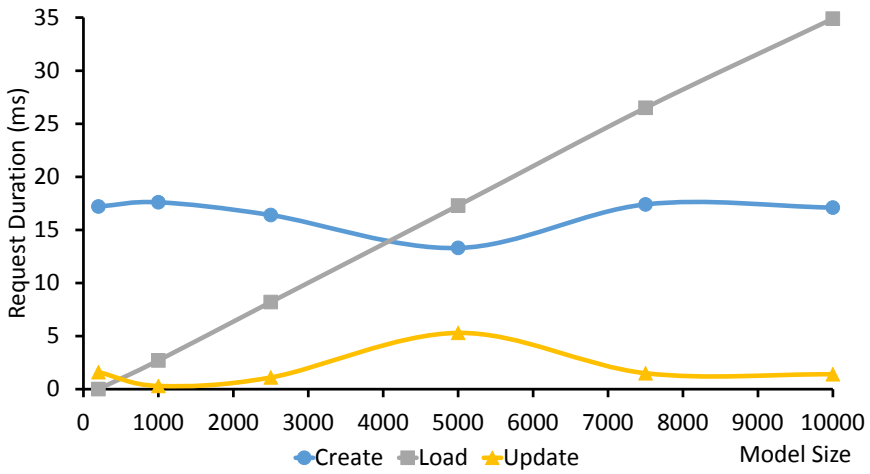


Figure 3.4: Create, Load, and Delete requests on a model controller with local MvK

series of load, create, and delete requests. Load is a read-all request sent to obtain an initial version of the model. Create is the most expensive single element operation. Delete is the least expensive single element operation. The load requests have a list of N dictionaries for a model of size N with each dictionary containing the relevant information for a specific model element. Each request is repeated 10 times by each user to minimize random machine and network effects on transmission time. Furthermore, all transmissions were performed on a LAN to minimize network latency and routing effects on transmission time. Below is a brief description of each experiment.

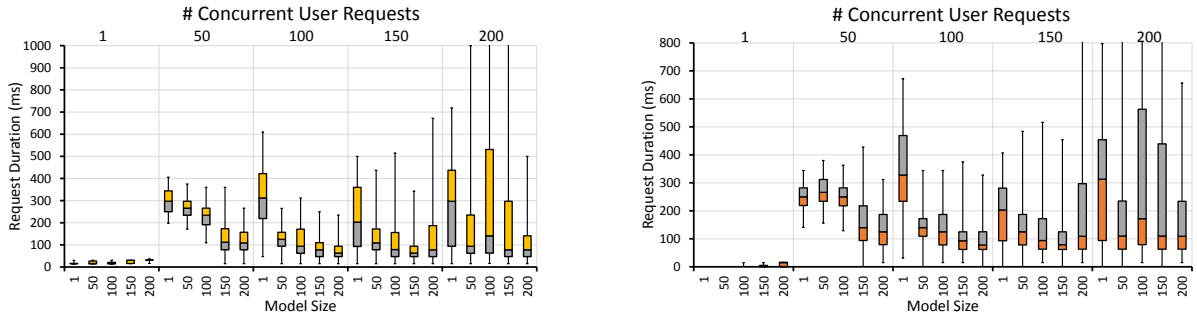
Experiment 1 Each user connects to the same view of the same model.

Experiment 2 Each user connects to a distinct view of the same model.

Experiment 3 Each user connects to a distinct view of a distinct model.

3.5.3 Results

Before discussing the three experiments, the Controllers were evaluated without network overhead in the communication with the MvK and with a single user (*i.e.,* a best case scenario).



(a) Create request

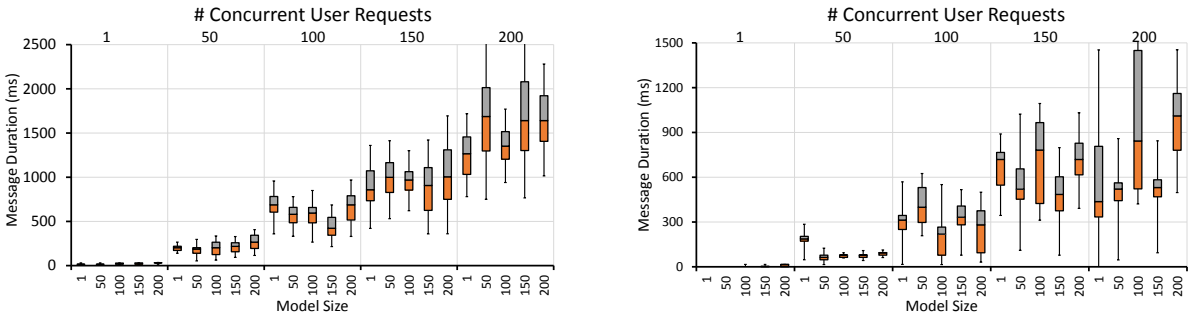
(b) Load request

Figure 3.5: Results for experiment 1

Figure 3.4 presents the results from a best-case scenario. Create requests are not affected by the size of the model, but are slower than delete requests. The time to perform a load request is linear with respect to the size of the model, because load requires accessing each element in the model. Also, unlike the other three operations, load bypasses the MvK using a cache in the model controller.

3.5.3.1 Experiment 1 - Same View, Same Model

This experiment demonstrates an approximately constant response time, except at 1 user, for all operations. Only the create and load operations are shown in Figure 4.10 because delete performs similarly to create. Smaller model sizes have a lot of variance, because the network communication is predominant. The most important result of experiment 1 is the large ranges for most runs. All runs contain approximately 1,500 data points that are typically uniformly distributed over a wide standard deviation. The average coefficient of variance (CoV) across all runs was 63.5% and the median CoV was 46.6%. The large variation is due to the fact that all messages are being sent at approximately the same time, similar to a distributed denial-of-service. This results in messages being queued at the model controller and MvK which must be processed sequentially. The variance grows more significant as the number of users increases (*i.e.,* the number of messages



(a) Create request

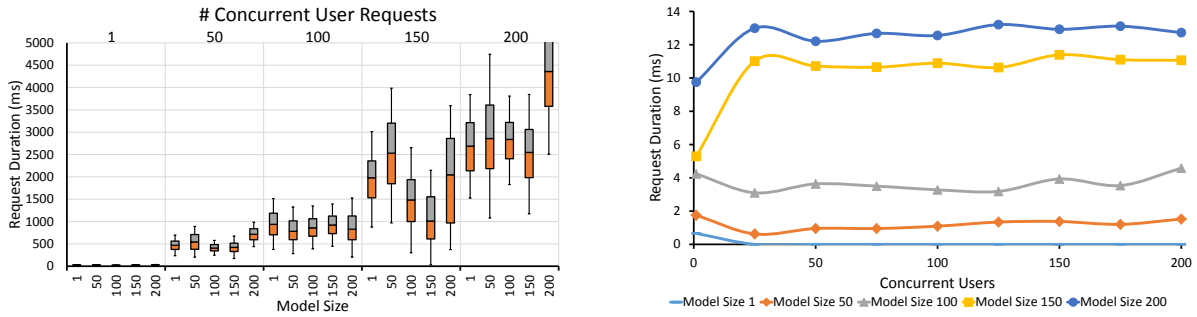
(b) Load request

Figure 3.6: Results for experiment 2

received concurrently increase). Experiments 1 and 2 share this quality due to their design: all users are on the same abstract model, thus both experiments will be processed by a single model controller. Nevertheless, all clients receive their response within less than a second.

3.5.3.2 Experiment 2 - Multiple Views, Same Model

This experiment demonstrates a linear increase as the number of users increases, but again model size has little to no impact, as depicted in Figure 3.6. This is because of the additional processing and fetching required to process many distinct view models, as opposed to experiment 1. Load operations, which ignore the MvK, scale similarly albeit slightly improved compared to the create requests. Thus, the bottleneck lies in how the model controller is managing multiple concurrent views. The performance of the operations are twice as slow as experiment 1. That is because priority is placed on the case where distinct users are typically using distinct abstract models and thus communicating with distinct model controllers. This design is more efficient with regards to the number of users, but not the number of views per model. Users are expected to collaborate on the same model and the architecture is designed for consistency in this scenario, but small numbers of users (on the order of 10 users per Model Controller) are expected in practice.



(a) Create request

(b) Load request

Figure 3.7: Results for experiment 3

3.5.3.3 Experiment 3 - Multiple Views, Multiple Models

This experiment is designed to test the scalability in the expected case where users are more evenly distributed across model controllers. In Figure 3.7, load is constant with respect to the number of concurrent users. However, the other operations perform slowly, taking at least a second. The main reason for that is the current implementation of the MvK. Although the Controllers are a distributed system, the MvK queues all requests and processes them sequentially. Performance of these operations will improve after the MvK is transitioned to a distributed design.

3.5.4 Discussion

The following discusses the implications of the results with regard to the two identified research questions.

3.5.4.1 RQ1: How does the architecture scale regarding the number of concurrent users?

Experiment 1 for all request types and experiment 3 for load demonstrate promising scaling with regards to increased number of concurrent users. However, experiment 2 identifies that many users accessing distinct views poses a concern for scalability with performance at $U = 200$ approaching 2 seconds for a create request. Nevertheless, the most significant concern is the limitation of the current MvK as demonstrated by comparing the create and load results for experiment

3. Create scales linearly reaching response times in excess of 5 seconds while load requests do not scale with the number of users in experiment 3. The results indicate a strong need for distributed processing in the MvK, and strong promise for the overall performance in experiment 3; *i.e.,* my expected average case. Overall, the system performs well with deficiencies when many users ($U > 50$) employ distinct views of the same system. Also, the system is unable to leverage the full potential of the model controllers due to the current performance bottle neck of the MvK. In conclusion, the Controllers portion of the architecture can efficiently handle multiple users sending concurrent requests, but large numbers of users on one or more views of a model will adversely affect scalability.

3.5.4.2 RQ2: How effective is the architecture in responding to a user request as the size of the model increases?

The results for the main experiments do not indicate scaling as model size varies. However, due to limitation of the MvK, it was not possible to test reasonably large models during the three main experiments. As a result, a smaller performance test was run using a single model controller with a localhost connection to the MvK. These results in Figure 3.4 report that create, update, and delete requests are not influenced by the size of the model. This is expected because these requests target a single element. However, load is affected by the model size, since it reads all the elements in both the abstract model and the view. In conclusion, the system scales as expected with regards to model size, but further work should investigate potential interactions of the variables N and U (*i.e.,* repeating the main experiments with larger scale models).

3.5.5 Threats to Validity

The first threat to validity regards the metamodel. The performance results have only been measured on similar Petri net models. Different formalisms may influence the results, in particular

those exposing complex inter-object relationships or heavy objects (*i.e.,* many attributes or large attributes). Another threat to validity is that the experiments did not measure models with more than thousands of elements, due to limitations at the MvK level where loading a distinct view model and Petri net model for hundreds of users exceeds memory bounds. Because of this deficiency, the experiments focused primarily on how the system scales with respect to the number of concurrent users. A third threat to validity regards the experimental setup. The way controllers are distributed on machines has an influence on the overall performance.

3.6 Related Work

Modern modeling tools are primarily native desktop applications, *e.g.,* EMF [77] and MetaEdit+ [78]. AToMPM is one of the first web-based collaboration tools for MDE. It takes advantage of the ever increasing capabilities of web technologies to provide a purely in-browser interface for multi-paradigm modeling activities. Nevertheless, Clooca [79] is a web-based modeling environment that lets the user create DSMLs and code generators. However, it does not offer any collaboration support.

Recently, Maroti *et al.* proposed WebGME, a web-based collaborative modeling version of GME [73]. It offers a collaboration where each user can share the same model and work on it. In contrast with the Modelverse, WebGME relies on a branching scheme (similar to the GIT version control system) to manage the actions of different users on the same model. Thus, WebGME supports multi-user single-view and multi-view single-model scenarios, but not in an always-online environment, unlike in AToMPM where operations immediately succeed or fail.

The GEMOC Initiative [30] has produced GEMOC Studio[6], a modeling system within the Eclipse ecosystem. Users can compose multiple DSMLs into a cohesive system with well-defined

[6] `http://gemoc.org/studio/`

interaction of the various models [80]. GEMOC Studio provides a solution for multi-view multi-model similarly to AToMPM, but does not handle concurrent users.

Gallardo *et al.* proposed a three-phase framework to create collaborative modeling tools based on Eclipse [81]. The difference between creating a regular DSML and creating a collaborative modeling tool in their system is the addition of the technological framework, which adds the collaboration support to the DSML. Naming it "TurnTakingTool," multiple users are able to modify an existing model by utilizing a turn-based system. The framework helps users create the DSML as a native Eclipse plug-in with concurrency controls, graphical syntax, and multi-user support.

Basciani *et al.* proposed MDEForge [82], a web-based modeling platform. MDEForge offers a set of services to the end-user by a REST API, including transformation, model, metamodel editing. The authors also mention adding the collaborative capabilities of the platform in the future.

The distributed databases community has also dealt with some of the issues of collaborative environments [83]. However, the primitives of modeling and distributed databases are not the same. Databases manage a restricted set of low-level data representations, and collaborative modeling systems utilize higher level abstractions that commonly include domain-specific concerns.

3.7 Conclusion

This chapter motivated the growing need for collaborative modeling environments to address the needs of domain experts working in a coordinated manner to model complex systems. The architecture of a cloud-based environment for collaborative modeling was presented with discussion of how it ensures consistency and synchronization among artifacts produced by each stakeholder. Finally, preliminary results were reported evaluating the scalability and performance of the arhitecture. The results reveal that the Controllers portion of the architecture is efficient with regards to the number of concurrent users, but distributing the MvK is necessary.

CHAPTER 4

OMNISCIENT DEBUGGING FOR MODEL TRANSFORMATIONS

This chapter discusses a technique for supporting omniscient debugging for model transformations, which are used to define core operations on software and systems models. Similar to software systems developed using GPLs, model transformations are also subject to human error and may possess defects. Existing MDE tools provide stepwise execution to aid developers in locating and removing defects. This chapter describes 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.

4.1 Introduction

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 [3]. 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 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 [84]. 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 [17]. Despite the focus on models and model transformations, traditional development concerns such as debugging must still be undertaken by developers adopting MDE practices.

Debugging is a fundamental SE task. However, despite the common need for debugging in software development, tool support for debugging has changed little over the past half century [4]. Several novel approaches to debugging have been introduced for general-purpose languages (GPLs), such as omniscient debugging [58]. 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 [48]). 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. As noted in Section 2.3.5, the only modeling tool identified in a survey of existing literature that includes an advanced dynamic debugging technique for model transformation is TROPIC [17], which provides support for QBD 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 (*i.e.,* error, failure, and fault) each receive a distinct definition in the IEEE 610.12-1990 standard glossary of software engineering terminology [85].

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 [86, 87] and QBD [17] 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 QBD 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, MTs are also subject to errors, and these errors may not manifest at the location of the defect. If a developer were to misidentify the location of a defect by targeting the location of an error, a traditional debugger would require restarting the system. Restarting can be expensive, requiring a nontrivial amount of time to re-execute or a significant delay due to manual input from the developer. Omniscient debugging avoids the need to re-execute to reach a prior state. MTs also have concerns not traditionally found in GPL systems that would benefit from omniscient debugging. Declarative MTLs commonly provide nondeterministic rule scheduling. The nondeterminism is commonly accepted because the rules should not be dependent on ordering to produce correct results. However, it is frequently possible to define transformations improperly such that the ordering of rule execution may vary the final result. In this scenario, it may be difficult to fully trace the source of a defect because the bug may manifest in one execution, but not in a subsequent execution. In these situations, an omniscient debugger enables the developer to fully explore the context in which the bug may be observed.

When considering support for omniscient debugging of MTs, the need for an efficient and scalable solution is evident. 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 [88]. Numerous works have been presented discussing topics such as parallel processing of MTs [89], techniques supporting incremental processing of model transformations [90], and cloud-based architectures for modeling and transformation [25, 82, 90]. Therefore, the technique presented in this chapter is designed to utilize a minimal structure to store required information in support of omniscient debugging, and has been developed with a set of algorithms designed to support efficient omniscient traversal of MTs. Kolovos et al. [91] define large-scale models as being on the order of millions of elements. As models continue to grow in size, the transformations and supporting transformation tooling operating on such large models must be designed for efficiency and scalability.

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

- Defining basic omniscient debugging features, and extending traditional stepwise execution features to support omniscient traversal (*i.e.,* both executing a transformation forward and reverting it back).
- Defining a minimal structure to store history for a MT engine.
- Discussing how the minimal history structure can be used to provide efficient omniscient traversal.
- Finally, providing an empirical evaluation of an implementation of the technique as com-

pared 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 chapter is structured as follows. Section 4.2 will overview related work in the area of omniscient debugging. Section 4.3 will present an illustrative scenario of a developer employing omniscient debugging. Section 4.4 will describe a novel technique enabling omniscient debugging for model transformations including theoretical analysis of runtime and space complexity bounds. Section 4.5, Section 4.6, and Section 4.7 will discuss the design and results of an empirical analysis of the performance and scaling of the technique. Finally, Section 4.8 will provide concluding remarks.

4.2 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. This section overviews relevant existing work on debugging in MDE and relates these works to the contributions presented in this chapter.

4.2.1 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 [67]. A particular focus of the work addressed handling simulated real-time. The scope of the novel contribution of this chapter concerns applying omniscient debugging to MTs. The work described in this chapter does not concern handling simulated real-time or relating the model entities with generated code. This work investigates applying omniscient debugging to model transformations toward a scalable and performant omniscient technique [32–

34]. Furthermore, in collaboration with researchers from IRISA/INRIA and University of Rennes, I explored applying omniscient debugging to an xDSML environment [35]. The collaborative work utilized domain-specific trace metamodels that were generated along with a 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. This chapter focuses on evaluating the changes to the underlying model transformation engine to support omniscient debugging. Because the multidimensional features are defined using a subset of omniscient debugging features (primarily jump), these additional features are not included in this chapter.

4.2.2 Tracing in MDE

My approach to providing omniscient debugging support for model transformations relies strongly on a trace of execution history. Several other researchers have investigated storing trace information for MDE systems. Falleri *et al.* presented work toward adding support for traces to Kermeta [65]. The presented trace framework for Kermeta implements basic traceability items including the concepts of link, step, and trace, trace serialization to XMI, and conversion of trace information to Graphviz's dot language for visualization. Jouault [21] also presented an implementation of traceability concerns for ATL. He mentioned that traceability has built-in support within ATL due to implementation concerns of declarative languages. However, the implementation of traceability generating code is not tightly coupled with program logic. In fact, the traceability presented by Jouault is woven into existing code in an aspect-oriented manner. In his implementation, the generation of traceability artifacts is a generic concern and can be applied via rules to existing

code without adding to the existing code base, which differs from the implementation in Kermeta discussed in [65]. Overall, the ATL implementations are very similar to the Kermeta-based implementation. The Kermeta-based implementation [65] provides a more detailed tracing metamodel and the ATL-based implementation [21] provides the ability to weave in traceability concerns in an aspect-oriented manner. The history trace used in my approach differs from these in several ways. The trace structure presented by Jouault [21] creates simple traceability links between elements, but does not contain any state information necessary to replay through a trasnformation. The trace structure presented by Falleri *et al.* [65] is much more similar to history structure defined in this chapter containing the basic concepts of steps with names that could be used to indiciate the transformation rules generating the step, but there is still no explicit storing of state data preventing it from being useful for an omniscient debugger. Furthermore, history is a much richer data structure. In addition to storing the necessary state data, history also caches information regarding when model elements were changed to assist in omniscient traversals (described further in Section 4.4.2).

Bousse *et al.* [92] introduce a trace structure similar to the history trace employed in my omniscient debugger. However, Bousse *et al.* are focused on generating domain-specific traces for executable domain-specific modelling languages (xDSML) within the GEMOC environment[1]. The trace structure they utilize stores a full set of data for all elements at each step even though most elements are not utilized at every step. Thus, my trace is much more efficient in terms of memory consumption. However, the trace structures presented in their work include links between what they term events, which would correspond to atomic rules within a MT context, and every occurence within the trace for the given event. In this way, their trace stores a richer set of information enabling potential for more complex views of history (*e.g.,* limited history to a view of only

[1] `http://gemoc.org/ins/`

steps where a certain event occurs). Chapter 5 describes a collaborative project with researchers from IRISA and the University of Rennes to bring a more efficient omniscient debugging implementation to GEMOC which employs the generated domain-specific traces introduced in [92].

4.2.3 AToMPM

My omniscient debugger prototype is implemented within the context of AToMPM, 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. The current release version of AToMPM was demonstrated at MoDELS 2013 [23]. 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 [93]. Although the context for exploration of omniscient debugging in this work is AToMPM, the algorithms and general approach of the omniscient debugger as described here can be ported to other modeling tools.

4.3 An Illustrative Omniscient Debugging Scenario

This section describes an illustrative scenario of a developer using an omniscient debugging technique to locate a defect in an MT. This scenario describes the efforts of a developer (who we will refer to hereafter as Anastasia) attempting to find a defect using omniscient features. This illustrative scenario makes use of a model transformation solution for the 2014 Transformation Tool Contest (TTC) Movie Database Case (Movie DB Case) [94] originally presented in [2]. 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

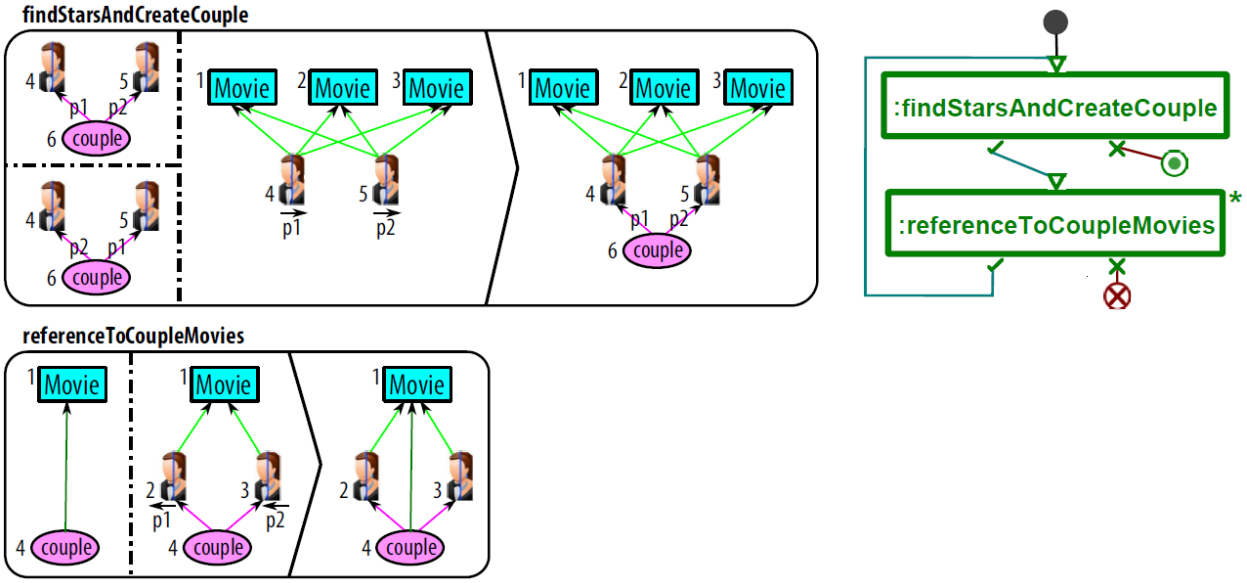


Figure 4.1: Solution to Task 2 of the 2014 TTC Movie Database Case as presented in [2]

broken into three subtasks: generating the data, identifying couples, and averaging the ratings of the movies. The scenario will focus on the second subtask, identifying couples. The solution was developed in MoTif [49] and executed in AToMPM [23]. For the sake of simplicity, this scenario focuses primarily on a specific rule, but the transformation contains many rules. See [94] for the full details of this transformation, and [2] for a solution created in AToMPM. In this example, focusing on a single transformation rule can be likened to focusing on a single method of a Java program.

4.3.1 Transformation Details

The transformation presented in Figure 4.1 (which employs graph grammar rules as discussed in Section 2.1.3) 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,

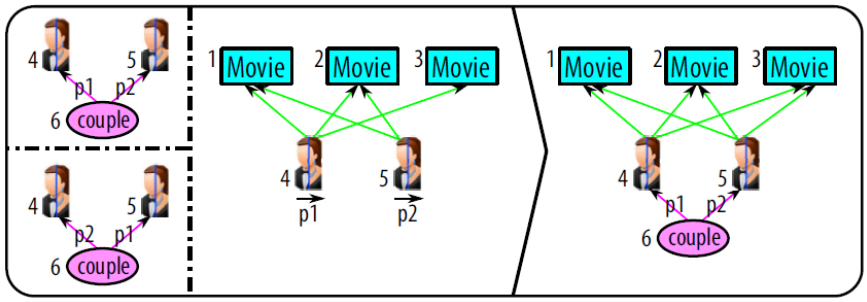


Figure 4.2: Defective Variant of `findStarsAndCreateCouple`

findStarsAndCreateCouple, identifies a pair of actors/actresses who 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 findStarsAndCreate-Couple 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 referenceTo-CoupleMovies fails to apply at least once.

Consider the scenario where the developer must fix a defective implementation of find-StarsAndCreateCouple, as presented in Figure 4.2. The defective rule may identify a pair of actors/actresses that only share two movies (where they should only be matched if the actors share three 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.

4.3.2 Omniscient Debugging Scenario

The developer, Anastasia, might execute a set of test cases where the resulting couples and couple averages (*i.e.,* average of all shared movie ratings for a given couple) are known. In the process of executing the tests, Anastasia notices that in a specific test case the average for a certain couple has been computed incorrectly. She 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 that computed the couple's average, Anastasia 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. She jumps back to Task 1 to observe the actors, movies, and linking edges being generated. However, she immediately finds that the two actors have been correctly generated with the expected movie links. Anastasia now continues re-executing the system to see the couple being created and the movies linked to the couple. She then steps through the defective findStarsAndCreateCouple rule. From this navigation sequence, she is able to observe that the rule has incorrectly matched a movie only connected to a single actor, and she can even back up the system and re-execute to confirm her initial observation. She observes the extra link being created and understands that this rule is the defect causing the error. Anastasia 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.

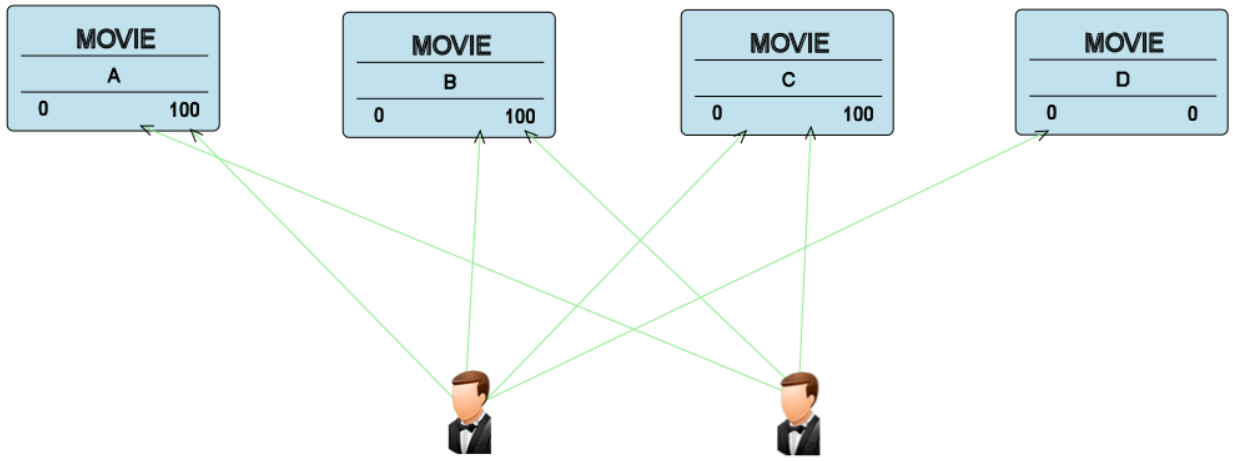


Figure 4.3: Sample Model for Movie DB Case

In the scenario, Anastasia is able to freely traverse the execution history of the system enabling her to directly follow the trail of clues to eventually identify the defective rule. When Anastasia first observes the defective behavior she is even able to immediately revert and re-execute the rule to verify the defective behavior. Anastasia uses the jump feature to quickly move through the system's execution history to an interesting point (*i.e.,* 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. Anastasia made use of numerous basic features (*e.g.,* jump, back, step) to explore the system. If Anastasia had been using a stepwise debugger, she would have needed to restart the system at least twice. The first time, she 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, she would restart when she 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. Individual rules involving complex searches of large models can take 5 to 10 minutes. 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.* describe large models as containing on the order of millions of elements [91]. 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 given the same input conditions. This is due to the nondeterministic behavior of MTs as discussed in Section 2.1.3.1. Consider the sample model presented in Figure 4.3. When processing this sample model, the transformation should couple the two actors and provide a couple average of 100. However, due to the nondeterministic 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 Movie A, B, and D in Figure 4.2). Thus, Anastasia 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. Re-execution may result in the nondeterministic matching algorithm identifying a distinct set of elements that may or may not trigger the defect. Thus, the omniscient debugger prevents Anastasia from losing the context of the defect by making it possible to reach prior states without needing to re-execute the system.

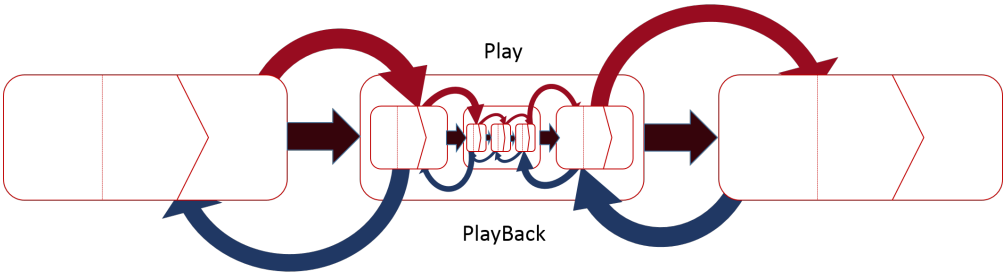


Figure 4.4: Stepwise and Omniscient Continuous Play Features

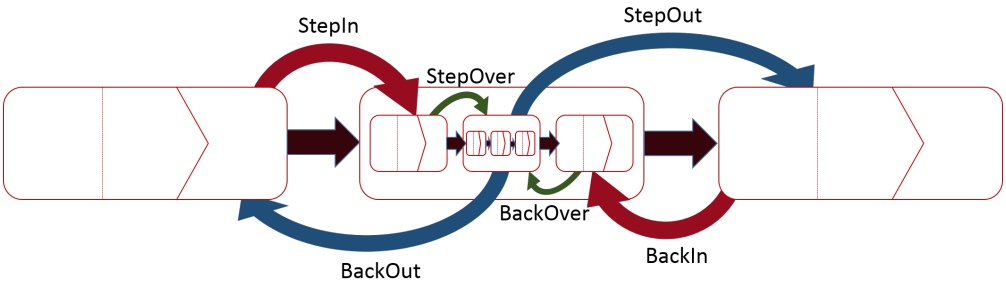


Figure 4.5: Stepwise and Omniscient Step Features

4.4 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 [32, 33]. AODB implements the technique described in this section to support omniscient debugging for MTs. The technique provides the common features of stepwise execution (*i.e.,* `play`, `pause`, `stepIn`, `stepOut`, `stepOver`, and `stop`) [33]. 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. The technique also provides a set of features that mimic stepwise execution, but revert execution. The omniscient features are `playBack`, `backIn`, `backOver`, and `backOut`. This section defines the supported

traversal features (both stepwise execution and omniscient) as illustrated in Figure 4.4 and Figure 4.5, and then discusses how history is collected and stored. Finally, this section also presents an algorithm for more efficient traversal of history.

4.4.1 Execution Traversal Features for Omniscient Debugging

The following defines standard stepwise execution traversal features as 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, executing the same stepwise execution process may occur 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 Triangle transformation (described in Section 4.5.2) 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 multiple rule applications can be reduced to a single set of changes, traversal time can be reduced. In each of these scenarios, the stored history of execution is utilized rather than the execution engine as typical for stepwise execution features. Thus, the implementation of AODB considers both of these scenarios as summarized in the following definitions.

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, a set of additional features is provided 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).

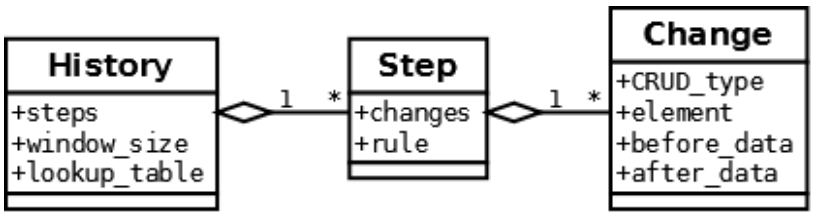


Figure 4.6: Structure of history.

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 [35]) could be introduced to the environment, but these features would be defined using a set of the features defined above. The goal of this chapter is to evaluate the efficiency and scalability of the model transformation engine modified to handle omniscient debugging. Thus, advanced traversal features are not defined here, such as traversing history by navigating through only the steps relevant to a specified element or transformation rule.

4.4.2 Collecting a History of Execution

The technique supporting omniscient debugging collects a history of execution to enable traversal without re-executing rules. Figure 4.6 presents the structure of the trace of execution used in AODB, hereafter referred to as history. History is defined 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 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 the current work 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, two changes would be stored. One for each distinct element, 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 history 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.* [93] 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.

4.4.2.1 *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 changes are defined 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 the technique.

The current space complexity, $O(As+Bc)$, ignores the impact of the revisions cache stored in history, because the cost of the lookup table can be amortized 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, there is a constant amount of increase to the overall size of history. Therefore, B can be redefined 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, the technique maintains a window of active history. As mentioned in Section 4.2, this technique has been explored previously by Lewis [6]. 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.

4.4.3 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, this chapter also explores a technique to efficiently, in terms of execution time performance, revert execution by identifying and executing a minimal set of changes. The technique described here utilizes the execution history to create a macro step that avoids unnecessary CRUD operations. A macro step contains changes from potentially many traditional steps (*i.e.,* those associated with a single rule). Changes store a complete state for the associated element. Thus, if a model element is found in several changes, then the macro step would use only the most recent change and all other changes can be ignored with one exception. If the element has been deleted, it must be recreated and then reset to the correct 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.

4.4.4 Recognizing Patterns of Change

The technique described here 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, an algorithm could recognize a set of special cases where it can ignore changes. The algorithm (defined in Section 4.4.5.1) recognizes five patterns to discern required vs. 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, illustrated in Figure 4.7, 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. Only the update most local to the target step is included. All other updates are ignored.

3. If a create and an update are identified for a given element, both the create and update are included in the minimal set of changes. The create operation is required to recreate the element and the update to reset the element to the appropriate state.

4. If an update and delete are identified for a given element, only the delete is included in the set of changes. Any update can be ignored, because the element will not exist after the set of operations.

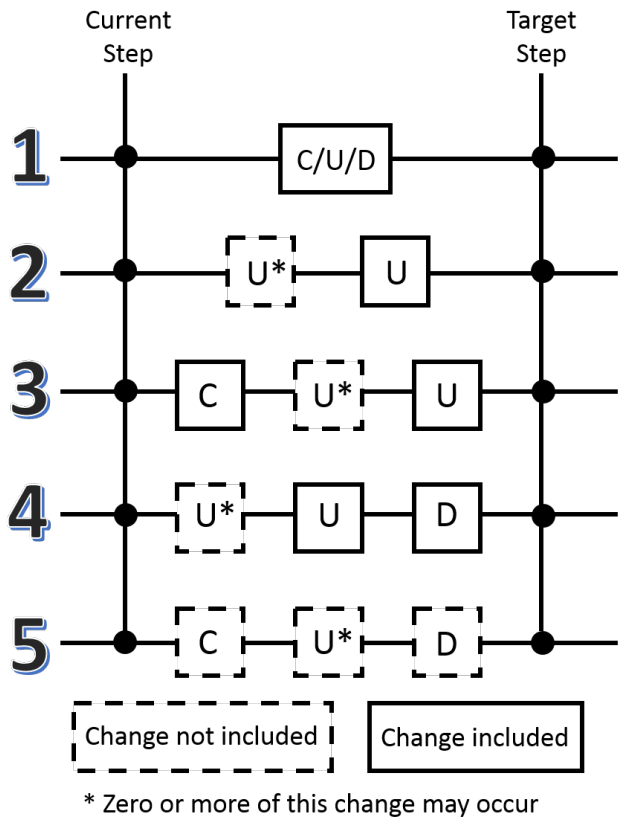


Figure 4.7: Patterns to identify required changes between the current step and a target step.

5. If a create and delete are identified for a given element, no changes are included for this element. All changes can be ignored, 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.

4.4.5 Efficient Omniscient Traversal Using MacroSteps

Thus far, traversal of execution has been defined using Steps, where a Step relates to executing a single transformation rule. However, when traversing through history it is not necessary to re-execute every CRUD operation. The omniscient debugger takes advantage of this fact by constructing and using MacroSteps to traverse history. As illustrated in Figure 4.6, 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 the patterns described in Section 4.4.4 are used to identify unnecessary changes that are then omitted from the MacroStep.

4.4.5.1 Algorithms to Construct a MacroStep

In the 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

```
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 (lastChange 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
```

Figure 4.8: `IterateElements` Algorithm: builds a macrostep by iterating over all elements that have been changed.

```
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
```

Figure 4.9: `IterateSteps` Algorithm: builds a macrostep by iterating over all steps in 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 constant time access to the associated change in history can be guaranteed. The `IterateElements` algorithm, provided in Figure 4.8, uses these facilities to construct a minimal set of changes for a MacroStep to traverse from the current step to the target step.

Assuming an element is changed at some point between the current and target steps, the `IterateElements` algorithm finds the first change and last change within the interval then applies each of the five patterns discussed in Section 4.4.4. The first and second patterns are straightforward. First, if there is only one change for the element then that change must be kept. Second, if there are only updates then only the update closest to the target step is kept. The algorithm can easily detect if 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. When traversing back through history, create and delete Changes are treated as their opposite. Thus, if a create and an update are recognized when moving forward, the algorithm applies pattern 3. If a delete and an update are recognized when moving backward, the algorithm also applies 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 a create and a delete are identified, 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 Figure 4.8 is designed assuming that when building the macro step, 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 is 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, the algorithm still ensures that a minimal set of changes is identified for the MacroStep. The upper bound of overall execution time remains the same, because the algorithm ensures that iterating over the changes will have similar lower bound or it iterates 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 `IterateSteps` algorithm in Figure 4.9 provides the details of how a macro step is generated when iterating over the intervening steps (from current step to target step).

The `IterateSteps` algorithm also identifies each of the five patterns discussed in Section 4.4.4 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, changes are stored in a set of caches that can be referenced later to identify cumulative effects. Each pattern is recognized incrementally until the full pattern has been identified. When the first change for a given element is identified, pattern 1 is assumed and the change is stored in the relevant cache. If no further changes are identified, then pattern 1 is confirmed. If multiple updates are identified, then pattern 2 is recognized and only the update closest to the target step is kept. There are two conditions where pattern 3 can be identified (one for each direction of traversal): 1) when traversing forward and an update is identified after having identified a create, and 2) when traversing backward and an update is identified after having identified a delete. Similarly, there are two conditions where the debugger identifies pattern 4: 1) when traversing forward and a

delete is identified after having identified an update, and 2) when traversing backward and a create is identified after having identified an update. Finally, if a delete is identified and a create has already been identified (or vice versa) then pattern 5 is recognized.

4.4.5.2 Evaluating the MacroStep Construction Algorithms

The `IterateElements` algorithm (Figure 4.8) has $O(n*lg(n)+n*lg(m))$ execution time complexity, which can be simplified to $O(n*lg(n))$ in practice, especially for large-scale models. 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 the algorithm makes use of a structure with constant time access for the relevant change and at least $O(lg(m))$ access to change locations stored in the cache. 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 per element required becomes unreasonable for most transformations. The number of changes for a given element are expected to be small. Thus, the execution time complexity upper bound will 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 macro step. Thus, the `IterateSteps` algorithm (Figure 4.9) has an $O(abs(c_{current}-c_{target}))$, where $c_{current}$ is the sum total number of changes for all steps up to and including the current step, and c_{target} is the sum total number of changes for all steps up to and including the target step. Therefore, the theoretical tipping point for choosing it-

erating over changes (Figure 4.9) rather than iterating over all elements (Figure 4.8) is when the number of changes, $abs(c_{current} - c_{target})$, is less than $n * lg(n)$.

4.4.6 Maintaining Scope in History

As mentioned previously, the omniscient technique described in this chapter is an extension of stepwise execution. As such, the technique described here provides scope-based operations (*i.e.,* `stepIn`, `stepOver`, `stepOut`, `backIn`, `backOver`, and `backOut`) that enable the developer to navigate based upon the current scope (*e.g.,* stepping into or over a given scope). In a traditional GPL such as Java or C++, typically methods or functions are used to define scope. Similarly, in model transformations there exist rules that may contain as part of their definition references to other rules. Thus, scope-based traversal can be used in a MT context as is typical in a GPL context. To support these operations while traversing through history, scope information must be provided. The naive solution is to provide a full scope stack at any given step. By providing the full stack, the debugger can also incorporate stack traces similar to those generated by exceptions in GPLs (*e.g.,* Python, C++, or Java). However, including a full copy of the stack trace for each element can create a state space explosion by replicating scope information between multiple steps. Every step stores a pointer to a step where the current (relative to the step) top of the stack is stored. A given scope's information is stored 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, history stores a single node for each time a scope is encountered. A flyweight pattern could also be applied to this technique [95]. This pattern prevents duplicate primitive constructs by linking all occurences to a single instance. Thus, the flyweight variant would present a minimal trace of scope stack

information through eliminating redundancy of changes and values. Consider if a string value is repeated throughout history that could be replaced with a single object referenced in each location rather than repeating the string value.

4.4.7 Supporting Omniscient Debugging in Other Modeling Platforms

Although the context for exploration of omniscient debugging is AToMPM, the algorithms and general technique of the omniscient debugging prototype implementing the technique described here should be able to be ported to other modeling tools. The 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. This modification should 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. Because these describe basic data structures, 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. A Step in MoTif is defined 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). The 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[2]), but the recognition of scoping will need to be tailored to the specific MTL. This recognition is expected to be a constant time addition to the initial execution time because it is already available within the implementation of AODB as part of the AToMPM environment. Thus, no significant differences in runtime are expected, but the implementation in other modeling tools must be customized to match the relevant MTL structure and semantics.

4.5 Study Design - Evaluating Efficiency and Scalability of AODB

To evaluate the performance and scalability of the omniscient debugging technique, an empirical study utilizing two distinct model transformations was performed. This section describes the design of the empirical study as well as a discussion of the threats to the validity of the results.

4.5.1 Research Questions

The primary goal of the study is to understand the execution time performance and scalability (in terms of memory consumption) of the omniscient debugging technique described in this chapter and implemented in AODB on two model transformations designed in two different MTLs. The focus of the study is to address the following questions:

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

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

RQ 2: Is there a significant difference in execution time between executing a model transformation with or without macro steps?

RQ 3: Is there a significant difference in execution time between the `iterateSteps` algorithm and the `iterateElements` algorithm (presented in Section 4.4.5)?

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

RQ 5: What is the effect on the changes and steps on memory consumption in history?

RQ 6: What is the impact of history on total memory consumption?

Wilcoxon signed-rank tests were performed to resolve *RQ 1*, *RQ 2*, and *RQ 3*. For each hypothesis test, the directionality of the difference was not presupposed during testing. Therefore, each hypothesis test is two-tailed. For each test, a null hypothesis was formulated to evaluate if there is a significant difference between the two sets under comparison. If, after testing the null hypothesis, it can be rejected with a high confidence ($p = 0.05$), then the alternative hypothesis is accepted. Accepting the alternative hypothesis corresponds to there being a significant difference between the two sets. The following presents an example null hypothesis (`H0`) followed by the corresponding alternative hypothesis (`HA`).

`H0 : timewithomniscience = timewithoutomniscience`

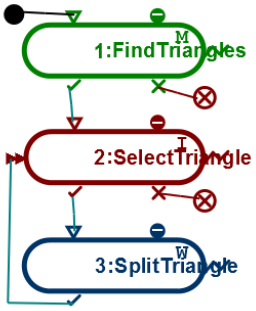
`HA :` $\texttt{timewithomniscience} \neq \texttt{timewithoutomniscience}$

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

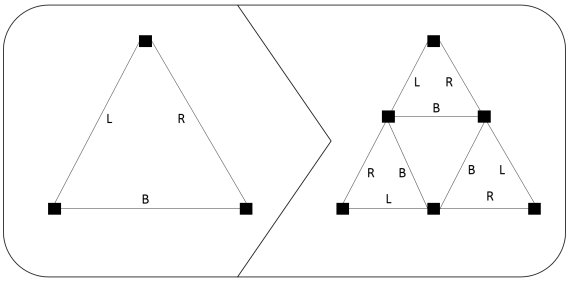
4.5.2 Debuggers and Model Transformations used in Evaluation

In this study, two model transformations were executed 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, `play` was executed until the end of the transformation was reached, and then `BackIn` was invoked repeatedly until the system reverted to the initial step. After returning to the intial step, each step of the transformation was re-executed using `StepIn`. During calls to `BackIn` and `StepIn` to proceed backwards and forwards through the transformation, execution time and memory usage statistics for the history of the transformation were collected. After finishing the `StepIn` task, the collected history was divided into ten partitions resulting in eleven boundary steps. The boundary steps were then used as jump points such that the system jumped from every jump point to every other jump point (*i.e.,* 1 to 2, 2 to 1, 1 to 3, *etc.*).

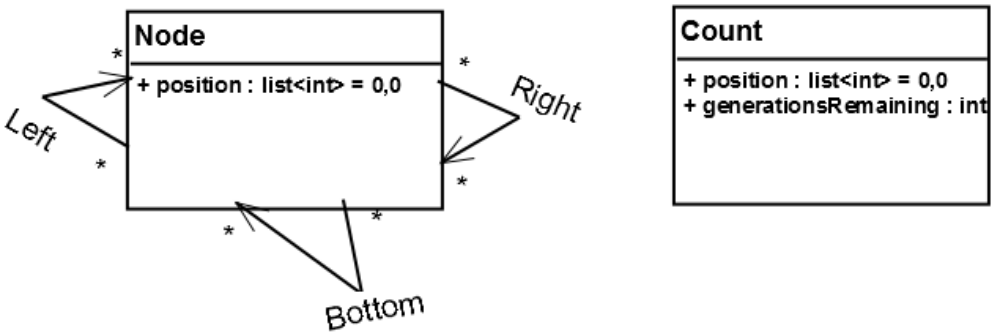
For the stepwise execution, `play` was used to execute the entire transformation. Because history does not exist and the omniscient traversal features are not supported for stepwise execution, this was the only task performed. However, incremental step times were captured throughout the `play` operation enabling direct comparison with the data collected from the omniscient debugger.



(a) Sierpinski Triangle TCore Transformation



(b) Graph Grammar Rule Used by the Sierpinski Triangle TCore Transformation



(c) Sierpinski Triangle Metamodel

Figure 4.10: Sierpinski Triangle TTC Details

The two model transformations selected for this study were acquired from the 2007 and 2014 TTC. The first of these transformations is the Movie DB Case described in Section 4.3 drawn from the 2014 TTC. This study includes models with 116, 580, and 1,160 elements for the Movie DB Case transformation. For details of the Movie DB Case transformation implementation, please refer to the solution from the 2014 TTC by Ergin and Syriani [94]. In this study, the transformation was modified only to connect the three primary tasks within a single transformation.

The second transformation, drawn from the 2007 TTC, constructs a Sierpinski triangle (metamodel provided in Figure 4.10c), 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

Generation	Model Elements
0	12
1	33
2	96
3	285
4	852
5	2,553
6	7,656
7	22,956
8	68,892
9	206,673

Table 4.1: Number of model elements for each generation of the Sierpinski Triangle transformation

three smaller triangles so that each triangle touches the two other triangles at a corner, effectively splitting each triangle into three smaller triangles (Figure 4.10b). The second step is then repeated to create each new generation. Table 4.1 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 4.1 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 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 Section 4.5.5. The Sierpinski Triangle transformation is able to perform a single search for all triangles that must be split to create the next generation. Figure 4.10a displays the T-Core transformation that generates the next generation of the Sierpinski Triangle transformation (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, 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 transformation. 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 transformation always starting with the same initial model for all target generations (*i.e.,* a single triangle representing generation 0).

4.5.3 Measures Used in the Evaluation

A central goal of omniscient debugging is to reduce the time for a developer to return to a previous state in execution. Omniscient debugging achieves this goal 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 AODB, the execution time for each step and jump through history was recorded in milliseconds and compared to the time required for standard execution. The system also recorded which rule was executed at each step. This added information allowed reference back to the transformation to determine the types of operations that were applied during the step.

In addition to the execution time required for moving through history, the system collected 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 Section 2.1.3.2), this additional memory usage is required to recreate the exact changes. For this reason, it is necessary to understand the impact history has on memory usage. To investigate the impact, memory usage (in bytes) was collected 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. Memory usage is also recorded for each macro step built during a jump. Each of these measures are used to give an overall understanding of the impact on the system.

4.5.4 Configuration of Experimental Platform

The study utilized 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[3] 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, communications to the client was disabled. This allowed isolating execution time on the model transformation engine and removed the additional overhead of network transmissions.

[3] `https://pypi.python.org/pypi/python-igraph/0.6.5`

4.5.5 Data Collection and Analysis

The model transformation engine was instrumented to collect information during the execution of the model transformation. For each step, the system recorded 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. The model transformation was then executed in the forward direction, backward through history, forward through history, and jumping through history after splitting the history into ten partitions. Afterwards, each model transformation was re-executed while forcing the macro step to be built using either the `iterateSteps` or `iterateElements` portions of the algorithm. The data collected from these distinct runs was used for comparison of the two techniques.

For *RQ 1*, *RQ 2*, and *RQ 3*, Wilcoxon signed-rank tests were conducted to determine whether a significant difference was observed. The Wilcoxon signed-rank test is the non-parametric analog of the t-test. The directionality of the difference was not assumed when performing these tests.

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

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

For *RQ 4*, cases were identified that highlight the different circumstances that can make traversing through history either faster or slower than re-executing the code. These cases were used 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*, the memory consumption of the system was captured as the model transformation was executed. This information was compared to the amount of information contained in history, as well as to the number of changes and number of steps currently in history (that were also captured during execution). Empirically evaluating these questions provided a sound understanding of the impact of omniscient debugging on the memory usage of the model transformation engine.

4.5.6 Threats to Validity

The study has limitations that may affect the validity of the findings reported here. This section describes the identified limitations as well as the measures taken to mitigate the them.

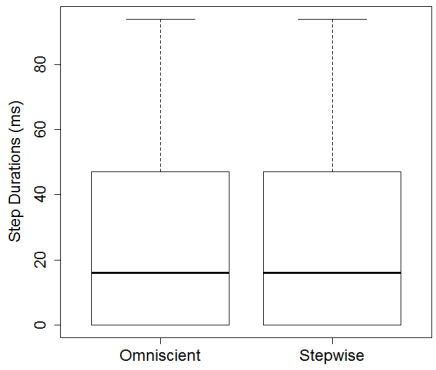
Threats to conclusion validity concern the degree to which the conclusions about the relationships in the data are reasonable. To mitigate these concerns, the comparisons made to execution time on the model transformation system were limited and the effects of other variables were limited as possible, such as communication time to the client. In addition, non-parametric statistical tests were used and no assumptions were made about the distributions of the data (*i.e.,* nonparametric tests were used).

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

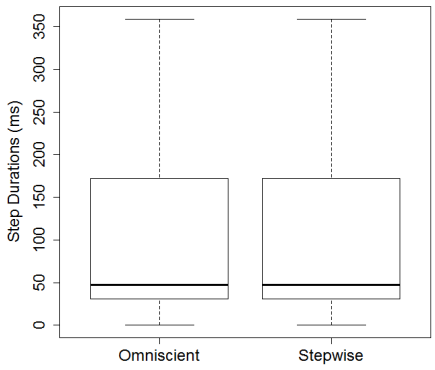
on the host machine have on the time required to execute the steps in the transformation. To limit these issues, a minimal set of processes were allowed to run on the host.

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, and to account for this possibility, logging was instrumented outside of the measured tasks to limit the total impact when possible. When not possible to instrument logging outside the measured tasks, all runs were performed similarly and any impact of the instrumentation was recorded to enable later removal of the impact from recorded observations.

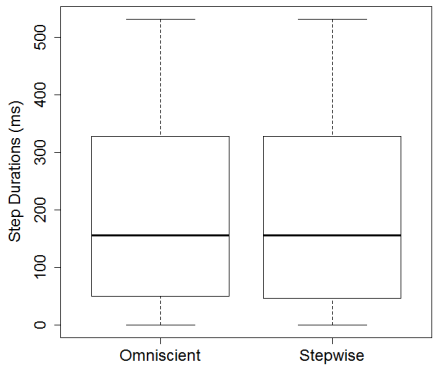
Threats to external validity concern the extent that results can be generalized. Two model transformations were chosen with, intentionally, varying factors to gain a more complete understanding of how the omniscient debugger will work on other model transformations written in these languages. Furthermore, model transformations written in two distinct MTLs were used. Understanding how the technique would be affected by other transformation languages would require replicating the experiments in the languages of interest. However, the observed results should be similar for other model transformations written in T-Core and MoTif, as well as languages with a similar feature sets.



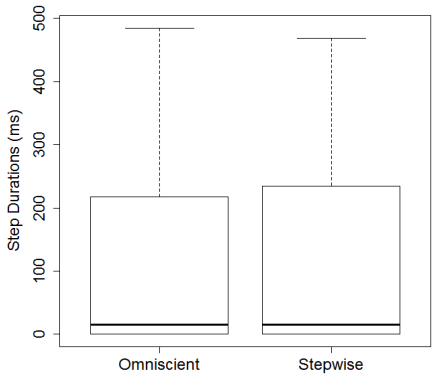
(a) Movie DB Case 1 Iteration



(b) Movie DB Case 5 Iteration



(c) Movie DB Case 10 Iteration



(d) Sierpinski Triangle

Figure 4.11: Measuring differences in execution time: omniscient vs. stepwise (*RQ 1*)

4.6 Results - Evaluating Efficiency and Scalability of AODB

This section presents quantitative data from statistical tests and descriptive statistics organized according to research question being addressed. Section 4.7 will discuss the implications and provide qualitative analysis.

4.6.1 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, the elapsed time was recorded while running each model transformation. Between the two transformations, the elapsed time was recorded for four different

levels: 1, 5, and 10 iterations of the Movie DB Case and 8 generations of the Sierpinski Triangle transformation. The evaluation can be limited 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. The elapsed time was also recorded 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 Triangle 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 Figure 4.11. From these boxplots it can be seen that the times for each step are similar. The biggest difference is visible for the Sierpinski Triangle transformation. A series of Wilcoxon Signed-Rank tests were conducted to determine if a significant difference exists between omniscient 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.

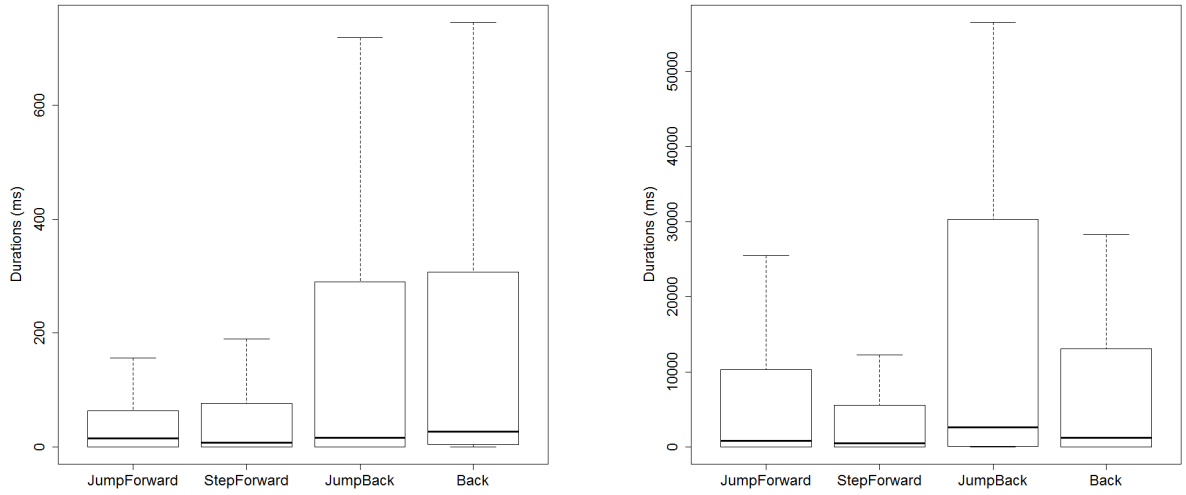
The results of the Wilcoxon tests failed to show any significant difference with the exception of the change step type for 10 iterations of the Movie DB Case model transformation. For this level, building the changes in history for omniscient debugging showed a slight increase in execution time over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

4.6.2 Is there a significant difference in execution time between executing a model transformation with or without macro steps? (*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, MacroSteps are computed 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, history was divided into ten partitions of steps. This resulted in eleven distinct endpoints for moving through history. Then, the elapsed execution time was recorded for jumping from each endpoint to each other endpoint (in both the forward and backward directions). This experiment was replicated for seven distinct levels across the two model transformations including 1, 5, and 10 iterations for the Movie DB Case transformation as well as 3, 5, 7, and 8 generations for the Sierpinski Triangle transformation. Finally, the combined result sets for each model transformation at all levels and the combination of all levels across both transformations were also analyzed.

Figure 4.12a contains boxplots representing the steps for all iteration levels (1, 5, and 10) of the Movie DB Case model transformation. The boxplots show 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, it was observed that moving backward through history results in higher execution times than stepping forward through history.

Figure 4.12b contains boxplots for the Sierpinski Triangle transformation. Unlike the results in Figure 4.12a, where an increase was observed in the spreads for moving (*i.e.,* either step-



(a) Movide DB Case All Levels

(b) Sierpinski Triangle All Levels

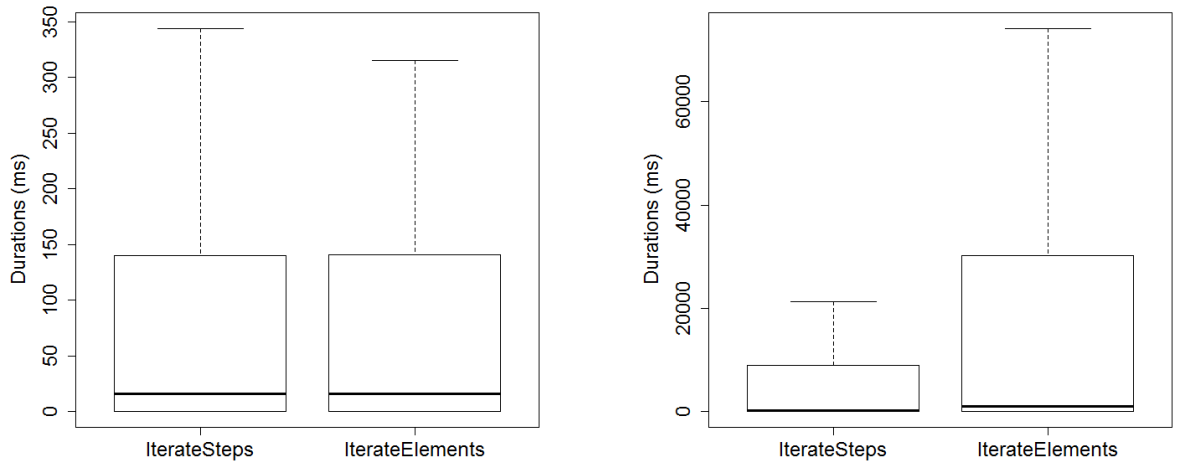
Figure 4.12: Measuring differences in execution time: jump vs. stepping (*RQ 2*)

ping or jumping) forward and backward through history, in Figure 4.12b a decrease was observed in the spread for both stepping forward and stepping backward in history versus jumping.

Wilcoxon Signed-Rank tests were conducted for each of the different levels and combinations. Furthermore, tests were conducted to determine if 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 transformation, 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.

4.6.3 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 normally is used by the MacroStep building



(a) Movide DB Case All Levels

(b) Sierpinski Triangle All Levels

Figure 4.13: Measuring differences in execution time: IterateSteps algorithm vs. IterateElements algorithm (*RQ 3*)

algorithm when the number of steps is less than the number of elements in the system. The second is `iterateElements`, which normally is used by the MacroStep building algorithm 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, MacroSteps were forced to be built in one of the two paths (without regard for the number of steps and number of elements in history) and then observations of execution time were compared. The same levels as described for *RQ 2* were used here.

Figure 4.13a 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 4.13b shows the spreads for all iterations of the Sierpinski Triangle transformation. In the case of the Sierpinski Triangle transformation, the `iterateElements` was observed to have a large increase in spread versus `iterateSteps`.

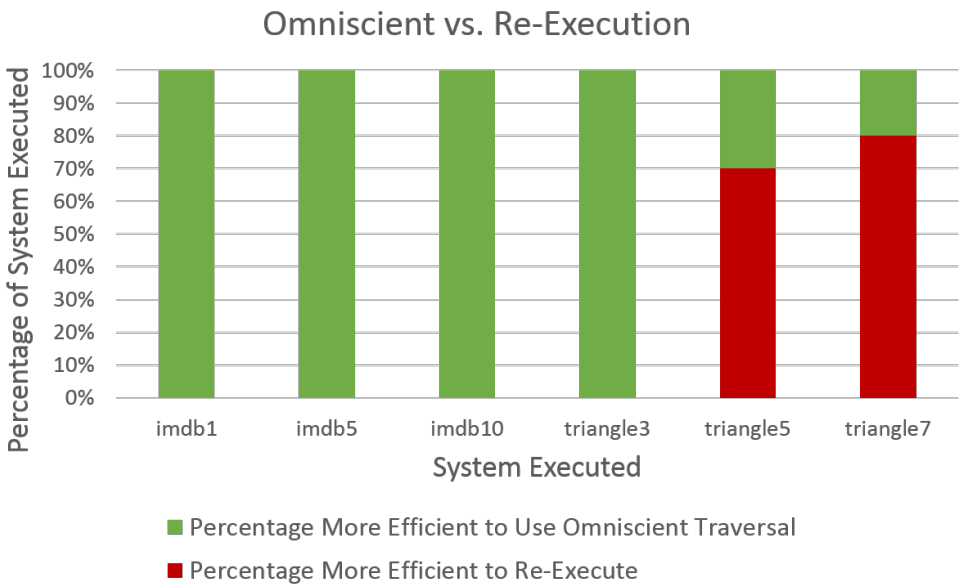


Figure 4.14: From the end of execution, what percentage of the system can be re-executed before omniscient traversal is more efficient. (*RQ 4*)

Again, Wilcoxon Signed-Rank tests were conducted to determine whether there was a significant difference between `iterateSteps` and `iterateElements` at any level. For the Movie DB Case, no significant difference was identified at any level. However, for the Sierpinski Triangle transformation and for the combination of the two systems, a significant difference was identified for every level.

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

It is possible 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, the elapsed time was recorded when executing from beginning to end of the model transformation, as well as when jumping back to different points in history. The case of jumping from the end of the model transformation to a previous point was then compared to that of re-executing to reach the same point.

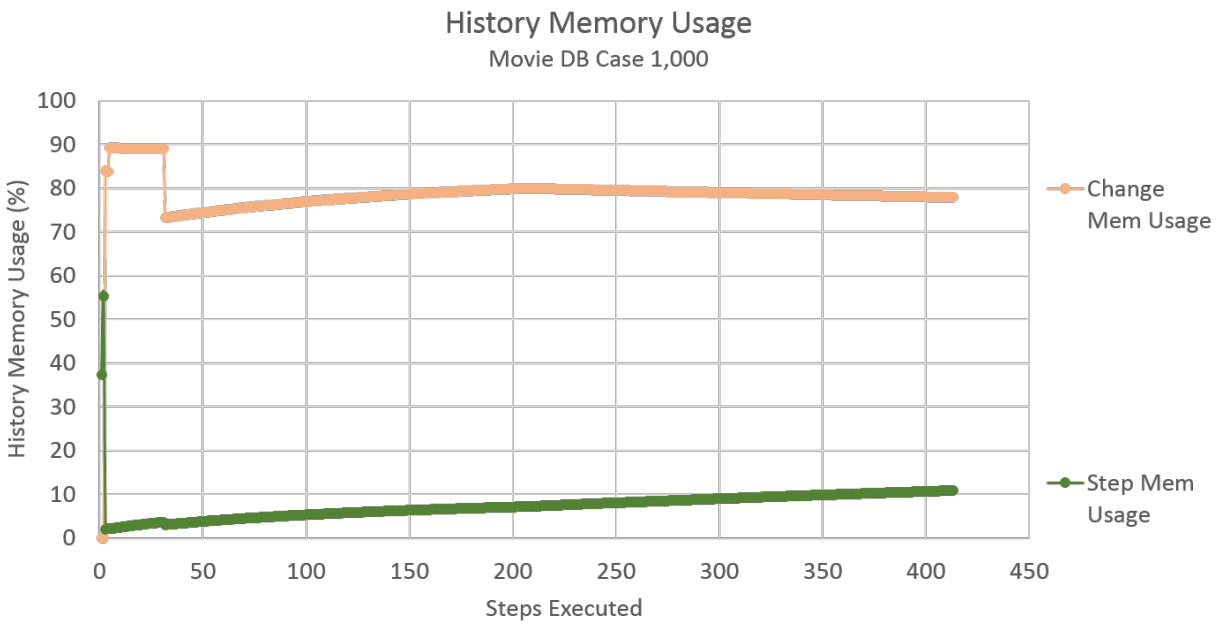
Figure 4.14 shows the tipping point when re-execution becomes better than omniscient traversal. In other terms, the graph shows the percentage of the model transformation that may be re-executed before it would be better to jump back from the end. For all levels of the Movie DB Case, and for 3 iterations of the Sierpinski Triangle transformation, it was always better to jump back from the end than to re-execute the transformation. For 5 iterations of the Sierpinski Triangle transformation, 70% of the model transformation may be re-executed before jumping back becomes better. For 7 iterations, the tipping point is 80%.

It is important to note that the performance focus of this discussion does not consider the importance of nondeterminism as described in Section 2.1.3.1. This question focuses only on evaluating matters of execution time. However, many situations exist where re-execution might lose the context of a defect (as illustrated in Section 4.3), and in these situations re-execution should not be used as a replacement for omniscient traversal.

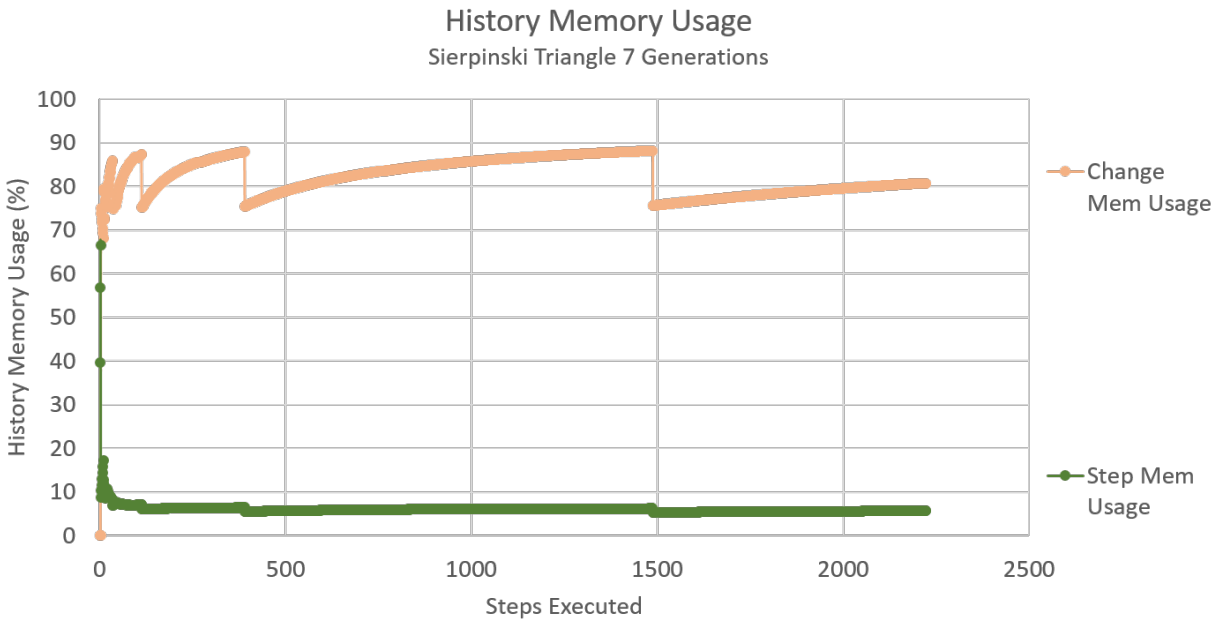
Multiple factors and characteristics influence the observed results as discussed further in Section 4.7.

4.6.5 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 history (see Section 4.4.2), AODB was instrumented to record 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, the changes included in the step were omitted to focus memory usage of a step on the other step features (*e.g.,* scope information and transformation rule information). After the system executed each subsequent step of the transformation, the percentage of memory composed of changes and the percentage composed of steps was calculated. The results of this analysis for 7 generations of



(a) Movie DB Case 10 iterations



(b) Sierpinski Triangle 7 Generations

Figure 4.15: History Memory Usage (*RQ 5*)

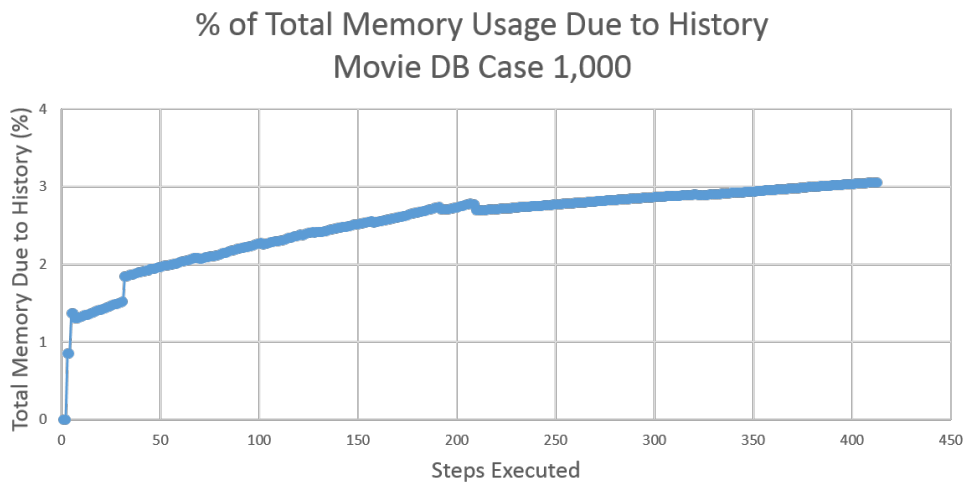
the Sierpinski Triangle transformation are shown in Figure 4.15b. The results of this analysis on the Movie DB Case with 10 iterations are shown in Figure 4.15a.

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 transformation, sudden drops are observed in the memory usage of changes with each new generation of the transformation.

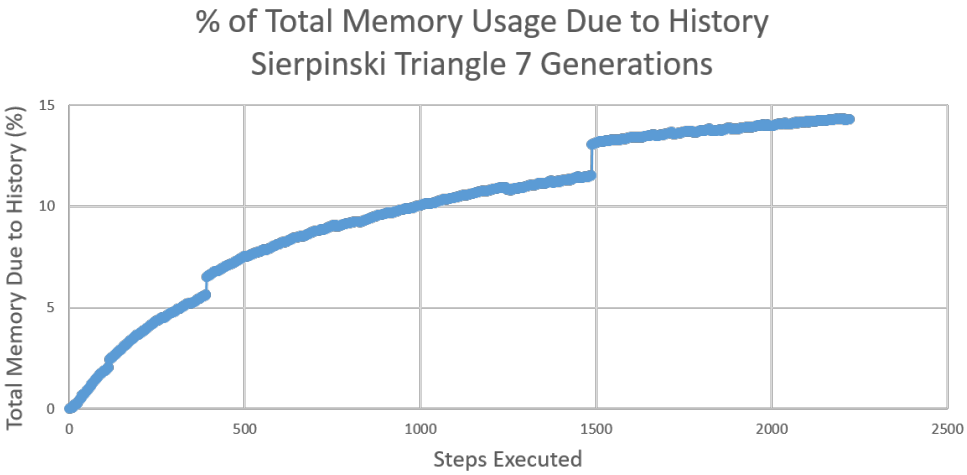
4.6.6 What is the impact of history on total memory consumption? (*RQ 6*)

In addition to the amount of memory used by changes and steps in history, the percentage of memory used by history for the entire model transformation engine is of interest, because it indicates the significance of history to overall memory usage. To calculate the ratio of history memory usage to overall memory usage, the system recorded the total amount of memory usage by history as well as the total memory usage of the entire model transformation engine. Figure 4.16b presents the results of this analysis for 7 generations of the Sierpinski Triangle transformation. Figure 4.16a 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 transformation is always below 15%.



(a) Movie DB Case 10 iterations



(b) Sierpinski Triangle 7 Generations

Figure 4.16: Percentage of total memory usage for the transformation engine due to history (*RQ 6*).

4.7 Discussion - Evaluating Efficiency and Scalability of AODB

This section addresses each research question providing a discussion about the implication of the results presented in Section 4.6. The section is organized according to the research question being addressed.

4.7.1 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 research question was to show that there is not a significant difference between running the model transformation using normal stepwise execution and omniscient debugging. Because building the dynamic structure of 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.

Three step types were chosen to be included in 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, a significant difference in observed execution times was not expected except for those produced from noise external to the system. 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. Similarly, a significant difference was not expected in this case. However, for the change step types, changes must be added to the trace in history. Thus, with enough changes, there could 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 because it indicates that building the history should not significantly decrease execution time performance of a model transformation engine.

4.7.2 Is there a significant difference in execution time between executing a model transformation with or without macro steps? (*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. To facilitate such a traversal, an algorithm was implemented that builds a MacroStep with all changes from the current step to the target step (see Section 4.4.5). Additionally, while building the MacroStep the debugger 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 macro step 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 all changes are executed for each step iteratively until the traversal is complete. The steps in history were partitioned into ten different sets. The partitioning resulted in eleven endpoints. For each of the eleven endpoints, the debugger jumped from each point to every other point in both the forward and the backward directions. For each direction, the elapsed time was computed for traversing using back and `stepForward` through history. Then, the times were compared for each of these moves. Observations showed 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 transformation, a significant difference was

detected between the jumps and `stepForward` and back. The main cause of this difference is a result of the difference in performance between `iterateSteps` and `iterateElements`. Thus, a result of these observations is that the performance of jump could be improved 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 a large increase in changes was found for moving backward versus moving forward resulting in added computation in the backwards 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.

4.7.3 Is there a significant difference in runtime between the `iterateSteps` algorithm and the `iterateElements` algorithm? (*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 transformation, 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 transformation, 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 (*e.g.,* attempting to update an attribute before an element has been created), the resulting set must have 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, it can be concluded 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.

4.7.4 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.

These boundary points were studied for all levels of the two model transformations to identify when it would be better to select re-execution over omniscient traversal. The history for each execution was divided into 10 partitions which enabled comparisons between re-executing from the first step to the target and reverting from the last step to the target. The set of targets included the beginning, end, and the boundaries between each of the 10 partitions of history (*i.e.,* 0% executed, 10% executed, 20% executed, 30% executed, 40% executed, 50% executed, 60% executed, 70% executed, 80% executed, 90% executed, 100% executed). 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. The Sierpinski Triangle 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, omniscient 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.

4.7.5 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, AODB recorded the amount of memory used to store the changes and the amount used to store the overhead of the steps (*i.e.,* the memory usage of the steps not including any contained changes). Only these two concerns can be considered because the cost of the revisions cache can be amortized among all changes to produce a constant increase per change, because the revisions cache stores an entry for each change that has occurred. 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, it was observed that the main contributing component to the memory consumption of history for

the observed systems (Sierpinski Triangle 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 notable reductions were observed in the percentage of history's memory usage due to the changes (as seen in Figure 4.15). These reductions are observed in both model transformations. For the Sierpinski Triangle, the model transformation is written in T-Core, which maintains an internal store of dynamic information that 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 Triangle 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 Triangle transformation.

4.7.6 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, the percentage of the model transformation engine's memory accounted for by history was mapped out 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 Triangle transformation grew at a roughly logarithmic rate as the number of executed steps in the transformation increased. Additionally, the Sierpinski Triangle 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, it was observed that the size of a model and other factors seem to be contributing more significantly to memory consumption than history.

4.7.7 Evaluating the efficiency and scalability of the 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 discussed in this chapter can be used by a developer during MT debugging. Due to the nondeterministic 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.

The results of the study 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, memory usage of the history structure required to make omniscient debugging possible was examined. History was found to have an overall 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 the MacroStep building algorithms, a significant difference was found between iterating through the steps (`IterateSteps`) and iterating through the elements (`IterateElements`). Analysis of the results indicated the performance could be improved by altering the cutoff points for using one algorithm over the other. This alteration should lead to significant gains when jumping through history and, thus, improve the overall omniscient debugging process.

Finally, even in cases where nondeterminism was not a factor and a model transformation could simply be re-executed by the developer, it is still often superior with regards 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 the performance of building the

macro-steps are improved (as discussed previously), this difference should grow more prominent for all cases.

4.8 Conclusions

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. This chapter 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. The technique provides 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. The chapter discussed the use of a trace of execution history to enable the various omniscient traversals. Finally, an algorithm was introduced that efficiently identifies a minimal set of changes, MacroStep, that 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.

The chapter also provides discussion of the theoretical execution time performance scaling of the MacroStep building algorithm and the theoretical memory consumption scaling of history. This theoretical discussion was followed 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.

The work presented in this chapter has focused on the application of omniscient debugging to an MT environment with specific focus on two transformation languages. However, modelers may also make use of directly executable modeling languages, and especially xDSMLs where the language and execution semantics are specific to a given domain. Several recent efforts have explored providing facilities to design xDSMLs [52, 96, 97]. Chapter 5 presents an exploration into applying omniscient debugging within this new context, as well as introducing a history visualization strategy and more advanced omniscient traversal features that build upon the work presented in this chapter.

CHAPTER 5

MULTIDIMENSIONAL OMNISCIENT DEBUGGING FOR XDSMLS

Omniscient debugging is a promising technique that relies on execution traces to enable free traversal of the states reached by a system during an execution. While Chapter 4 explored omniscient debugging for MDE in the context of MTs, modelers may also make use of xDSMLs where the processes inherent to a given domain may be defined directly within the language semantics. One solution to providing omniscient debugging for xDSMLs is to define a generic omniscient debugger for all xDSMLs. However, generically supporting any xDSML both compromises the efficiency and the usability of such an approach. The contribution of this collaborative work relies on a partly generic omniscient debugger supported by generated domain-specific trace management facilities. Being domain-specific, these facilities are tuned to the considered xDSML for better efficiency. Usability is strengthened by providing multidimensional omniscient debugging. Results show that our approach is on average 3.0 times more efficient in memory and 5.03 more efficient in time when compared to a generic solution that copies the model at each step. Additionally, in this chapter, a novel set of omniscient traversal features are described, and a novel visualization of history is presented.

5.1 Introduction

Many recent efforts have explored providing tools and techniques to support xDSMLs [52, 96, 97], which allow system engineers to analyze behavioral properties early in the development process. Debugging is a common dynamic feature to observe and control an execution in order to better understand a behavior or to look for the cause of a defect. However, standard debugging features only provide the ability to pause and step forward during execution. Thus, developers must restart from the beginning to obtain a second look at a state of interest. To cope with this deficiency, a technique supporting omniscient debugging of MTs was explored in Chapter 4. However, when designing a technique to support xDSMLs, the variety of language semantics and features must be considered. Despite the specificities of each xDSML, it is possible to identify a common set of debugging facilities applicable to any xDSML. Thus, to avoid manual creation of each debugger, a possible solution is to define a generic omniscient debugger that would work for any xDSML. However, providing generic debugging facilities to handle any xDSML has two main consequences:

1. There is a trade-off between genericity and efficiency of the debugging operations, because supporting any xDSML requires the use of expensive introspection, conditionals, or type checks to support a wide variety of abstract syntax and runtime data structures. Moreover, because debugging is an interactive activity, responsiveness is of primary importance. Hence, the efficiency of a generic debugger is a primary concern.

2. The execution data structure defined in an xDSML can be arbitrarily complex (*e.g.,* a large object-oriented structure), and therefore difficult to comprehend in a debugging session, especially if the execution leads to a large number of states.

Another key concern is the usability of omniscient debugging for xDSMLs, and specific advanced facilities can significantly aid in managing the complexity and size of the executions. The following summarizes key objectives that drive the focus of this chapter:

Objective 1: Support efficient omniscient debugging, to ensure responsiveness of the debugger.

Objective 2: Offer advanced omniscient debugging traversal features, to improve the usability.

Previously, I investigated efficient, generic omniscient debugging services for model transformations [34] (presented in Chapter 4), and Bousse *et al.* explored generating domain-specific trace metamodels [92]. In the work presented in this chapter, we collaborated to explore adapting domain-specific trace metamodels to support generic omniscient debugging services for xDSMLs. To address Objective 1, this chapter presents a generative approach to define omniscient debuggers. Such a generative approach can provide an efficient and finely tuned omniscient debugger for any xDSML. Yet, considering a generic set of debugging services for all xDSMLs, both the interface and some underlying logic of the debugger can remain generic without compromising efficiency. Hence, the contributions of this chapter rely on a partially generic omniscient debugger supported by generated domain-specific trace management facilities. The trace management facilities include a domain-specific trace metamodel that precisely captures the execution state of a model conforming to the xDSML, and a domain-specific trace manager providing all the required services to manipulate the execution trace generically. Because the trace manager is domain-specific, it is finely tuned to the considered xDSML and to the generated trace metamodel, and hence more efficient than a generic one. To address Objective 2, the contributions of this chapter provide multidimensional omniscient debugging services, which mix both omniscient debugging services, and advanced facilities to navigate among the values of specific elements of the executed model.

In this collaborative project, my primary contributions were as follows:

- designing the omniscient debugging features
- designing and prototyping the visual interface to support the new debugging features

I also contributed significantly to the following components of the collaborative work:

- designing an interface between the generated traces and the omniscient debugging features
- designing the empirical evaluation
- evaluating the results of the empirical study

We implemented our approach as part of the GEMOC Studio[1], a language and modeling workbench, and we conducted an empirical evaluation. To evaluate the efficiency of our solution, we assessed its quality with regard to both memory consumption and the time required to run omniscient debugging operations. We compared our approach with two generic omniscient debuggers: one that simulates omniscient debugging by resetting the execution engine and re-executing until the target state is reached, and one that copies the model at each execution step. Obtained results show that our approach is on average 3.0 times more efficient in memory when compared to the second debugger, and respectively 54.1 and 5.03 times more efficient in time when compared respectively to the first and the second debugger.

The remaining sections are as follows. Section 5.2 defines the considered scope of model execution and model debugging. Section 5.3 presents our generative approach to provide generic multidimensional omniscient debugging. Section 5.4 describes a prototype supporting the tech-

[1] `http://gemoc.org/studio/`

nique in the GEMOC Studio. Section 5.5 discusses the evaluation of our approach. Finally, Section 5.6 discusses related work and Section 5.7 concludes the chapter.

5.2 Model Debugging

This section defines the scope of the advanced omniscient debugging facilities proposed to analyze executable models.

5.2.1 Debugging Approaches

Debugging an executable model involves controlling the model's execution and observing the states traversed. Figure 5.1 shows four approaches to achieve this, with different levels of control over the execution. First, stepwise debugging only traverses forward through the states reached by the model through the application of the operational semantics rules. Second, we call weak omniscient debugging the possibility to go backward in the exploration of the states through a restart of the execution engine and a re-execution until the target state is reached. Note that this can be accomplished manually with any stepwise debugger. Third, omniscient debugging relies on an execution trace to revert the executed model into a prior state. Using a trace makes the procedure deterministic (*i.e.,* the exact same states are visited) even if the model or the operational semantics are nondeterministic. Finally, our proposal relies on multidimensional omniscient debugging, which adds facilities to navigate among the values of mutable fields of the model. In the remainder of this section, we present these debugging approaches as sets of provided services. Note that all these services are only valid when the execution is paused; *i.e.,* when the execution engine waits for instruction before applying a transformation rule.

Stepwise Debugging Most debuggers only provide stepwise debugging, as discussed previously in Section 4.4.1.

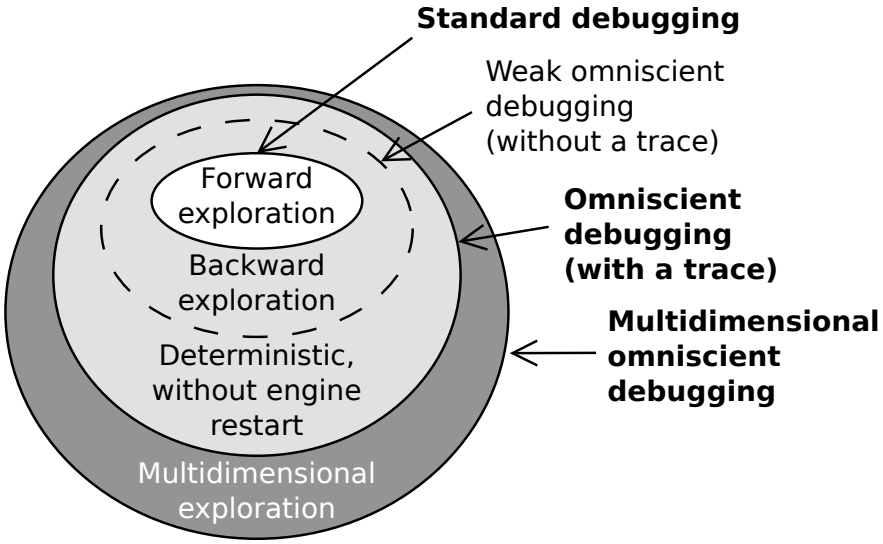


Figure 5.1: Feature comparison of the debugging approaches

Omniscient Debugging To explore previously visited states, omniscient debugging relies on the construction of an execution trace to extend stepwise debugging as discussed previously in Section 4.4.1.

Multidimensional Omniscient Debugging With the ability to go both forward and backward, a developer can explore any state of a model's execution. Yet, large traces are difficult to navigate practically, and information stored within a state can be arbitrarily complex, compromising usability (Objective 2). To cope with this issue, we investigate multidimensional omniscient debugging; *i.e.,* facilities to navigate among the values of the mutable fields of the model:

- ***jumpValue***: jump to the first state in which a given mutable field has a given value.
- ***stepValue***: given a mutable field, jump to the next value of this field.
- ***backValue***: given a mutable field, jump to the previous value of this field.
- ***visualization of the value sequences***: display an interactive representation of the reached values of the mutable fields and show which values are the current ones.

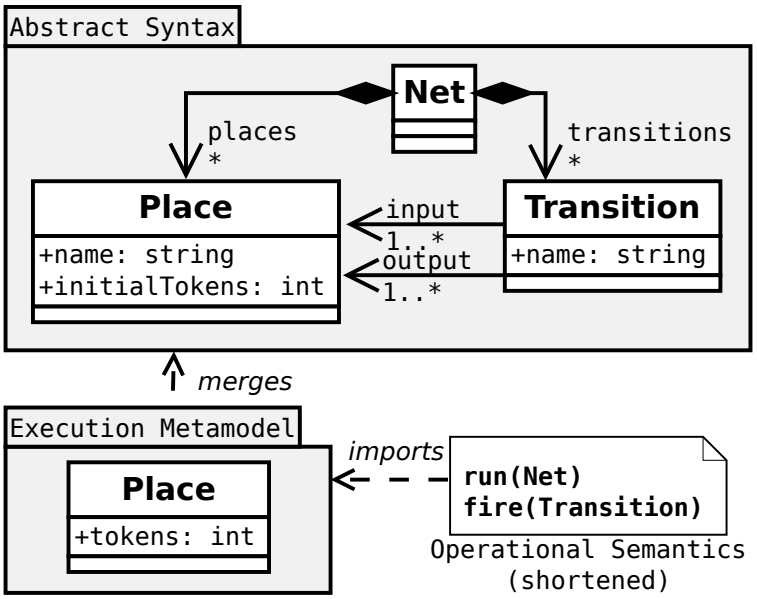


Figure 5.2: Petri net xDSML

5.2.2 Sample xDSML

Before covering a sample scenario that illustrates the omniscient debugging features, the following is a simple xDSML to be used during the scenario.

Figure 5.2 shows an example of a Petri net xDSML. At the top, its abstract syntax is depicted with three classes `Net`, `Place` and `Transition`. At the bottom-left is the execution metamodel, that extends the class `Place` using package merge with a new mutable property `tokens`. The initialization function (not shown) transforms each original object into an executable object (*e.g.,* a `Place` object gains a `tokens` field) as defined in the execution metamodel. The function also initializes each `tokens` field with the value of `initialTokens`. At the bottom-right are the signatures of the rules defined in the operational semantics. The rules are defined as follows: `run` continuously looks for enabled transitions, and uses `fire` to transfer tokens from input to output places of such transitions. Note that a `Place` object cannot be created during execution, because the `Place` class was introduced in the abstract syntax.

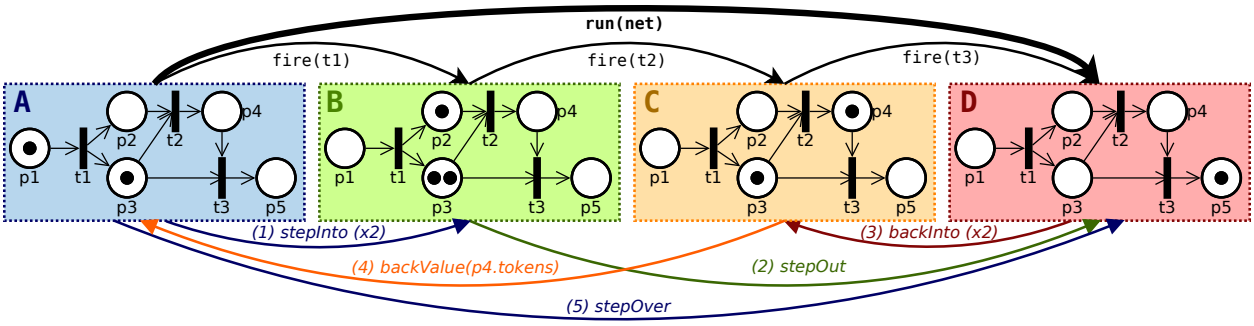


Figure 5.3: Example of Petri net execution trace annotated with the use of a selection of debugging services

Figure 5.3 presents a trace from the execution of a Petri net model conforming to the Petri net xDSML shown in Figure 5.2. The trace is composed of four states, on top of which the steps of the execution are depicted. States are separated by three *small steps* that represent the applications of the *fire* transformation rule. A *big step* goes from the first state to the last state to represent the application of the *run* rule.

5.2.3 Example Debugging Scenario

Consider a complete execution and debugging scenario with a Petri net model conforming to the xDSML shown in Figure 5.2. The initial state of the considered Petri net model is depicted at the left of Figure 5.3 with the label A. First, we set a `breakpoint` to pause execution at state `A`. Next, we execute `stepInto`, which does not change the current state, but enters into the execution process of the `run` rule pausing immediately before executing the first `fire` rule. Then, we execute `stepInto` a second time, which applies the `fire` rule and brings us to state `B`. From there, we use `stepOut` (2) to move out of the current *big step* (*i.e.,* `run`), which brings us to state `D`. At this point, the system has been completely executed, and the trace is fully constructed Thus, no additional transformation rules will be applied, but we can still traverse the system using omniscient traversals.

Similar to the beginning of the scenario, we execute `backInto` twice (3) to move into `run` and revert the last `fire` to reach state `C`. Then, we use `backValue` (4) to go back to the previous value reached by the `tokens` field of the `p4` Place object. While `p4` has one token in state `C`, its previous amount was zero, which started in state `A`. Using this traversal we have moved directly to the previous state of interest for the `p4` Place object avoiding needing to manually traverse and inspect the intervening steps. Hence, we reach state `A` again. Finally, we use `stepOver` (5) to execute the entirety of `run` without considering each contained rule separately, and we reach state `D` again. Note that in this case, `stepOver` (despite being a common stepwise execution feature) does not execute any transformation rules, but simply uses information from the execution trace to directly revert the executed model into the stored state `D`.

5.3 Efficient and Advanced Omniscient Debugging for xDSMLs

This section presents our approach that provides efficient and advanced omniscient debugging for xDSMLs using a partially generic, multidimensional omniscient debugger supported by generated domain-specific trace management facilities.

5.3.1 Overview of the Approach

Defining an xDSML implies the definition of a number of domain-specific facilities to edit or analyze a model conforming to the language. In particular, one method to provide a visual animation of a model execution is to observe the model and react to changes. Because such a pattern is common when defining tools for xDSMLs, our approach is designed to have a single instance of the executed model loaded at any given time that can be modified throughout the execution and the debugging session.

Figure 5.4 shows an overview of our approach. We assume the initialization function of the xDSML was already applied to an input model, creating the executable model. The first step

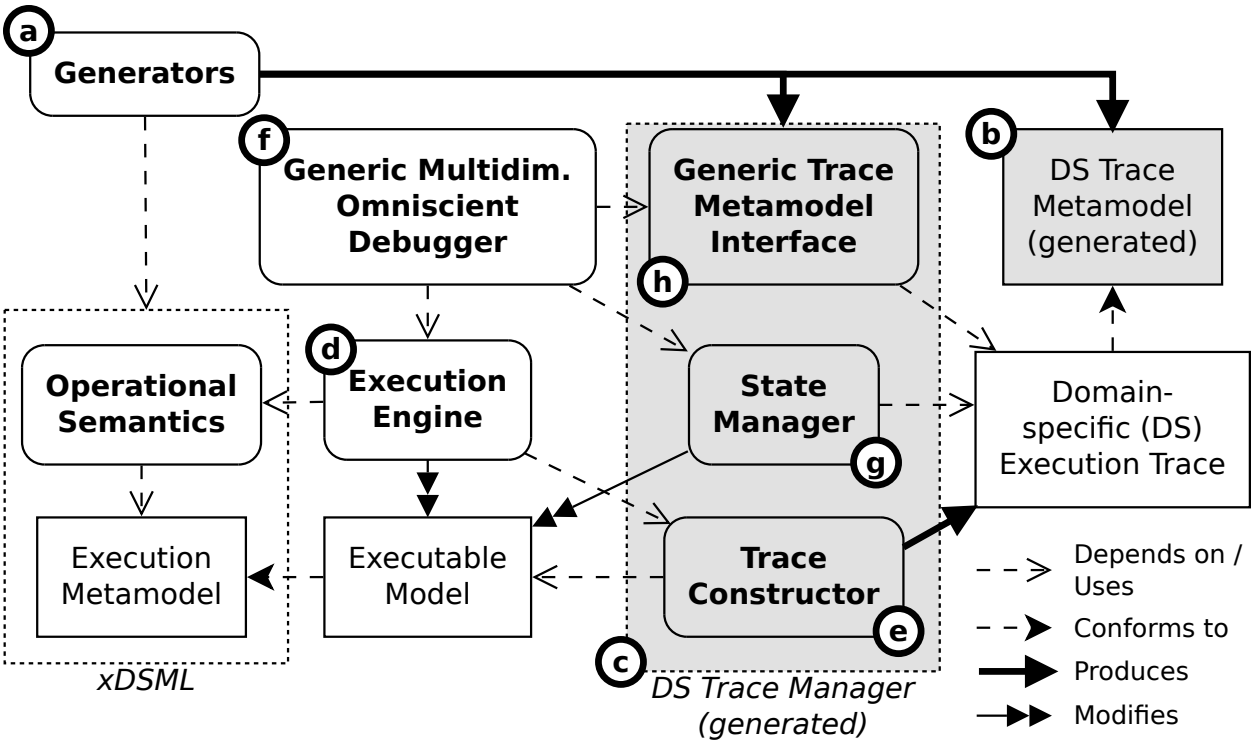


Figure 5.4: Overview of the approach

of our approach relies on generators (a), which take the considered xDSML as input to produce two domain-specific components: a trace metamodel (b) and a trace manager (c). The second step is the execution and the debugging of the model. The execution engine (d) applies the operational semantics to change the model and uses the trace constructor (e) from the trace manager to construct a domain-specific trace. The generic multidimensional omniscient debugger (f) provides all the services described in Section 5.2.1 by controlling the execution engine and relying on the state manager (g) to revert the model into previous states. Additionally, the debugger relies on the generic trace metamodel interface (h) to manipulate the trace.

To illustrate a subset of the interactions between the components shown in Figure 5.4, Figure 5.5 shows a sequence diagram that sketches what happens when a *small step* must be computed and stored in the trace. Duration bars depicted in gray represent changes made in the affected el-

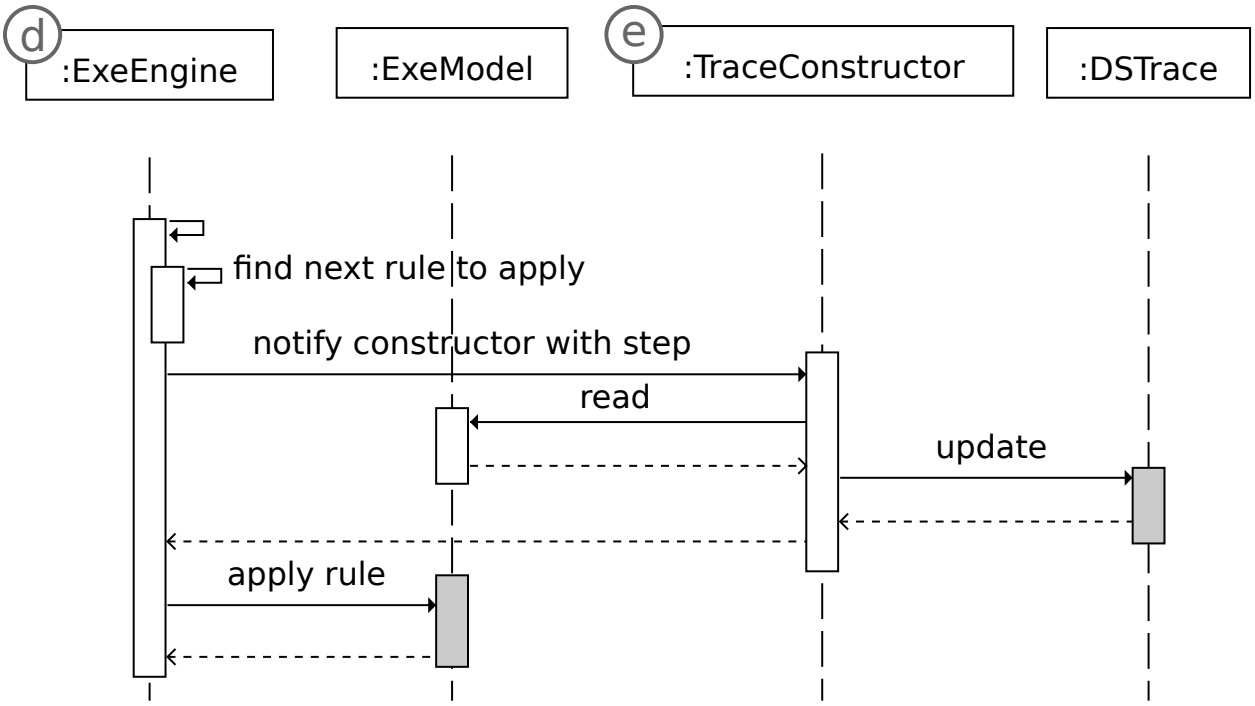


Figure 5.5: Interactions when a *small step* is to be computed and added to the trace

ement. First, the engine (d) determines the next rule to apply then notifies the trace constructor (e) that a small step will occur. As a result, the trace constructor reads the executed model, and updates the domain-specific trace with new elements accordingly (*e.g.,* add a new *small step* and, if the model was altered, a new state). Finally, the execution engine applies the rule and modifies the executed model accordingly.

We present all of these components in more detail in the remainder of this section.

5.3.2 Execution Engine

First and foremost, an omniscient debugger must provide precise control over the execution of a model, such as the ability to pause during execution or traverse the trace in a controlled manner. For this to be possible, the execution engine (d) must adhere to certain specifications. The engine must be able to drive the execution of the model (*i.e.,* initialization, start, stop), and to provide the debugger some control over the execution. This includes the ability to pause the execution at a

specific state during execution, and the ability to resume the execution from a paused state. We assume that the engine provides at least the following services:

- ***pauseWhen****:* suspend the execution in between two transformation rule applications as soon as a given predicate is `true`.
- ***isPaused***: return `true` if the engine is paused.
- ***resume***: resume execution (*i.e.,* cancel a pause).

As presented in Section 5.4.1, we developed an execution engine that encompasses the aforementioned services.

5.3.3 Domain-Specific Trace Metamodel

In the prior work of Bouse *et al.* [92], we presented a generative approach that automatically provides a rich domain-specific trace metamodel for an xDSML. Instead of relying on clones of the executed model to construct a trace, the metamodel precisely captures its execution state through an efficient object-oriented structure based on the mutable properties of the xDSML. In addition, the structure provides rich navigation facilities to browse a trace according to the values reached by the mutable fields of the model. To benefit from such efficient trace structures (Objective 1), we rely on this approach for the automatic generation (a in Figure 5.4) of the domain-specific trace metamodel (b in Figure 5.4).

To support omniscient debugging, we extended this generative approach with the notion of *big step*, in addition to *small step*. This included adding a `BigStep` class to the base classes that are generated, the derivation of domain-specific *big step* classes from the xDSML operation definitions, and extending the generation algorithm to create inheritance links from *big step* classes to `BigStep`.

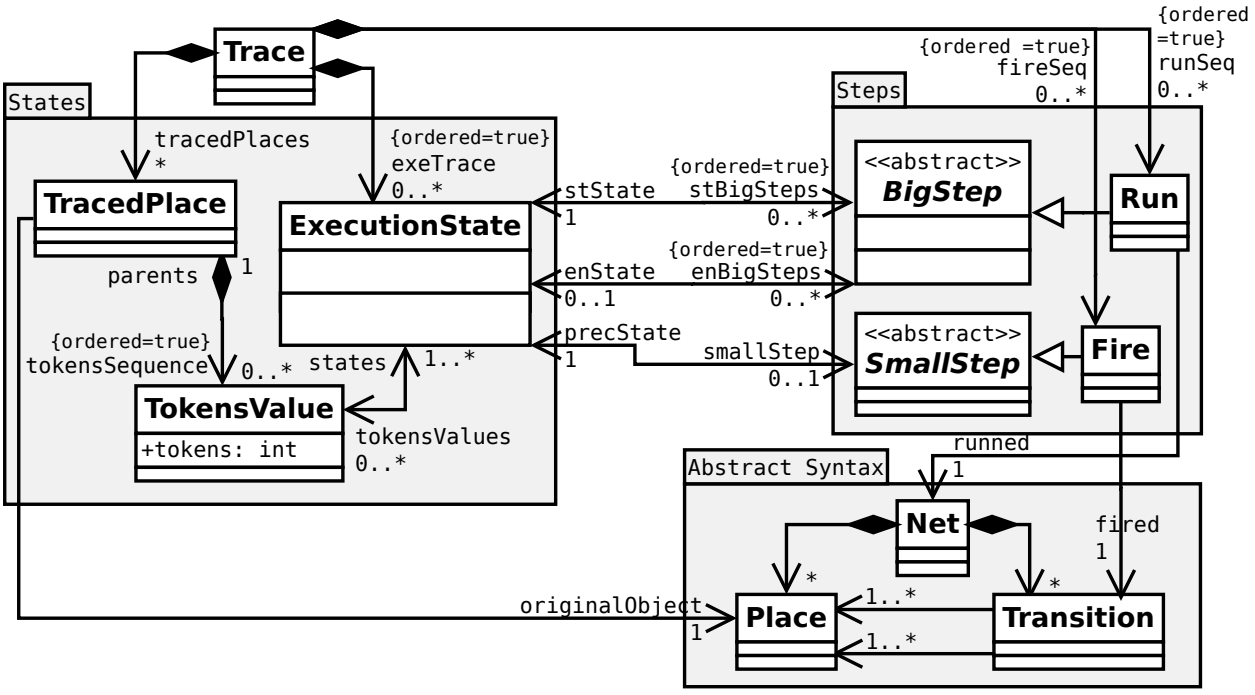


Figure 5.6: Petri net rich domain-specific trace metamodel

Figure 5.6 presents an example of a rich domain-specific trace metamodel generated to capture state information for Petri nets using our approach. In the trace metamodel, the state of the model is captured in the `ExecutionState` class, which is composed of a tuple of `TokensValue` objects, with each representing the value of a `tokens` field at a given point during execution. Each `TracedPlace` object captures all the values of the mutable fields of a specific `Place` object of the executed model. On the right, the steps are `Run` and `Fire`, which inherit `BigStep` and `SmallStep` respectively.

5.3.4 Trace Constructor

To provide omniscient debugging, we must construct an execution trace during the execution of the model. We have defined the following set of operations to be provided by the trace constructor (e in Figure 5.4):

- ***initialize***: create the base elements of the trace.
- ***addState***: add a new state in the trace if a mutable field of the model changed, or if instances of classes introduced in the execution metamodel are created/deleted.
- ***addSmallStep:*** add a *small step* in the trace.
- ***bigStepStarted***: notify that a *big step* has started.
- ***bigStepEnded***: notify that a *big step* has ended.

As explained in Section 2.1.4, the execution of a model consists of the application of a sequence of transformation rules. To capture an execution state that matches a model conforming to the execution metamodel, the operation `addState` must be called just before or after the transformation rule.

Since a *big step* is simply a sequence of *small steps*, we only need to capture states before and after *small steps*. However, we also need to capture when steps occur, hence `addSmallStep` must be called at each transformation rule that matches a *small step*, while `bigStepStarted` and `bigStepEnded` must be called before and after a rule matching a *big step*, respectively. In summary, all the calls required to construct the trace are as follows:

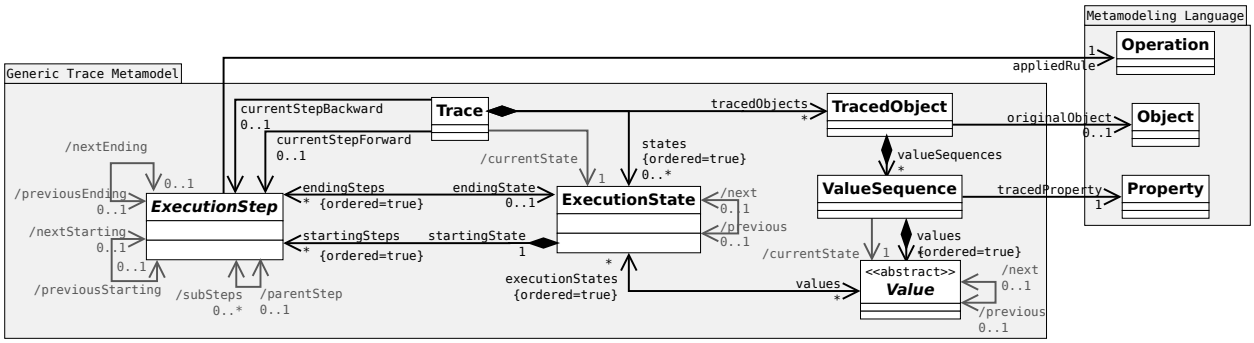


Figure 5.7: Generic Trace Metamodel Interface

- Just before the first *small step*: *initialization*
- Just before a *small step*: *addState*, *addSmallStep*
- After the last *small step*: *addState*
- Just before a *big step*: *bigStepStarted*
- Just after a *big step*: *bigStepEnded*

5.3.5 Generic Trace Metamodel

Our approach relies on the generation of a domain-specific trace metamodel for the considered xDSML. Because the debugger is generic, an interface must also be defined to manipulate traces in a generic way despite their various possible data structures. We defined this structural interface as a generic trace metamodel (h in Figure 5.4) specifying all the information that should be accessible within a domain-specific trace. Thus, it has a similar structure to generated domain-specific trace metamodels, except it contains less classes and properties.

Figure 5.7 shows the generic trace metamodel interface. To summarize, we have the same base classes (`Trace` and `ExecutionState`) as generated domain-specific trace metamodels (*e.g.,* the Petri net trace metamodel shown in Figure 5.6), and classes to represent both steps

(ExecutionStep) and values (TracedObject, ValueSequence, Value). Primitive types that extend the Value class (*e.g.,* IntegerValue) are not shown due to space limitations. We use references to elements of the execution metamodel, operational semantics, and executed model: appliedRule to specify which rule was applied, originalObject to specify which object of the original model is traced by a TracedObject, and tracedProperty to specify the property traced by a ValueSequence. Also note that derived properties are defined to facilitate the navigation among the trace, such as nextState. Finally, ExecutionStep objects are ordered either by starting time, or by ending time, hence the derived properties nextStarting and previousStarting for the starting time, then nextEnding and previousEnding for the ending time.

In order to go back and forth through the execution states and steps, a Trace has a reference currentStepForward to the ExecutionStep object that represents the next forward execution step, and a similar reference currentStepBackward for the next backward step (*e.g.,* to backOver the last step handled by the debugger). The current state is accessible with currentState, which is derived from currentStepForward. Similarly, the property currentValue of ValueSequence is indirectly derived from currentState.

To provide this interface, our solution relies on a generated one-way model transformation from the domain-specific trace metamodel to the generic trace metamodel. Thereby, we have a generic read-access to the trace. Regarding write-accesses, we store the debugging state (*e.g.,* currentState) in a separate generic structure, hence avoiding the need to modify the domain-specific trace.

5.3.6 State Manager

An omniscient debugger must be able to revisit a previous state by reverting the executed model into the state stored in the execution trace. The operation enabling a debugger to return to a past state is provided by the state manager (g in Figure 5.4), which we specified with a single service, ***restoreModelToState***, which restores the executed model into a given execution state.

The idea is similar to the well-studied *memento* design pattern [95], albeit at the model level. The originator is the model being executed; the memento is an execution state of the trace; and the caretaker is both the trace and the trace manager.

5.3.7 Domain-Specific Trace Manager

To implement both the trace constructor and the state manager and to generically expose as much information as stated in the generic trace metamodel, our approach relies on the generation of a domain-specific trace manager (c in Figure 5.4). The reason for generating this component is efficiency (Objective 1), because trace manipulations can be tuned for both the considered xDSML and the generated domain-specific trace metamodel (introduced in Section 5.3.3).

Consequently, the domain-specific trace manager and trace metamodel generation are coupled. Because all generated operations manipulate a trace conforming to this metamodel, a set of traceability links obtained from the generation of the domain-specific trace metamodel is provided to the generator. From there, the main steps of the generation are as follows:

1. The systematic base structure of the generated trace metamodels is known from the domain-specific trace metamodel generator. Thus, `initialize` can be generated;

2. The mutable fields of the execution metamodel and the corresponding classes in the trace metamodel are known. Thus, `addState` can be generated. An implementation of this ser-

vice includes looking for changes among mutable fields then creating a state and new values if any change is detected. Likewise, `revertModelToState` can be generated, which relies on links from the trace to the model to restore values and re-create objects.

3. The operational semantics (defined in Section 2.1.4) and the corresponding step classes in the trace metamodel are known. Thus, step creation can be generated. While `addSmallStep` is straightforward, `bigStepStarted` requires stacking big steps that are in progress, and to unstack them in `bigStepEnded`.

4. Finally, the systematic shape of generated trace metamodels is known. Thus, a generic trace metamodel interface can be provided, as defined in Section 5.3.5.

5.3.8 Generic Multidimensional Omniscient Debugger

The last component to define is the generic multidimensional omniscient debugger (f in Figure 5.4) that relies on the execution engine to control the current execution, on the state manager to restore previous states, and on the generic trace metamodel interface to manipulate traces.

Figures 5.8, 5.9, and 5.10 provides a precise definition of each service required for multidimensional omniscient debugging using the services of the three aforementioned required components. These components are represented by three singletons: `engine` represents the execution engine, `trace` represents the root element of a model conforming to the generic trace metamodel, and `manager` represents the state manager. In the following paragraphs, we explain the definitions of all the services provided by the debugger defined in Figures 5.8, 5.9, and 5.10.

5.3.8.1 Jump services

Figure 5.8 starts with the definition of the most important omniscient debugging service, `jump`. Jumping consists of going back to a chosen state in the execution trace, and is accomplished

Omniscient Debugging Service	Definition

jumpToState(*state* : ExecutionState)	**if** *state* ≠ *trace*.currentState **then**
	  manager.restoreModelToState(*state*);
	if *state*.startingSteps ≠ ∅ **then**
	  trace.currentForwardStep ← *state*.startingSteps.first();
	else
	  trace.currentForwardStep ← *null*;
	if *state*.endingSteps ≠ ∅ **then**
	  trace.currentBackwardStep ← *state*.endingSteps.last();
	else
	  trace.currentBackwardStep ← *null*;

jumpBeforeStep(*step* : ExecutionStep)	jumpToState(*step*.startingState); *trace*.currentForwardStep ← *step*;

jumpAfterStep(*step* : ExecutionStep)	**if** *step*.endingState ≠ *null* **then**
	  jumpToState(*step*.endingState);
	  trace.currentBackwardStep ← *step*;

backInto()	jumpAfterStep(*trace*.currentBackwardStep.previousEnding)
backOver()	jumpBeforeStep(*trace*.currentBackwardStep)

backOut()	**if** *trace*.currentBackwardStep.parent ≠ *null* **then**
	  trace.currentBackwardStep ← *trace*.currentBackwardStep.parent;
	  backOver();

playBackwards()	**while** *trace*.currentBackwardStep.previousEnding ≠ *null* ∧ ¬*engine*.isPaused() **do** backInto()

Figure 5.8: Definition of the Omniscient Debugging Services

Standard Debugging Service	Definition
toggleBreakpoint(*p* : Predicate)	*engine*.pauseWhen(*p*)

stepInto()	**if** *trace*.currentForwardStep.nextStarting = *null* **then**
	  steppedInto ← *trace*.currentForwardStep;
	  engine.pauseWhen(*steppedInto*.nextStarting ≠ *null*);
	  engine.resume();
	else
	  jumpBeforeStep(*trace*.currentForwardStep.nextStarting);

stepOver()	**if** *trace*.currentForwardStep.endingState = *null* **then**
	  steppedOver ← *trace*.currentForwardStep;
	  engine.pauseWhen(*steppedOver*.endingState ≠ *null*;
	  engine.resume();
	else
	  jumpAfterStep(*trace*.currentForwardStep);

stepOut()	**if** *trace*.currentForwardStep.parent ≠ *null* **then**
	  trace.currentForwardStep ← *trace*.currentForwardStep.parent;
	  stepOver();

play()	**while** *trace*.currentForwardStep.nextStarting ≠ *null* ∧ ¬*engine*.isPaused() **do** stepInto() *engine*.resume()

Figure 5.9: Definition of the Standard Debugging Services

Multidim. Omniscient Debugging Service	Definition
jumpToValue(*v* : Value)	jumpToState(*v*.executionStates.first())

stepValue(*valueSeq* : ValueSequence)	**if** *valueSeq*.current.nextValue = *null* **then**
	  previousValue ← *valueSeq*.current;
	  engine.pauseWhen(*previousValue*.nextValue ≠ *null*;
	  engine.resume();
	else
	  jumpToValue(*valueSeq*.current.nextValue);

backValue(*valueSeq* : ValueSequence)	jumpToValue(*valueSeq*.current.previousValue)

Figure 5.10: Definition of the Multidimensional Omniscient Debugging Services

via the `jumpToState` service. First, it uses the `restoreModelToState` service from the state manager to modify the model, then updates the debugger state represented by `currentForwardStep` and `currentBackwardStep`. Additionally, we need to be able to jump back either right before or after an execution step, which is provided by the services `jumpBeforeStep` and `jumpAfterStep`.

5.3.8.2 Other omniscient debugging services

Next, we define the remaining omniscient debugging services. `backInto`, `backOver` and `backOut` directly rely on `jumps` to reach the correct state. The last service, `playBackwards`, is a loop backwards until either the initial state is reached or the engine is paused.

5.3.8.3 Standard debugging services

Figure 5.9 defines the standard debugging services; *i.e.,* breakpoints and forward stepping. `toggleBreakpoint` provides a generic way to define a breakpoint through a predicate, which can be defined on the model state (*e.g.,* watching for a specific instruction to be reached) or on the trace (*e.g.,* verifying a temporal property or watching for a specific step to be applied). It is defined using the `pauseWhen` service that must be provided by the execution engine. The next services are the standard step operations: `stepInto`, `stepOver`, and `stepOut`. There are two cases to consider: (1) When the current step is at the end of the trace, we rely on `pauseWhen` and `resume` to apply the operational semantics up until the correct situation is reached (*e.g.,* waiting

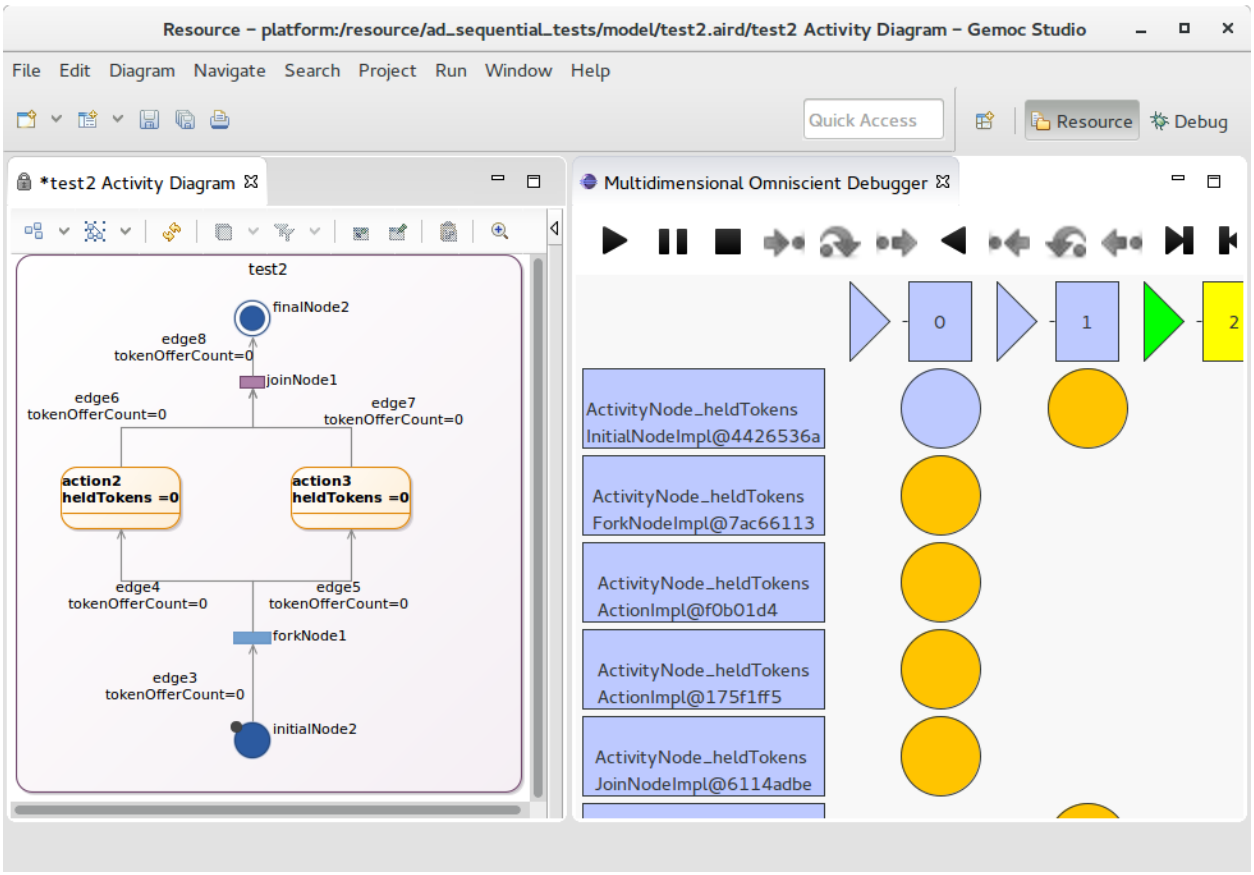


Figure 5.11: GEMOC Studio with the multidimensional omniscient debugger prototype running an fUML activity.

for the current big step to be finished with `stepOver`). (2) When the execution state is at a past state (*e.g.,* after a jump backwards), `jump` services are called (even though these step services are not specific to omniscient debugging) while the engine remains paused.

5.3.8.4 Multidimensional omniscient debugging services

Figure 5.10 define the final set of services providing multidimensional omniscient debugging facilities. The goal of these services is to provide the capacity to debug a model by following the sequences of values of specific mutable fields, thereby improving the usability of omniscient debugging for xDSMLs (Objective 2). Implementing these services is simplified by the structure of the trace metamodel providing access to each of the value sequences. Thus, `jumpToValue` is

a use of `jumpToState`; and `backValue` directly uses `jumpToValue`; while `stepValue` is very similar to `stepOver`.

5.4 Tooling for Omniscient Debugging

This section presents the language and modeling workbench called GEMOC Studio and explains how we applied a subset of our approach (*i.e.,* the generative part and a debugger with basic operations) to offer a proof-of-concept prototype multidimensional omniscient debugger.

5.4.1 The GEMOC Studio

The GEMOC Studio is an Eclipse package atop the Eclipse Modeling Framework (EMF) including both a language workbench to design and implement tool-supported xDSMLs, as well as a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. The modeling workbench includes an advanced generic execution engine that can be used to execute any model conforming to an xDSML defined within the language workbench. An API is available to extend the engine with addons through the use of an observer pattern [95]: the engine sends notifications to all its addons to inform them about the execution progress (*e.g.,* a step is starting/ending). Lastly, it supports pausing and resuming the execution between steps.

Our prototype focused on the operational semantics implementation language Kermeta [51] that relies on aspects to modularly implement the operational semantics and weave them into the provided metamodel.

5.4.2 Omniscient Debugging in the GEMOC Studio

The prototype demonstrating our approach to support multidimensional omniscient debugging is implemented in GEMOC Studio. The prototype requires the creation and integration of the components described in the following subsections.

5.4.2.1 Trace addon generator

The generative part of our approach takes the form of a trace addon generator, that receives as input an xDSML composed of an Ecore abstract syntax and Kermeta aspects. Note that Kermeta aspects function both as an execution metamodel and operational semantics, because aspect weaving allows the definition of new mutable properties in classes and operations used as transformation rules. The generator produces a GEMOC engine addon with a domain-specific trace metamodel and a domain-specific trace manager. Using the addon mechanism, the manager reacts to the execution steps of the engine to construct the trace accordingly. It also provides an interface to revert the state of the model (*i.e.,* a state manager), and an interface to query the trace (*i.e.,* a generic trace metamodel interface).

5.4.2.2 Generic debugger logic

The generic part of our approach includes multidimensional omniscient debugging services within the GEMOC studio. Our prototype provides `toggleBreakpoint` with only one kind of predicate (*i.e.,* a model element is targeted by a step), `stepInto`, `jumpToState`, `jumpToValue`, and visualization (see next paragraph).

5.4.2.3 Graphical interface

The graphical user interface of the debugger includes a prototypical graphical widget that shows both the execution trace and the value sequences of all mutable fields. Figure 5.11 shows the GEMOC Studio with a model animator on the left and the omniscient debugging widget on the right. Double clicking on a model element triggers a `toggleBreakpoint` that pauses the execution when this element is targeted by a step. In the right widget showing the trace, the first row of numbered squares represents all the execution states. Each subsequent row represents the value sequence of a specific mutable field. The yellow rectangle indicates the current execution state and

orange circles indicate current values of the mutable fields. Double-clicking on a state (numbered squares) triggers a `jumpToState` to the corresponding execution state. Similarly, double-clicking on one of the values (circles) triggers a `jumpToValue` to the corresponding value. Because the GEMOC Studio provides animation of the executed model, the left view showing the model is updated at each action.

5.5 Evaluating Efficiency and Scalability of a Generative Approach to Omniscient Debugging for xDSMLs

In this section, we first present the design and results of an empirical study providing an initial evaluation of the efficiency of our approach. Then, we discuss the benefits of multidimensional omniscient debugging.

5.5.1 Research Questions and Experimental Setting

To evaluate the efficiency of our approach (Objective 1), we considered the following research questions:

RQ 1: Is our approach more efficient in memory as compared to a clone-based omniscient debugger?

RQ 2: Is our approach more efficient in time for omniscient debugging services as compared to a weak omniscient debugger and to a clone-based omniscient debugger?

The evaluation of efficiency presented in this section is the comparison of three omniscient debuggers as presented in Section 5.2.1 and in Figure 5.1. First, *WeakDebugger* is a weak generic omniscient debugger. Such a debugger is expected to be efficient in memory, because there is no trace to store; and inefficient in time, because the execution engine must be restarted at each jump backward. Second, *CloneBasedDebugger* is a clone-based generic omniscient debugger that

constructs a generic trace using *deep cloning* (*i.e.,* the complete model is copied at each step) and implements jumps using the model differencing library EMF Compare[2]. Because this debugger relies on an execution trace, it is expected to be less efficient in memory and more efficient in time than *WeakDebugger*. Finally, *MultiDimDebugger* is the prototype multidimensional omniscient debugger applying our approach. All three debuggers were implemented in the GEMOC Studio.

We applied our approach to a subset of a real-world xDSML, namely fUML [98]. The considered subset contains the Activity Diagram portion of the language. In summary, a model conforming to this xDSML is made of an activity, which consists of control nodes and action nodes. Nodes are linked by control flow links, starting with an initial node and ending with a final node. Similarly to a Petri Net, tokens are passed along nodes to drive the execution. In addition, variables can be defined in activities and modified with actions. The xDSML was implemented with GEMOC Studio using Ecore for the abstract syntax and Kermeta for both the execution metamodel and the operational semantics.

As in our previous work [92], we used models[3] taken from the case study of Maoz *et al.* [99]. This choice was made to help establish a benchmark, facilitate comparison with future work, and because the models were drawn from industrial sources. The dataset we obtained contains 40 models whose sizes range from 36 to 51 objects. We plan to integrate larger models to the dataset for a future study, but are confident in the current ones to provide initial meaningful comparison.

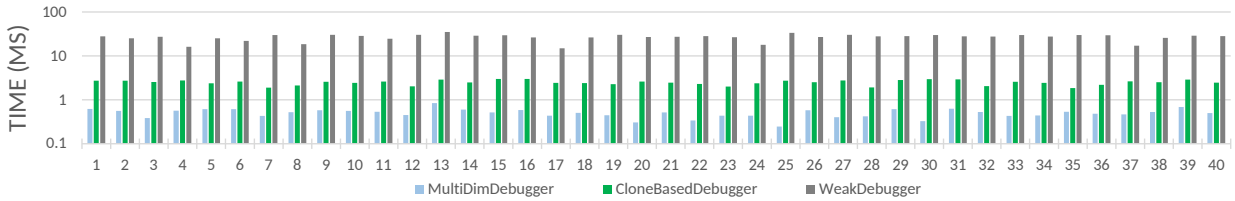


Figure 5.12: Time required to perform a *jumpToState*

5.5.2 Data Collection and Analysis

To compare efficiency in memory, instead of observing the memory usage of the complete environment (*e.g.,* execution engine and loaded model), we measured the memory used only by the debugger. More precisely, for each of the considered models, we collected the amount of memory required to store the execution trace at the end of its execution by making precise memory measurements using heap dumps and Eclipse MAT[4].

To compare efficiency in time, we focused on the main operation used by all omniscient debugging services: `jumpToState`. More precisely, for each of the considered models, we measured the average amount of time required to perform a `jumpToState` by jumping to each previously visited state once and in a random order. Measures were done using Java's operation `System.nanoTime`.

Data was collected in a reproducible way through a programmatic use of GEMOC Studio's engine (see the companion webpage). Each result is an average value computed from five identical measurements made using an Intel i7-3720QM CPU with 8GB of RAM.

5.5.3 Results and Discussion

Here, we provide a summary and discussion of the results we obtained from our evaluation.

[2] `https://www.eclipse.org/emf/compare/`
[3] Models available at `http://www.se-rwth.de/materials/semdiff/`
[4] `https://www.eclipse.org/mat/`

5.5.3.1 RQ 1: Efficiency in memory

Figure 5.13 shows the results obtained regarding the memory required to store an execution trace. The *x*-axis shows the number of elements in the trace, while the *y*-axis shows the amount of memory used in kB. First, *WeakDebugger* does not use memory, because it does not store a trace. Second, we observe that our approach is always more efficient in terms of memory usage than the *CloneBasedDebugger* with 3.0 times improvement on average. We hypothesize this is due to the domain-specific traces obtained with our approach that are designed to only contain the evolution of the mutable fields of the model with minimal redundancy, whereas cloning implies significant redundancy. In addition, we note that our approach has a gentler slope than *CloneBasedDebugger*, which suggests better scalability with large traces. To summarize and answer RQ 1, we observe that our approach is more efficient in memory than a clone-based approach.

5.5.3.2 RQ 2: Efficiency in time

Figure 5.12 presents the results obtained regarding the average amount of time required to perform a *jumpToState*. The *x*-axis shows the identifier of the executed model, while the *y*-axis shows the amount of time in milliseconds. First, we observe that trace-based debuggers are always better than *WeakDebugger* (right), in particular *MultiDimDebugger* (left) being 54.1 times faster than *WeakDebugger*. This is explained by the time required to reset the execution engine. Second, we observe that *MultiDimDebugger* is more efficient than *CloneBasedDebugger* (center) with 5.03 times improvement on average. We hypothesize this is due to the generated trace manager, which contains code specific and tuned to both the xDSML and the generated domain-specific trace meta-model. To summarize and answer RQ 2, we observe our approach is more efficient in time than the traceless approach and clone-based approach.

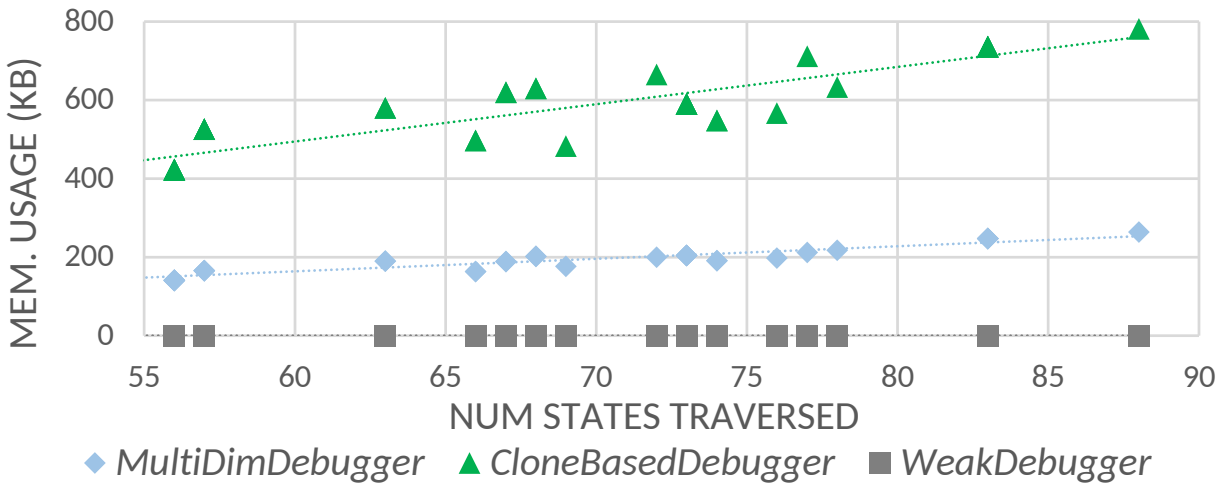


Figure 5.13: Memory used by the execution trace

5.5.3.3 *Benefits of Multidimensional Facilities*

To ensure the usability of omniscient debugging (Objective 2), our approach provides multidimensional omniscient debugging; *i.e.,* facilities to navigate among values of mutable fields of an executed model. In essence, we believe that providing explicit visualization of the dimensions of a trace (see Figure 5.11) and means to traverse such trace according to specific dimensions (*e.g.,* `stepValue`), has a significant positive impact on usability (Objective 2). To completely validate Objective 2 requires user experiments to empirically assess the expected benefits of multidimensional facilities. We defer this task to future work.

5.6 Related Work

This section overviews relevant existing work on debugging and trace visualization in MDE and relates these works to the contributions presented in this chapter.

5.6.1 Omniscient Debugging in MDE

Maoz and Harel [100] and Hegedüs *et al.* [101] present trace exploration tools that contain similar facilities to an omniscient debugger. However, these techniques are defined for post-

mortem analysis rather than use during live sessions, whereas our technique supports live debugging sessions. Additionally, Maoz and Harel do not support domain-specific execution traces for xDSMLs.

Chapter 4 explored applying omniscient debugging to model transformations within the context of AToMPM. The focus of that debugger was the two basic transformation languages provided by AToMPM. Also, Van Mierlo recently presented a proposal toward the debugging of executable models defining simulation semantics [67]. A particular focus of the work addressed handling simulated real-time. The work presented in this chapter was concerned with xDSMLs.

5.6.2 Trace Visualization and Debugging in MDE

Existing work on trace visualization, such as MetaViz by Aboussoror *et al.* [102], or the work of Maoz and Harel [100], would be strongly complimentary with our approach. Indeed, while we focused on the back-end concern of omniscient debugging, trace visualization is required for the front-end.

More recently, the work of Chis *et al.* on a Moldable Debugger [103] can be compared to the work presented in this chapter. The technique described in this chapter provides generic debugging operations supported by domain-specific trace management facilities, but Chis *et al.* provide a framework to define domain-specific debugging operations and user interfaces. Also, the approach described here is completely automatic given a well-formed xDSML, whereas manual work is required to extend the Moldable Debugger to support an xDSML. Yet, both approaches tackle different and independent challenges, and provide complementary results.

5.6.3 Domain-Specific Execution Traces in MDE

Meyers *et al.* introduced the ProMoBox framework [104], which generates a set of metamodels from an annotated xDSML, including a domain-specific trace metamodel. Among others,

a difference with our work is that they consider an annotated abstract syntax whose properties are annotated either as runtime or event, while we consider the abstract syntax and the execution metamodel distinct to improve separation of concerns. In addition, they do not provide alternative ways to explore a trace, while we provide various navigation paths for multidimensional debugging.

Similarly, Gogolla *et al.* [105] generate filmstrip models from UML class diagrams. Such filmstrip models match what we call domain-specific trace metamodels, and provide some navigation paths among object states. However, film strip models do not address redundancy. Object states are always recreated at each model change, and the states store values of both mutable and immutable fields.

5.7 Conclusion

Omniscient debugging is a promising dynamic V&V approach for xDSMLS that enables free traversal of the execution of a system. While most GPLs already have efficient debuggers, bringing omnicient debugging to any xDSML is a tedious and error-prone task. A solution is to define a purely generic debugger, but this requires managing both *efficiency* and *usability* issues that emerge. The approach we presented relies on generated domain-specific trace management facilities for improved efficiency and provides multidimensional omniscient debugging facilities for improved usability. The debugger relies on an execution engine to control the execution and a generated domain-specific trace manager to provide omniscient services. The states reached during an execution are stored in a trace conforming to a generated domain-specific trace metamodel. Outcomes of this project included a prototype within GEMOC Studio, a language and modeling workbench, and an evaluation performed using the fUML language. An improvement was observed regarding both the memory consumption and the time to perform a *jump*, when compared to two generic omniscient debugger variants.

CHAPTER 6

HOW DEVELOPERS DEBUG

Maintenance tasks make up a significant portion of the effort and cost of software projects. To address this concern, techniques and tools have been developed to provide support during maintenance and comprehension tasks. Among such tools, several have emerged that enable developers to query the system in both static and dynamic contexts. This chapter presents the design and results of an exploratory study concerning the use of informal queries during a debugging task. The study was designed to identify how developers utilize queries during a maintenance task in OO programming. The empirical human-based study investigated how developers acquire knowledge about a software system, specifically with regards to forming and resolving queries. The results of the study provide insight and new understanding about how these queries impact both task completion and source navigation during a debugging session. The insights gained from this study are intended to inform future research efforts focused on developing improved query-based debuggers and general debugging tool support by identifying the importance of queries and categorizing the types of queries developers seek to answer during debugging tasks.

6.1 Introduction

Maintenance tasks are a significant software development concern with estimates of around 20% of this effort focused on defect correction [106]. To address this concern, techniques and tools have been developed to provide support during maintenance and comprehension tasks. Among

such tools, several have emerged that enable developers to query the system in both static and dynamic contexts [12, 14, 17, 56, 59, 61, 72].

QBD provides developers with query languages that facilitate asking questions about a program (typically either structure or runtime behavior) [14]. QBD tools must provide some range of available query types. This leads to the question of what query types are most important to support? Previous researchers have approached this question. Sillito *et al.* present a set of 44 generic questions drawn from an exploratory study [107]. The 44 questions are categorized based on when the question was posed during the mental model building process. LaToza *et al.* presented a detailed analysis of a single type of question that they termed reachability questions [108]. A reachability question was defined as a question searching across all possible paths of execution. Hibberd, Lawley, and Raymond [53] introduced three categories of queries that developers may ask during the debugging process for model transformations: queries concerned with logical bugs, queries concerned with well-formedness bugs, and queries concerned with analyzing the system to identify bugs. Collectively, these works have categorized query types [53, 107] and discussed in depth a single query type [108].

The work presented in this chapter focuses on categoring query types according to the type of information the query is seeking. The novel query categorization presented in this chapter is complimentary to existing work in further examining and categorizing the types of queries developers utilize during a debugging task. Additionally, because the types are defined according to the kind of information the observed queries are seeking, they can be mapped to existing and future tool support and techniques to identify any gaps or to evaluate the impact of the tooling. In addition to identifying and categorizing query types, the impact of queries and the use of existing tools with a standard IDE to resolve queries was investigated.

A grounded theory approach was used to identify the types of queries formed by developers. Grounded theory [109] is a research methodology that relies on deriving results from analyzing and coding observed results. The results are coded, and as more results are compiled, categorizations can be derived. These categorizations are based on empirical data and can serve to found new theories or support existing theories.

The study presented in this chapter was performed in the context of OO development using the Java language within the Eclipse IDE. Participants were asked to debug two defects in the APACHE ANT system. Despite the larger scope of this dissertation work focusing on an MDE context, this study was performed in a GPL context. This decision was made based on several factors. First, the more common nature of GPL development provides for a larger participant pool that has a more consistent educational background in the specific development paradigm of GPLs when compared to MDE. This larger pool of participants eases recruitment, and the more consistent background of participants ensures more reliable results. Second, the available development environments for GPLs are overall more mature than those available in MDE with researchers commonly commenting on tools as a factor in the lack of MDE adoption in industry [110]. However, projects such as Epsilon[1], EMF[2], AToMPM[3], the GEMOC Studio[4], and GME[5] are all making strides toward a more mature and industry ready stable of development environments for MDE. Finally, an initial study in the more common development paradigm of GPLs serves as a foundation on which to expand to an MDE context (as discussed in Section 7.4). By first investigating within a GPL context and then replicating to an MDE context, the results of the later study may be linked to

[1] http://projects.eclipse.org/projects/modeling.epsilon
[2] https://eclipse.org/modeling/emf/
[3] http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm
[4] http://gemoc.org/studio/
[5] https://webgme.org/

other related work in a GPL context through the more direct comparison to the study presented in this chapter. Thus, an initial study in the context of GPL development improves quality of analysis possible and the contribution of a later study in the context of MDE.

The following research questions guided the design of the study.

RQ 1: Do developers form queries during debugging tasks?

RQ 2: What types of queries are formed by developers during debugging tasks?

RQ 3: What leads developers to generate new queries and how do they relate to previous queries?

RQ 4: Which (if any) observed aspects of queries correlate with successful debugging task completion?

To address these research questions, an exploratory empirical study investigating queries formed by developers without formal query-based debugging support was performed with developers using Eclipse to debug several defects in an industrial quality system, ANT. The remainder of the chapter is structured as follows. Section 6.2 overviews QBD and discusses previous studies exploring query types and the processes used by developers. Section 6.3 details the study design. Then, Section 6.4 overviews the results of the empirical study and discusses implications and outcomes of the results. Finally, Section 6.5 provides concluding remarks.

6.2 Background and Related Work

QBD implementations provide query languages that enable developers to ask questions about a program. The developer-provided queries analyze the program for portions of the code that are relevant to the query [14]. The query-based approach provides a semi-automated solution for bug localization. Each query-based debugger provides a formal language that developers use

to formulate queries. These languages have varying syntaxes ranging from machine-like to natural language.

6.2.1 QBD Systems

One noteworthy query-based debugger is Whyline [12]. Whyline focuses on providing a guided debugging session by offering suggested queries to the user. The queries presented by Whyline focus on "why did" and "why didn't" questions such as, "Why did property x of object y have this value?" and "Why did property x of object y not get set to this value?" The query language used by Whyline focuses on presenting questions in a natural language, but does not enable the user to provide custom queries.

Other query-based debuggers offer more machine-like languages. Lencevicius presented query-based debuggers for Java and Self that utilize the expressions of the specific language for the query language [14]. Potanin *et al.* introduced FQL (Fox Query Language), which provides a set of predefined functions that allow the developer to filter and summarize the information contained in the object graph, as well as control functions that allow actions such as loading and saving object graphs and query results [59]. Coca enables developers to query events during execution using a query language based on Prolog [56]. PQL (Program Query Language) enables developers to produce queries by specifying minimal code snippets using the target language [61]. Hobatr and Malloy discuss using OCL queries to debug C++ systems [72]. Schoenboeck *et al.* also utilize OCL queries in their debugger, TROPIC, which enables debugging QVT systems that are converted to a variant of Petri nets that the authors refer to as Translation Nets [17].

Each of these proposed techniques derive their query language from a programming language. However, these works focus on providing support to developers before exploring the type and focus of queries used by developers. The study presented on this chapter provides insights

regarding the queries formed by developers during debugging tasks including what types of information developers seek, why they form queries, and the dominant types of information developers seek.

6.2.2 Query Types

As research has emerged exploring query-based debugging techniques, an important component to the discussion of query-based debugging has investigated the type of queries that developers use during the debugging process.

Sillito *et al.* present a set of 44 generic questions drawn from an exploratory study [107]. The 44 questions are categorized based on when the question was posed during the mental model building process. LaToza *et al.* presented a detailed analysis of a single type of question that they termed reachability questions [108]. A reachability question was defined as a question searching across all possible paths of execution.

Hibberd, Lawley, and Raymond [53] introduced three categories of questions that developers may ask during the debugging process for model transformations. The first two categories focus on understanding behavior (typically as it relates to producing the observed output), and the third category focuses on gathering information not directly related to the behavioral concerns covered by the previous categories. These categories were identified with a specific focus on model transformation concerns. Furthermore, the categories were not identified from observed query usage, but rather from analyzing the context of MTs and MDE. As such, the work presented in this chapter is not directly related to these queries, but a separate empirical study exploring the formation and use of queries would be warranted to explore fully categorizing queries in the context of an MT system. However, it is worth noting that the query types covered focus primarily on behavior of the system (two of three categories), and the final category covers a very broad spectrum of

questions. The results presented in this chapter will be built up from the set of observed queries. Thus, the resulting query types will each cover specific focuses, and the count of observed queries will be presented for each type. The count of queries for each type provides insight into the focus of developers (*e.g.,* do developers focus on queries related to system behavior?).

6.2.3 Exploring the Debugging Process

Several existing studies have explored developers' behavior during maintenance tasks [111, 112]. However, these studies have focused on how developers navigate code during a maintenance task. Ko *et al.* had developers debug a Java system using Eclipse, and the developers were observed during the tasks. They were focused on studying how developers seek information during debugging tasks [111]. Lawrance *et al.* performed a similar study, which compared the process used by subjects during the debugging task with a conceptual model known as Information Foraging [112]. The information foraging model would posit that a developer (referred to as the 'predator') follow a series of cues (referred to as 'scents') in search of the defect (referred to as 'prey'). The model is inspired by the process used by predators to hunt prey in the natural world. Unlike the study detailed in this chapter, the studies by Ko *et al.* and Lawrance *et al.* investigated how developers navigate code without considering queries. However, the design of the study presented in this chapter is modeled after the design used in both of these previous studies with the focus shifted to exploring how developers form and use queries. The work presented in this chapter identifies the queries developers form and categorizes them according to the type of information the developer is seeking. Additionally, the factors that lead a developer to form a query and how the query is resolved are collected. This information provides an improved understanding of the process used by developers as well as key insights into how available tool support can address the needs of developers during debugging tasks.

6.3 Exploratory Study of Query Formation, Use, and Impact

This section briefly introduces the design of the study, overviews the demographics of the participants, describes the data collection and analysis processes, and discusses the potential threats to validity for the study.

6.3.1 Study Procedure and Setting

Subjects were asked to assume the role of a new developer for Apache Ant and work to resolve two distinct bugs (IDs 38175 and 38082). The goal was to identify all relevant portions of the source code and perform any necessary modifications to the system. The two bug reports provided to the subjects were retrieved from the Apache Bugzilla repository[6].

Subjects were given instructions for completing the debugging tasks which included bug reports and discussion of the available resources to be used during the tasks. The subjects were provided with Eclipse v4.2 and a workspace including the source code for Apache ANT and two runtime configurations designed to test the bugs. Apache Ant (v1.6.5, rev 367135) was selected as the subject system based on size (tens of thousands of lines of code) and quality of the system (*i.e.,* the system adheres to common practices regarding Java coding style and structure, and the system is well-tested). Additionally, subjects were provided with the ANT documentation and Java SE7 javadocs.

A screen capture application was run during the experiment. Subjects were required to use the screen capture software monitoring their interactions with the system and to capture their vocalizations as part of the think-aloud method. Subjects were provided with instruction in the think-aloud method [113] that was required during the study. The think-aloud method requires subjects to explain verbally their internal processes that are not otherwise observable. Thus, de-

[6] https://issues.apache.org/bugzilla/

velopers would provide continuous discussion of their actions and why they were taking these actions. Through analyzing these vocalizations, we were able to much more accurately identify the queries being formed, if and how the query was resolved, and why the query was formed. Upon completing the task or reaching one hour of elapsed time, the subjects were asked to complete a post-survey for each task to rank perceived difficulty, frustration, effort, as well as briefly describe the process used to complete or attempt the task. Additionally, participants were asked to complete a brief demographic survey collecting their experience with the language and environment used during the study.

6.3.2 Bug Reports

This section provides the two bug reports, as presented to participants, including information about how to reproduce the bugs.

6.3.2.1 Bug 1: `The failonerror="no" Option Does not Work for Locked Files`

When we try to perform a recursive copy in a directory that contains a locked file, the copy fails before the end of the whole copy, even if I have the attribute failonerror set to "no."

How to reproduce Open Mozilla Thunderbird, then in Eclipse, run using the "bug1" configuration. The "bug1" configuration will invoke Ant with testBuild/bug1/build.xml. If Mozilla Thunderbird is open, there is a locked file in the profile directory, parent.lock. The locked file will fail to copy. As a result of the failed copy, you will receive a warning message stating parent.lock could not be copied. This single failure should result in the source directory having exactly one more file than the destination directory. If you right click on a directory in windows explorer and select properties, you will be able to view the number of files in the directory. The defect will cause the

number of files in the source directory to exceed the number of files in the destination directory by more than one.

How to verify If the process is working correctly, the destination directory should have exactly one less file than the source directory.

6.3.2.2 *Bug 2:* `SCP Task Password with Special Characters`

The SCP task does not handle passwords with special characters such as "@".

How to reproduce In Eclipse, run using the "bug2" configuration. The "bug2" configuration will invoke Ant with the build.xml in the testBuild/bug2/ directory. You should receive output similar to that presented in Figure 6.1.

```
Buildfile: .\testBuild\bug2\build.xml
default:
[scp] Connecting to t@sgtpepper.cs.ua.edu:22
BUILD FAILED
C:\Users\Student\Desktop\HowDevelopersDebug\workspace\apache-ant\testBuild\bug2\build.xml:4:
com.jcraft.jsch.JSchException: java.net.UnknownHostException: t@sgtpepper.cs.ua.edu
Total time: 0 seconds
```

Figure 6.1: Sample Defect Output for Bug 2

How to verify When the bug has been fixed, the SCP task will connect to the server and transfer the files, and you should receive output similar to that presented in Figure 6.2

6.3.3 Demographics

The study included 16 participants including both graduate students and senior level undergraduate students. Subjects had an average of 4.25 years experience working in Java including an average of 8.5 projects. Subjects also had an average of 4 years experience working in Eclipse

```
Buildfile: .\testBuild\bug2\build.xml

default:

    [scp] Connecting to sgtpepper.cs.ua.edu:22

    [scp] Receiving file: /home/zod/TEST

    [scp] Receiving: TEST : 0

    [scp] File transfer time: 0.0 Average Rate: ? B/s

    [scp] done

BUILD SUCCESSFUL

Total time: 1 second
```

Figure 6.2: Sample Correct Output for Bug 2

including an average of 6.75 projects. A few subjects had prior industry experience, but subjects primarily reported on academic experience. The subjects had all taken an SE course, and some participants were enrolled in a Verification and Validation course. Furthermore, these participants did not have prior experience with the ANT project used during the study as would be expected of a new developer in an organization. Thus, these subjects represent a typical range of experience expected of entry level developers.

6.3.4 Data Collection

The screen captures were reviewed and results were collected into a data collection form that collected a variety of aspects related to the query occurence (as illustrated in Figure 6.3). Instances of queries were recorded according to the data collection form. Queries could either be directly expressed or implied. An implied query was identified through developer actions (*e.g.,* using a print statement to determine a variable's state). The implied queries represent queries that were explicitly stated, but clearly present the developer seeking some information in pursuit of

Participant #:
Task #:
Was task successfully completed (y/n):
Query #:
Time of Query (mm:ss):

Was this query satisfied (y/n)?

Did existing tool support help resolve the query (y/n)?

If so, describe tool support used:

Did the existing tool support used resolve the query automatically (y/n)?

Was this query used to aid in resolving another query (y/n)?

If so, which query(ies)?

Query Description:

Describe Context of Query:

Describe Events Leading to Query:

Additional Notes:

Figure 6.3: Data Collection Form

completing the debugging task. To avoid misinterpreting participants or failing to identify queries, the recorded sessions were independently reviewed by at least two separate researchers during data collection. Each video was reviewed carefully and any queries noted by the reviewer were collected using a data collection form (see Figure 6.3). Thus, each reviewer formed an indepently identified set of queries for each subject. The independently collected query sets were then compared and assessed as part of a *first round review* to identify the most accurate and complete set of queries. After, the *first round review* the resulting query set was carefully reviewed in a *second round review*. The *second round review* was used to identify query types, query generation factors, and to clarify and formally code previously noted query relationships. The queries and related informa-

tion collected through the *second round review* were then used to address the research questions driving the study. This process resulted in seven hundred ten identified queries across the sixteen participants.

Results of the post and demographic surveys were collected through a Google form into a spreadsheet that enabled basic statistical analysis. The results of the post survey failed to reveal any additional insight beyond the data collected from the screen capture video recordings.

6.3.5 Data Analysis

The process used to address each research question is briefly summarized below, and the full discussion of the results and implications are provided in Section 6.4.

6.3.5.1 Do developers form queries during debugging tasks? (RQ 1)

The primary thrust of *RQ 1* is addressed through confirming that each developer formed queries. However, to fully address the intent of the question, basic descriptive statistics are included. These descriptive statistics explore the number of queries formed by participants (*e.g.,* mean, median, min, and max) and the rate at which developers form queries over the course of the debugging task.

6.3.5.2 What types of queries are formed by developers during debugging tasks? (RQ 2)

To address this research question, the full set of 710 queries were reviewed by two researchers during the *secound round review* and each individual query was categorized according to the type of information the developer was seeking. A grounded theory approach revealed a set of 28 individual query types being identified through reviewing the observed queries. The full hierarchy of query types will be discussed in Section 6.4.2.

6.3.5.3 *What leads developers to generate new queries and how do they relate to previous queries? (RQ 3)*

To address this question, the full set of 710 queries were reviewed by two researchers during the *secound round review* and each individual query was annotated with the general cause for forming the query and the relationship to any existing queries. A grounded theory approach revealed a set of generic causes and relations describing why developers formed the observed queries. The full discussion of the query generation process is provided in Section 6.4.3.

6.3.5.4 *Which (if any) observed aspects of queries correlate with successful debugging task completion? (RQ 4)*

The final research question focused on identifying aspects that were indicative of successful debugging task completion. To identify these aspects, a set of interesting aspects were identified by reviewing the full set of collected data (after the *second round review*). Then, the developers were separated into two groups, 1) those successfully completing the task, and 2) those that failed to complete the task. These groups were then compared based upon the relevant details as appropriate to each aspect to identify any statistically significant relationships. A sample aspect considered as part of *RQ 4* is listed below along with a sample null hypothesis (H0) and alternative hypothesis (HA):

Query completion rate: the percent of queries resolved by a participant for a given task.

H0: Participants that successfully completed task 1 have a higher query completion rate than those that failed to complete task 1.

HA: Participants that successfully completed task 1 have either the same or a lower query completion rate when compared to those that failed to complete task 1.

The remaining aspects considered during data analysis are as follows:

- **Total Queries Observed**: the total number of queries observed for each participant.
- **Fundamental Query Type Completion Rate**: the percentage of queries of a given fundamental query type resolved by a participant for a given task (*e.g.,* the percentage of Behavior queries resolved by participant 5).
- **Use of Tooling**: the percentage of queries resolved compeltely or in part through the use of available tool support in the development environment.
- **Fundamental Query Type Prevalence**: the percentage of overall queries formed by a given participant that are a given fundamental query type (*e.g.,* percentage of overall queries formed by participant 2 that were Behavior queries).

The full set of aspects considered, the set of aspects found to be indicative of successful task completion, and the implications of these results are discussed in Section 6.4.4.

6.3.6 Threats to Validity

A vital concern is the implied queries (*i.e.,* queries not directly stated by the developer, but evident through their actions). Implied queries are a strength of this study. The implied queries represent queries that might be expressed by developers given the proper query language or tools. To avoid misinterpreting participants, the recorded sessions were independently reviewed by at least two individuals during initial data collection and then reviewed again during the *first round review* by two reviewers collectively before inclusion in the final set of queries. As such, we are confident that every query included represents an occurence of the participant seeking some information during the observed debugging tasks.

The experimental environment imposed a limited set of conditions that may not be present in a typical development environment. In particular, all participants were required to both record

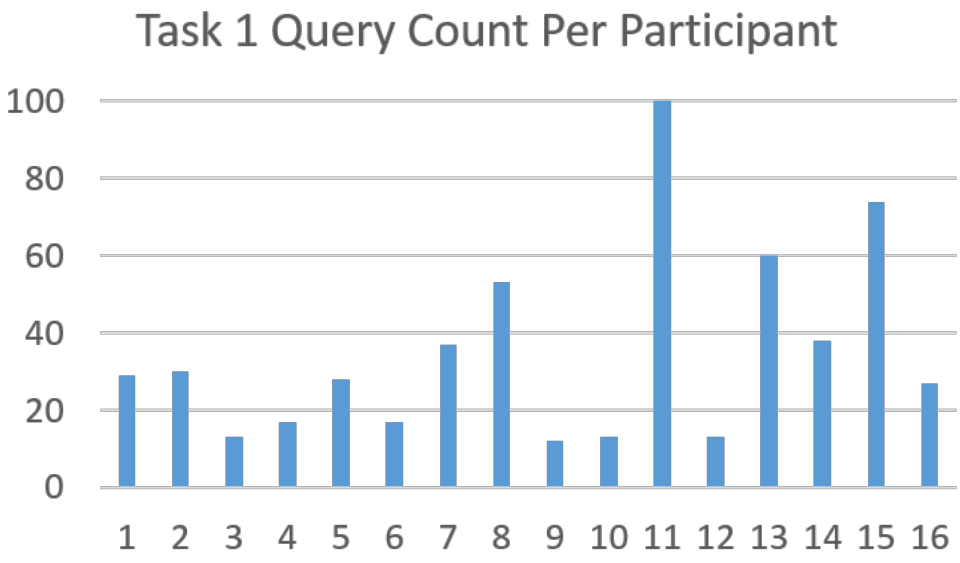
their actions and speak throughout the debugging process. Developers may have acted differently due to this variation. However, the experimental environment is necessary to capture all relevant data. The recording and vocalizations were required for data collection, but may have influenced participant behavior.

These results may not be representative of the overall population. The results provide an initial exploratory evaluation of the use and impact of informal queries during a debugging task. Future replications of the study, both exact and variant, should be undertaken to provide more significance to the results reported here.

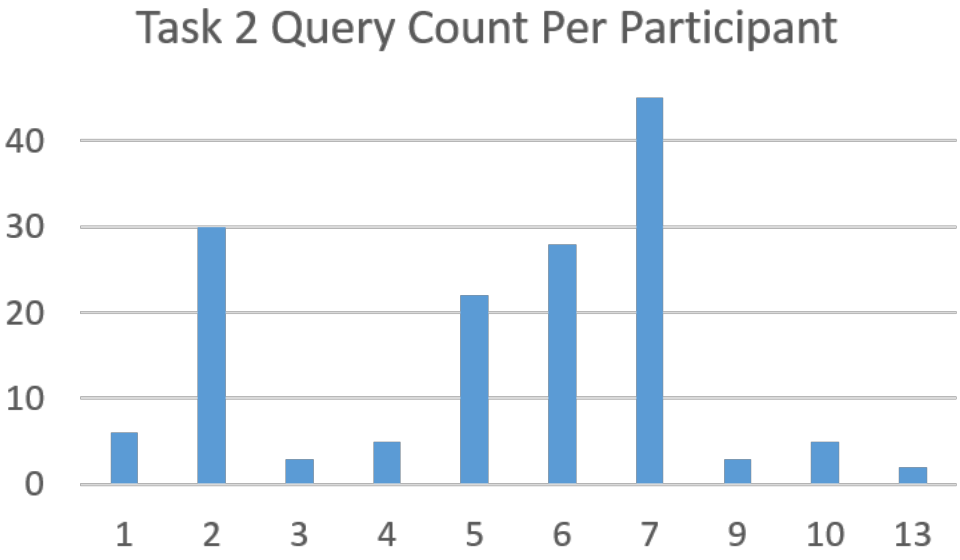
These results may not be typical for developers using languages other than Java. Only 2% of queries were coupled to the specific syntax or semantics of Java, and these queries typically concerned understanding the functionality of a standard library method which is a concern generalizable to many other languages. Therefore, the results reported here are expected to be typical of other OO languages. However, nearly one quarter of the queries observed related to OO concerns such as a Class or Method. While many of these queries could be related to similar constructs in other development paradigms, a replication study is needed to verify the generalizability of results in another development paradigm (*e.g.,* MDE).

6.4 Observations and Discussions Regarding Queries During a Software Debugging Task

This section begins with an overview of the general data collected as a response to *RQ 1* in Section 6.4.1 and to discuss the proliferance of queries and some basic information collected with regards to the queries. Then, the results of the grounded theory analysis performed in support of *RQ 2* and *RQ 3* are provided along with discussion of the implications in Section 6.4.2 and Section 6.4.3. Finally, the results of the statistical analysis performed in support of *RQ 4* are summarized and discussed in Section 6.4.4.



(a) Total Number of Queries Observed per Participant for Task 1



(b) Total Number of Queries Observed per Participant for Task 2

Figure 6.4: Number of Queries Observed for Each Task

6.4.1 Do developers form queries during debugging tasks? (*RQ 1*)

To briefly answer *RQ 1*, each participant demonstrated a minimum of 13 queries across the two tasks attempted. The lowest number of queries identified for a single task was 2 queries presented for task 2. However, the subject that generated 2 queries for task 2 spent less than 3 minutes working on the defect during the allotted hour for the two tasks. Similar experiences were observed for other participants with regards to task 2. The following two paragraphs describe the number of queries observed and how often developers formed queries.

Queries Observed Overall, we had 16 subjects participate in the study. These subjects presented 710 total queries with an average of 44.38 queries per subject over the two tasks. The minimum number of queries spanning both tasks was 13, and the maximum was 100. While task 1 was attempted by all 16 participants, task 2 was only attempted by 10 participants. A breakdown of the total queries observed for each participant for each task is presented in Figure 6.4.

Occurence of Queries During the Debugging Process For each query an estimate of the time it was presented during the task was recorded. Figure 6.5 plots a running sum of the total queries presented at each 5 minute interval during the task. This visualization clearly illustrates that the subjects (both those that succeed and those that failed at the tasks) produced new queries with an overall linear trend during the tasks. This is an interesting observation, because it indicates that queries are a driving force during the debugging process.

Every subject that attempted a given task formed queries during that task. Furthermore, participants formed queries consistently throughout the observed time period spent debugging. Therefore, *RQ 1* can be answered positively with strong evidence supporting the claim that developers do form queries during debugging tasks.

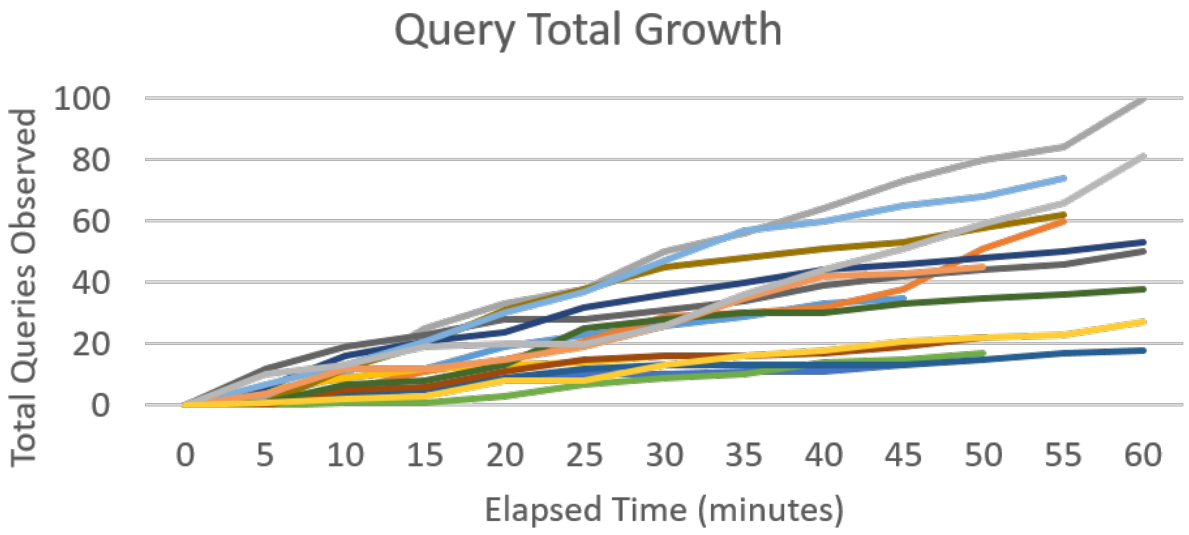


Figure 6.5: Occurence of Queries During the Debugging Process

6.4.2 What types of queries are formed by developers during debugging tasks? (*RQ 2*)

To address *RQ 2*, each of the 710 observed queries was reviewed and categorized according to the type of information the subject was seeking. The analysis revealed a query hierarchy with 7 fundamental types that are decomposed down to 29 specific query types as illustrated in Figure 6.6. Several specific query types were observed to have strong similarities to a subset of the specific types under their fundamental type. These highly related specific query types were then reorganized under group types to further distinguish them from the less related specific types under the same fundamental type. For example, Class and Method are grouped under the Definition group within the Location fundamental type to distinguish these query types from text that is also under the Location fundamental type. Each specific query type represents a general type of information that developers were observed to seek during a debugging task, and the fundamental query types represent general categories of information. These types are presented with the number of occurences indicating the prevalence of developers seeking the specified type of information. As

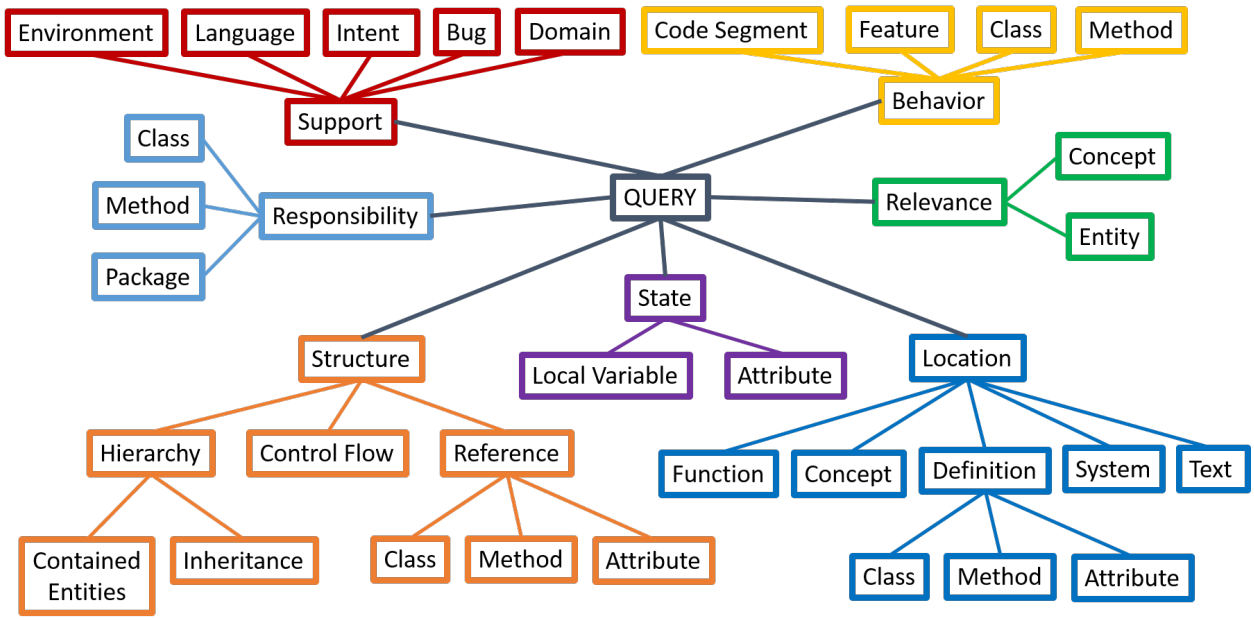


Figure 6.6: Query Type Hierarchy

Query Type	Total Occurences	Percentage of Total Queries
Location	261	36.8%
System	13	1.8%
Function	18	2.5%
Text	152	21.4%
Method Definition	29	4.1%
Class Definition	10	1.4%
Attribute Definition	5	0.7%

Table 6.1: Summary Statistics for Location Queries

such, the query type categorization presented here provides insight that can be used to inform the development of future tools as well as providing a method of reviewing existing tools (with the latter concern discussed further in Section 7.4.3). A detailed description of each fundamental type and the various specific query types follows. Each Specific query type includes a brief description, sample query, the number of occurences observed, and the typical purpose of the query as part of the debugging process.

6.4.2.1 Location

The most commonly occuring fundamental query type was Location. This type is focused primarily on locating entities within the system source. However, the fundamental type also includes System queries that focus on locating non-source entities. **System** queries seek where a file or non-source entity is stored. For example, "Where is the directory that contains the output?" These queries were primarily used to identify relevant input or output entities. **Function** queries seek the entity in the source that handles a specific behavior. For example, "What entity is printing the error message?" These queries were primarily used to identify occurences of specific observable behavior related to the observable error (*e.g.,* localizing the entity generating a given portion of output or setting a key flag such as the failonerror flag). **Concept** queries seek the entity(ies) that handle a given responsibility or feature for a system. For example, "What entity(ies) handle logging?" These queries were primarily used as large encompassing queries seeking entities related to the process associated with the defect (*e.g.,* Copying), and often involved resolving more focused queries. **Text** queries seek the source entity(ies) that include specific text. For example, "Where does 'failonerror' occur in the system?" These queries were well supported by standard tools (though approximately 10% of Text query occurences were resolved manually), and Text queries were commonly used as part of larger query types such as Concept queries.

Definition Group The Definition group contains those queries associated with identifying where in the source specific coding structures are defined or declared. These queries were used primarily during navigating the system source, and are often associated with more complex queries falling under the Fundamental types of Behavior or Responsibility. This group of queries was well supported within Eclipse with tools available to jump from occurences of a Class, Method, or Attribute

Query Type	Total Occurences	Percentage of Total Queries
Structure	54	7.6%
Control Flow	1	0.1%
Contained Entities Hierarchy	15	2.1%
Inheritance Hierarchy	7	1.0%
Method Reference	25	3.5%
Class Reference	3	0.4%
Attribute Reference	3	0/4%

Table 6.2: Summary Statistics for Structure Queries

to the relevant definition or declaration. **Method** queries seek where in the source a given method is defined or declared. For example, "Where is Copy.execute defined?" **Class** queries seek where in the source a given class is defined or declared. For example, "Where is Copy defined?" **Attribute** queries seek where in the source a given attribute is defined. For example, "Where is Copy.failonerror defined?"

6.4.2.2 Structure

The Structure fundamental type is driven by the inherent structure of source code entities. Specific types focus on various relationship types (*e.g.,* parent/child inheritance relationships or caller/callee method relationships). The typical structure query was used as part of source code navigation to expand the search space based upon the relationship to an existing relevant source entity. **Control Flow** queries seek a portion of the source related via the structure of the control flow. For example, "What is the if related to this else?" Control Flow queries can be resolved trivially without need for dedicated tool support or a significant manual process. However, navigating the source to observe control flow and following these relationships is a basic task when examining source code. This query may have had a higher occurence than recorded due to the trivial nature of most occurences.

Hierarchy Group The Hierarchy Group contains query types dedicated to identifying source entities that form a clear hierarchy. These queries can be explored easily through tree-based viewers such as the Package Explorer available in Eclipse. **Contained Entities** queries seek entities contained within a given entity. For example, "What classes exist within org.apache.tools.ant.util?" These queries were commonly used as part of an exploratory process to aid in localizing relevant entities. **Inheritance** queries seek the parent or children of a given class. For example, "What class does Copy inherit from?" These queries were used to further identify the design intent of a given class or to identify relevant inherited structures (*e.g.,* inherited method behavior).

Reference Group The Reference Group contains query types focused on identifying source entities related via a reference (*e.g.,* method call or attribute reference). This group of queries was most commonly used to identify the context in which a method was called or to dive further into the details of a methods behavior by exploring called methods. However, the references to a given class provided context for the use of classes (*e.g.,* "Where is the Copy class used?"), and attribute references explored the use of relevant attributes (*e.g.,* "Where is Copy.failonerror used?"). **Method** queries seek the entity(ies) that reference a given method. For example, "What entity(ies) call Copy.execute?" **Class** queries seek the entity(ies) that reference a given class. For example, "Where is the Copy class used?" **Attribute** queries seek the entity(ies) that reference a given attribute. For example, "What entity(ies) reference Copy.failonerror?"

6.4.2.3 Behavior

The Behavior fundamental type was the second most common category of query presented. These queries focused on the behavior of the system with regards to specific portions of the source or the behavior of features that may be crosscutting concerns relevant to numerous source code en-

Query Type	Total Occurences	Percentage of Total Queries
Behavior	126	17.7%
Method Behavior	45	6.3%
Class Behavior	18	2.5%
Code Segment	16	2.3%
Feature	47	6.6%

Table 6.3: Summary Statistics for Behavior Queries

Query Type	Total Occurences	Percentage of Total Queries
State	26	3.7%
Local Variable	22	3.1%
Attribute State	4	0.6%

Table 6.4: Summary Statistics for State Queries

tities. **Method** queries seek to determine how a particular method behaves at runtime. For example, "How does Copy.doFileOperations work?" **Class** queries seek to determine how a particular class behaves at runtime. For example, "How is failonerror used in Copy?" This query type identifies behavior of concerns that are inherent to a given class such as the use of the failonerror flag to determine when to raise or suppress exceptions in the Copy class. **Code Segment** queries seek to determine how a particular code segment behaves at runtime. For example, "How does this loop work?" These queries focus on small segments of the source code to identify the pecularities of the relevant logic and control flow. This query type was commonly used when investigating the behavior of loops and decision structures within the source code. **Feature** queries seek to determine how a particular feature is performed in the system. For example, "How is copying done?" This query structure concerned large crosscutting behaviors of the system that involve many different entities in the source. Often, Feature queries provided a starting point for investigation as key concerns were identified during the debugging process.

Query Type	Total Occurences	Percentage of Total Queries
Responsibility	62	8.7%
Method Responsibility	23	3.2%
Class Responsibility	37	5.2%
Package Responsibility	2	0.3%

Table 6.5: Summary Statistics for Responsibility Queries

6.4.2.4 State

The State fundamental type is strongly related to both Location Definition queries, Structure Reference queries, and Behavior queries. This type of query focuses on the typing and state of a given variable. These values available at runtime are strongly related to the behavior of the associated entities, and investigating a State query could provide insight into resolving a Behavior query. On the other hand, both Location Definition and Structure Reference queries could be used to resolve a State query. This relationship between query types underscores a key finding of the study (as will be discussed further in Section 6.4.3), which observes that the queries posed by developers are not isolated occurences that are investigated completely before beginning a new branch of exploration. Rather, developers form queries as part of an iterative and branching process. Resolving a State query might not provide enough insight to resolve a defect in isolation, but the contribution of this fundamental query type is typically part of a larger process. **Local Variable** queries seek the type and/or value of a given local variable. For example, "What is the state of toFiles?" This query type was typically used to gather further information regarding a Method Behavior query. **Attribute** queries seek the type and/or value of a given attribute. For example, "What is the state of userInfo.password?" This query type was typically used to gather further information regarding a Method Behavior query.

Query Type	Total Occurences	Percentage of Total Queries
Support	64	9.0%
Environment	16	2.3%
Language	15	2.1%
Intent	11	1.5%
Bug	20	2.8%
Domain	2	0.3%

Table 6.6: Summary Statistics for Support Queries

6.4.2.5 Responsibility

The Responsibility fundamental query type investigates the concerns related to a given source entity. Through these queries, subjects identified the purpose and often began to explore entities at various levels. In the process of resolving these queries, subjects often resorted to exploring comments (especially docstrings), but other queries such as Text Location queries and Inheritance Hierarchy Structure queries were also generated to resolve these queries. This fundamental type was often used as a first step for resolving Relevance queries and Behavior queries for the associated type of source entity (*e.g.,* Method or Class). **Method** queries seek to identify the concerns related to a given method. For example, "What is the purpose of Copy.execute?" **Class** queries seek to identify the concerns related to a given class. For example, "What is the purpose of Copy?" **Package** queries seek to identify the concerns related to a given package. For example, "What is the purpose of TaskDef?" This query type was often formed as developers were exploring the source code and as part of resolving Concept Location queries. Typically, this query type formed a filter that would be followed by subsequent more focused exploration depending on the answer.

6.4.2.6 Support

The Support fundamental query type is centered around providing relevant information to further more focused exploration of the source code. The results of these queries were often used to identify relevant concepts or text to spark a new path of investigation. However, Support queries were also used in support of the processes being employed by the subject, such as learning how to search in Eclipse or how language features and standard library components function. Thus, this query type is not typically focused on directly resolving the defect, but the associated queries provide valuable information to the developer in support of localizing or correcting the defect. **Environment** queries seek information related to the development environment. For example, "How do I search in Eclipse?" These queries were typically used to find tool support for resolving existing queries. **Language** queries seek to understand or identify relevant language features or standard library components. For example, "How do I get the last index of a String?" **Intent** queries seek information from documentation sources. For example, "What does the documentation say about tasks?" Both Intent and Bug queries were typically used as a starting point for investigation into the source of the defect. **Bug** queries seek information from the bug report, relevant error messages, or similar sources referring to the software bug or related information. For example, "What can I gain from the bug report?" **Domain** queries seek information related to the domain of the problem space. For example, "What is a valid URI structure?" Domain queries explore the concepts drawn from the problem space in support of understanding the related structures and processes in the source code.

6.4.2.7 Relevance

The Relevance fundamental query type was typically used by subjects to filter through the source code entities or concepts recognized from the bug report or similar source. This query

Query Type	Total Occurences	Percentage of Total Queries
Relevance	117	16.5%
Entity	116	16.3%
Concept	1	0.1%

Table 6.7: Summary Statistics for Relevance Queries

type served as an early stage of the process where subjects were inspecting superficial elements to determine promising paths for further investigation. **Entity** queries seek to identify if a given source element (*e.g.,* class or method) is relevant to the bug. For example, "Is Copy relevant?" The second most common specific query type, Entity queries were often used to filter through the identified source code entitites. Subjects frequently formed Responsibility queries in support of Entity Relevance queries, and followed with Behavior queries to further investigate the relevant entity. **Concept** queries seek to identify if a recognized concept (*e.g.,* logging) is relevant to the bug. For example, "Could the bug be due to an error in parsing?"

6.4.3 What leads developers to generate new queries and how do they relate to previous queries? (*RQ 3*)

The context and events leading to each observed query were collected during the initial, independent data collection process. The resulting data collection forms for each query were then reviewed during the *first round review*, and a set of general factors that lead subjects to generate each query were identified during the *second round review*. Furthermore, many queries were noted to have related to a prior query as part of the recorded events leading to the query. In response to this observation, during the *first round review* each query was further analyzed to determine if the query was related to a previous query. Then as part of the *second round of review*, a set of general relationships was defined based on the observed relationships. Thus, to address *RQ 3*, a set of general factors leading to generating new queries and query relationships have been identified.

Query Relationships	Total Occurences	Percentage of Total Queries
Subquery	288	40.6%
Filtering Results	50	7.0%
Formed Through	59	8.3%
Follows Result	209	29.4%
No Relation	104	14.6%

Table 6.8: Summary Statistics for Query Relationships

6.4.3.1 Relationship to Existing Query(ies)

Approximately 85.4% of all observed queries were found to be associated with an existing query or queries. Additionally, after carefully reviewing the query relationships as part of the *second round review*, it can be noted that the query relationships demonstrate a clear overarching process used by developers to follow along a series of queries. This observation would seem to support the idea of developers using an information foraging strategy as claimed by Lawrance *et al.* [112]. However, significant further analysis would be needed to claim support of information foraging, and this additional analysis is outside the scope of the current study intent. The query relationships are detailed below along with a brief example.

Subquery A query used to assist in resolving a prior query was considered to be related as a *subquery* of the prior query. For example, the Text Location query, "Where does the text 'copy' occur in the system?" could be used to aid in resolving the Concept Location query, "What entity(ies) are related to copying?" Thus, the Text Location query would be identified as a *subquery* of the Concept Location query.

Filtering Results A query used to reduce the set of results identified from a previous query was considered to be related as *filtering the results* of the prior query. For example, the Entity Relevance

query, “Is the method CopyFile.execute executed when exercising the defect?” might be used to filter the results of the Text Location query, “Where does ‘failonerror’ occur in the system?”

Formed Through A query that is not directly related to a prior query, but was formed due to some information encountered during the investigation of a prior query was considered to be related as a query *formed through* a prior query. For example, while investigating the Class Responsibility query, “What is the purpose of the Copy class?” the subject could right-click on the class name in Eclipse and be presented with the context sensitive option *Open Type Hierarchy*. As a result of identifying this new environment feature, the subject might form the Environment Support query, "What is the purpose of the Open Type Hierarchy tool?" Thus, the Environment Support query is formed through investigating the Class Responsibility query.

Follows Result A query that is formed directly as a response to the results of a prior query was considered to be related as *following the results* of the prior query. For example, the Entity Relevance query, “Is the Copy.doFileOperations method invoked when exercising the defect?” would result in identifying that the Copy.doFileOperations is invoked, and result in forming the Method Behavior query, “How does the Copy.doFileOperations method work?” Thus, the Method Behavior query would be identified as *following the results* of the Entity Relevance query.

6.4.3.2 Query Generation Factor

After carefully reviewing the context and the events leading to each query as identified in the data collection forms, along with further investigating the relevant video segments for numerous queries, a set of 6 generic query generation factors was identified. Each query generation factor is listed below along with a brief description and discussion of any significant query relationship(s) associated with the factor.

Query Generation Factor	Total Occurences	Percentage of Total Queries
Finding A New Path	113	15.9%
Search Results	42	5.9%
Related Entity	45	6.3%
Conceptual Trigger	77	10.8%
Textual Trigger	73	10.3%
Further Investigation	360	50.7%

Table 6.9: Summary Statistics for Query Generation Factors

Finding A New Path The second most common reason subjects generated a new query was to identify a new starting point of exploration. This process was typically not related to any existing query as would seem expected. However, occasionally developers did return to the results of a prior query to find a new path of exploration. For example, a developer might return to the results of an initial Text Location query to identify a new set of entities to investigate.

Search Results Often subjects identified a set of entities or source locations as the result of a prior query. These were collectively referred to as *search results*, and were a common source of new queries. The *search results* factor was nearly always related to a prior query through the *filtering results* relationship. Thus, it can be noted that queries identifying large result sets are followed by a process of filtering the results to identify relevant entities. An immediate implication of this outcome is that work to identify relevance of result in an automated fashion through textual or other similarity measures are likely to present a significant benefit to developers.

Related Entity As subjects investigated the source code of the system, often queries were spawned through identifying related entities such as a method call or inheritance relation. This generation factor was most commonly formed through investigating an existing query that spawned an unrelated query to investigate the related entity. Investigating related entities was also observed

to follow the results of a prior query (*e.g.,* following the Entity Relevance query, "Is Copy.execute invoked during exercising the defect?" could be followed by an Entity Relevance query on the Copy.doFileOperations method called by the Copy.execute method) or as a subquery to help resolve a prior query (*e.g.,* the Method Behavior query "How does the Copy.doFileOperations method work?" could be used to help resolve the Method Behavior query, "How does the Copy.execute method work?"). This generation factor would offer support to visualization strategies such as call graphs or class diagrams that aid developers in identifying related entities during the debugging process.

Conceptual Trigger Subjects were observed to generate some queries due to identifying a particular concept. For example, a developer might inspect the bug report and identify the concept of Copy as being related to the defect, and then investigate the Concept Location query, "What entity(ies) are related to Copy?" Queries generated from conceptual triggers were typically subqueries, and the result of the investigation spawned by the concept was used to inform the prior query.

Textual Trigger Similar to conceptual triggers, developers formed some queries as a result of identifying some specific portion of text. Textual triggers were also most commonly used as subqueries.

Further Investigation The most common cause for subjects to generate new queries was to further investigate an entity, concept, or other concern. This result shows a strong support for the Information Foraging model of the debugging process as proposed by Lawrance *et al.* [112]. This generation factor was typically related to prior queries as either a subquery or following the results

of the prior query. For example, after resolving the Entity Relevance query, "Is Copy.execute invoked during exercising the defect?" the developer might investigate the Method Behavior query, "How does the Copy.execute method work?"

6.4.4 Which (if any) observed aspects of queries correlate with successful debugging task completion? (*RQ 4*)

As a final contribution of the work presented in this chapter, several aspects related to queries were investigated to identify if subjects that were successful in completing the task presented a statistically signficiant difference in that aspect's measure as compared to subjects that were not successful. The purpose of this investigation was to identify some measure of the aspects related to the observed queries that indicates the developer is more likely to succeed in completing the debugging task. The following will describe each aspect investigated and the results of the appropriate statistical tests.

Successful Task Completion For each subject, the researchers recorded if the subject successfully completed each task attempted. To resolve the task successfully, the subject was expected to clearly identify the source of the defect. All but one participant that clearly identified the source of the defect, also corrected the defect. The one exceptional subject clearly identified the source of the defect (*i.e.,* stated the incorrect logic and explained why the logic was incorrect), but ran out of time before correcting the defect. A total of seven subjects successfully completed both tasks. An additional one subject successfully completed task 1 (*i.e.,* corrected bug 1) only, and two subjects successfully completed task 2 only (*i.e.,* corrected bug 2). Leaving a total of six subjects that failed to complete either task successfully.

Task 2 was only attempted by ten of the sixteen participants. Of the six participants that did not attempt task 2, five failed to complete task 1. Furthermore, several participants attempting

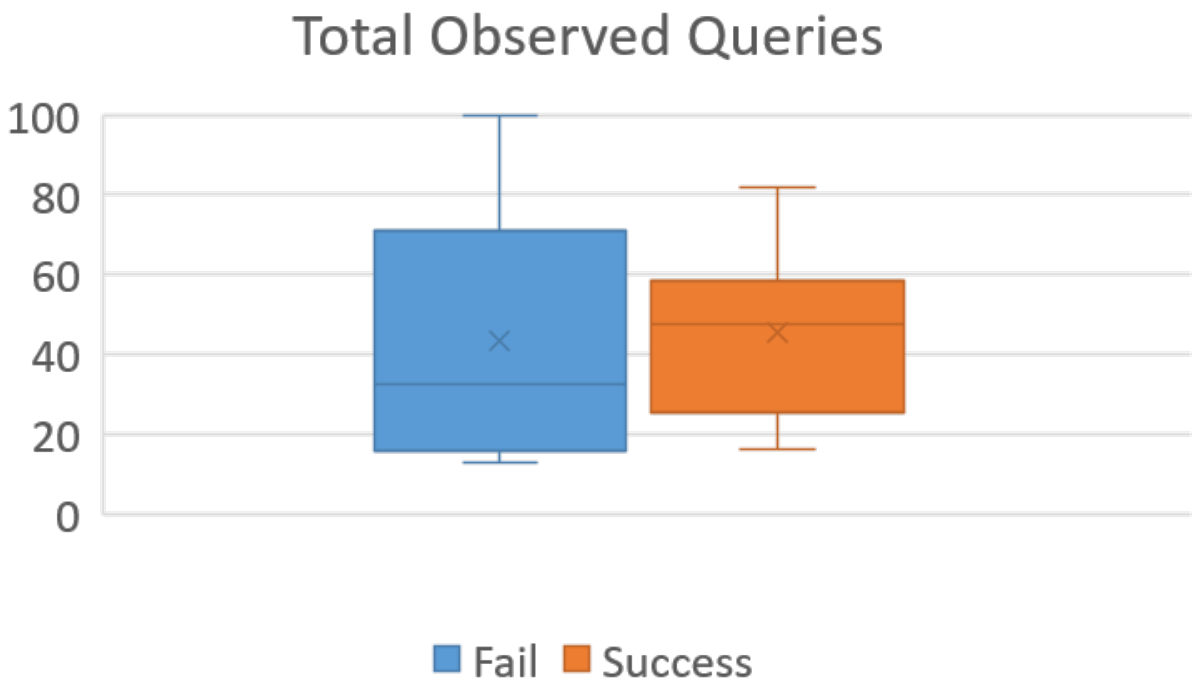


Figure 6.7: Total Observed Queries for Participants that Successfully Completed Task 1 (Success) or Did Not Complete Task 1 (Fail)

task 2 (including the one that attempted and failed task 2) spent the majority of their allotted time completing task 1, and did not have a comparable amount of time remaining to attempt the second task. Thus, due to the overall inconsistent amount of time and effort spent on the second task and the reduced set of participants, only queries related to task 1 are considered in the following results.

6.4.4.1 Total Queries Observed

The first measure explored was the total number of queries presented by a developer. A developer who explored a larger set of queries might have spent more time inspecting unrelated queries (as was observed of at least one subject). Alternatively, a developer presenting a smaller set of queries might have spent the majority of their time failing to identify a relevant path to explore more completely, which would then generate more queries (again, this was observed of at least one subject). The total queries observed by participants is summarized in the box plots presented

in Figure 6.7. The box plots presents the distribution of total observed queries per participant for each group (successful and unsuccessful).

To investigate this aspect, the following null hypothesis and alternative hypothesis were tested using a two-tailed t-test with $\alpha = 0.05$.

H0: Subjects that successfully completed task 1 have generated the same number of queries as those that failed to complete task 1.

HA: Subjects that successfully completed task 1 have generated a different number of queries that those that failed to complete task 1.

Despite the large variance in number of queries presented, there was no significant relationship identified between the number of queries presented by the developers and success in completing the two tasks. This indicates that presenting more or less queries during a debugging task was not related to successfully completing the debugging task.

6.4.4.2 Overall Query Completion Rate

To identify a resolved query, the recordings were carefully reviewed during the initial independent data collection and during the *first round review* to identify if the subject discovered an answer for the query. A clearly identified lack of result was considered an answer for a given query. For example, a developer might search for the occurence of a portion of the error message for bug 1 in the CopyFile class. The message does not occur in this class (*i.e.,* the error message originates from the Copy class), but a rigorous exploration of the CopyFile class would identify that the error message does not occur in the CopyFile class. The range of observed query completion rates for the participants that successfully completed task 1 or did not complete task 1 is presented in Figure 6.8.

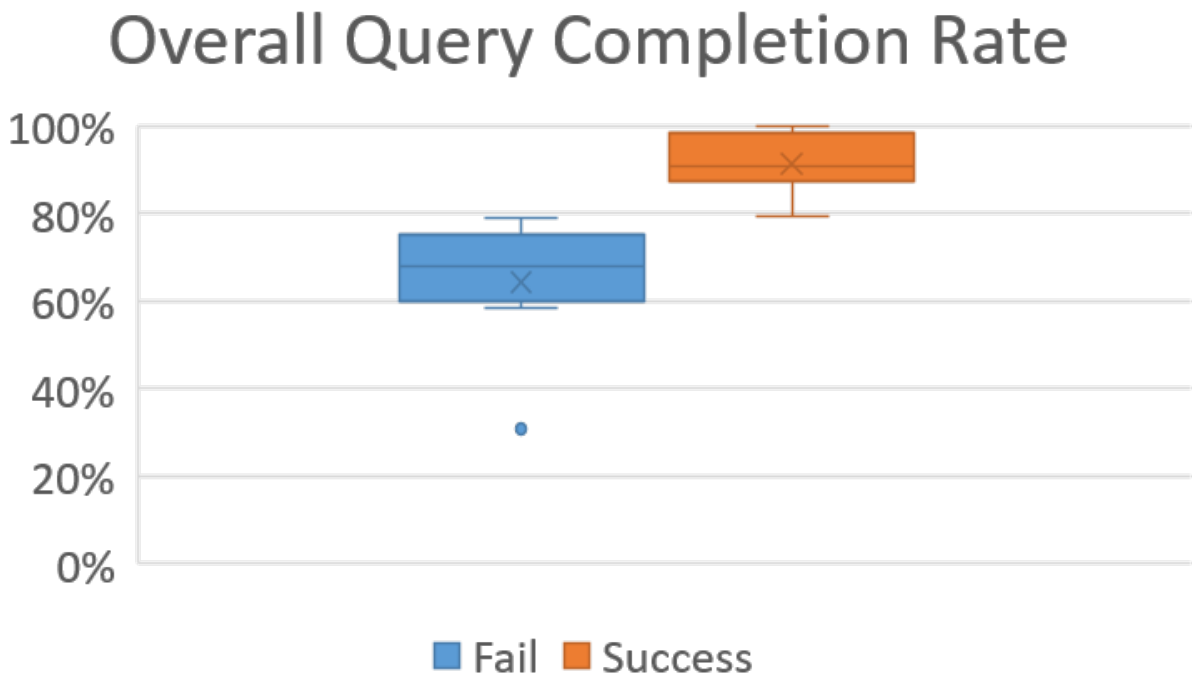


Figure 6.8: Overall Query Completion Rate Distribution for Participants that Successfully Completed Task 1 (Success) or Did Not Complete Task 1 (Fail)

The ability of developers to resolve queries posed during the debugging process could contribute to completing the debugging task successfully. Thus, the query completion rate (or percentage of queries a subject resolved) was explored in a similar fashion as the total queries observed. To investigate query completion rate the following null hypothesiss and alternative hypothesis were tested using a single-tailed t-test with $\alpha = 0.05$.

H0: Subjects that successfully completed task 1 displayed a higher query completion rate than those that failed to complete task 1.

HA: Subjects that successfully completed task 1 displayed a lower or the same query completion rate compared to those that failed to complete task 1.

With $p < 0.01$, the subjects successfully completing task 1 were confirmed to display a statistically significant increase in overall query completion rate. The successful participants had

an average query completion rate of 91.3% (minimum of 83% and maximum of 100%), and unsuccessful subjects had an average query completion rate of 64.3% (minimum of 31% and maximum of 83%). These results indicate that the percentage of queries resolved by developers is related to successfully completing the debugging task. This would support providing relevant tool support to aid developers in resolving queries.

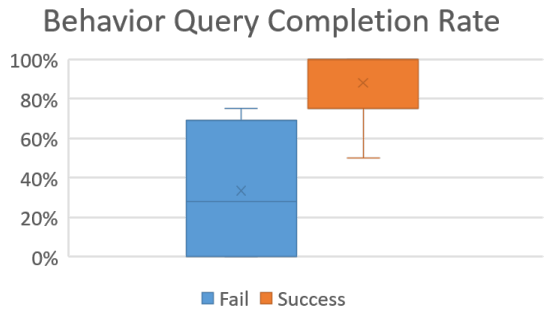
6.4.4.3 Fundamental Query Type Completion Rate

To further investigate the significance of query completion rate, the query completion rate of each fundamental query type was investigated to identify statistically significant differences between the two groups (*i.e.,* successful and unsuccessful subjects). The completion rates for each fundamental query type are summarized in the box plots presented in Figure 6.9. To investigate query completion rate for each fundamental query type, a null hypothesis and alternative hypothesis similar to the following were tested for each fundamental query type using a single-tailed t-test with $\alpha = 0.05$.

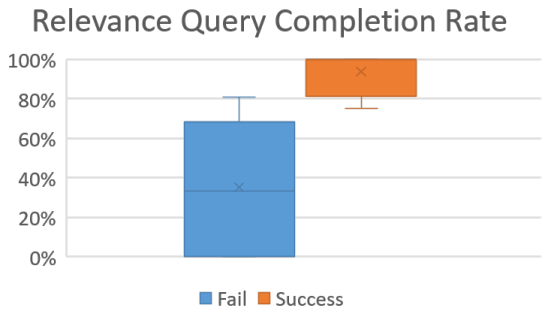
H0: Subjects that successfully completed task 1 displayed a higher query completion rate for Behavior queries than those that failed to complete task 1.

HA: Subjects that successfully completed task 1 displayed a lower or the same query completion rate for Behavior queries compared to those that failed to complete task 1.

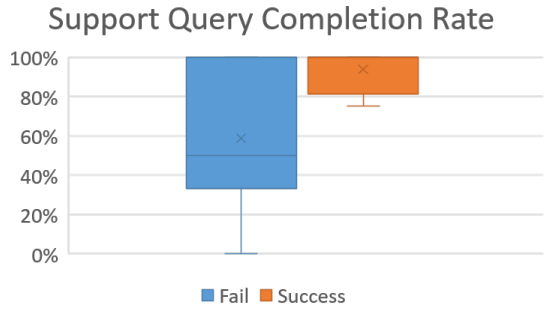
The resulting statistical tests identified four fundamental query types to display a significantly larger query completion rate: Behavior with $p < 0.01$, Responsibility with $p < 0.01$, Relevance with $p = 0.01$, and Support with $p = 0.05$. The significance of Behavior, Responsibility, and Relevance queries is rather intuitive as identifying relevant entities and understanding both their purpose and behavior are significant concerns when localizing and correcting a software defect. The significance of Support queries indicates that understanding the language features and



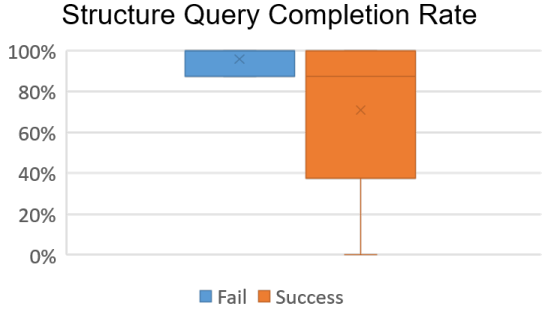
(a) Behavior Query Completion Rate



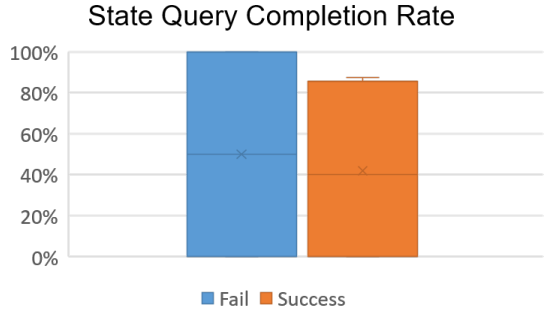
(b) Relevance Query Completion Rate



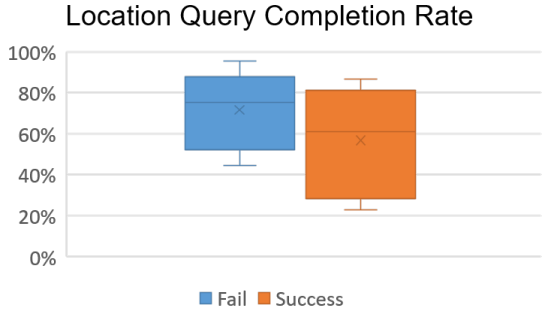
(c) Support Query Completion Rate



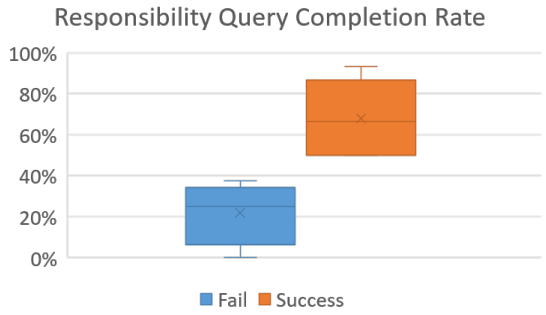
(d) Structure Query Completion Rate



(e) State Query Completion Rate



(f) Location Query Completion Rate



(g) Responsibility Query Completion Rate

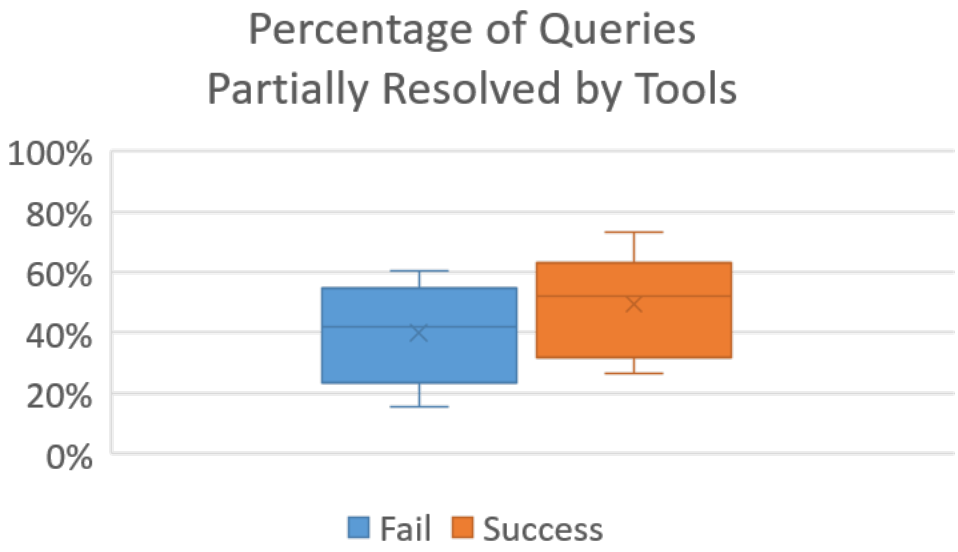
Figure 6.9: Fundamental Query Completion Rate Distribution for Participants that Successfully Completed Task 1 (Success) or Did Not Complete Task 1 (Fail)

environment, as well as being able to identify relevant information from the bug report and similar entities are also significant concerns for the debugging process.

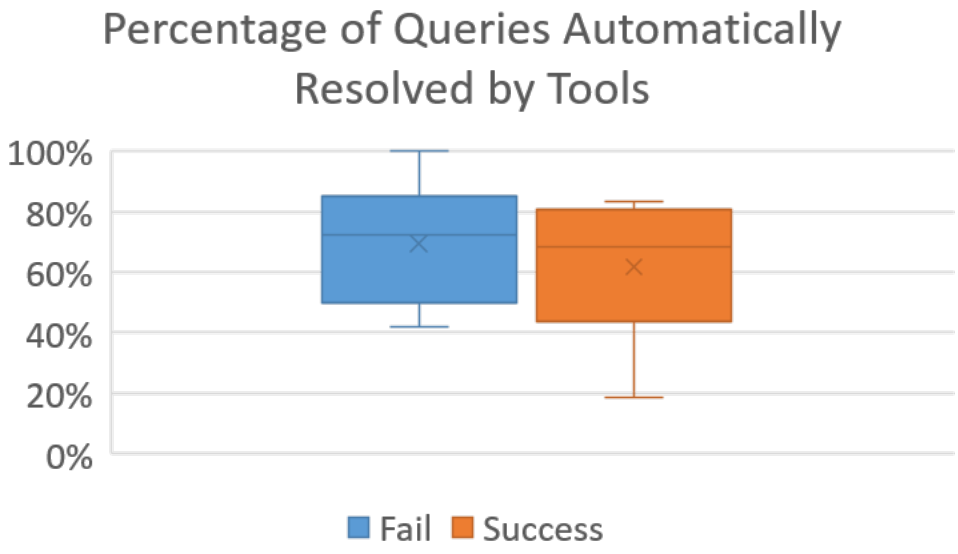
Other fundamental query types either possessed very similar completion rates (*e.g.,* Structural queries that had a 100% rate for successful and a 96% rate for unsuccessful) or too few occurences (*e.g.,* State queries with only 3.7% of overall queries observed) to identify a significant p value.

6.4.4.4 Use of Tooling

Every developer made use of some tooling in the development environment ranging from basic text searches to more complex tools, such as viewing type hierarchies or call hierarchies. The use of tools was investigated to determine if the developers that successfully completed the task were observed to use more tools from the development environment. The percentage of queries observed that were resolved at least partially through tool support is summarized in the box plots presented in Figure 6.10a. Additionally, data was collected regarding if the tool used for a given query was able to resolve the query in a completely automated process (*e.g.,* linking directly to the definition of a method from an identified method call). The percentage of queries observed that were resolved completely through tool support is summarized in the box plots presented in Figure 6.10b. These two aspects (*i.e.,* number of queries resolved using tooling, and number of queries resolved automatically using using tooling) were investigated similar to prior occurences using a two-tailed t-test with target $\alpha = 0.05$. The test did not indicate a statistically significant difference between successful and unsuccessful subjects in the number of queries resolved using tools. This result only indicates that the observed developers did not demonstrate a significant difference in their use of tools. A distinct experiment would be required to investigate this aspect completely. A control group composed of developers that are deprived of tools completely with an additional



(a) Percentage of Queries Resolved with the Assistance of Tool Support



(b) Percentage of Queries Resolved Automatically by Tool Support

Figure 6.10: Summary of Tool Usage for Participants that Successfully Completed Task 1 (Success) or Did Not Complete Task 1 (Fail)

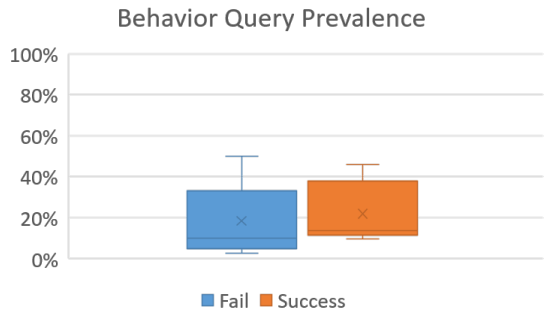
group(s) provided with some set of tooling would be more appropriate to investigate the significance of tooling used by developers. However, we can conclude that despite variations in the experience with Ecipse (subjects ranged from 0 years of experience to 10+ years of experience with Eclipse), successful subjects did not demonstrate a significant difference in their use of tools compared to unsuccessful subjects.

6.4.4.5 Fundamental Query Type Prevalence

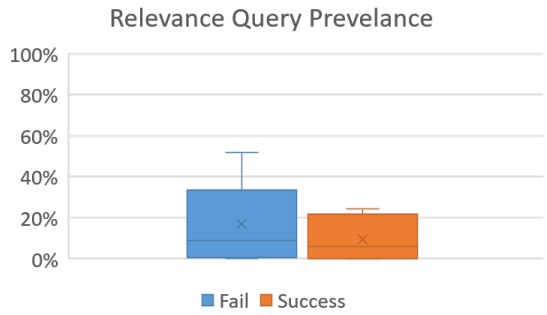
The final aspect considered was the percentage of queries presented by a user corresponding to a specific fundamental query type. The prevalence (*i.e.,* percentage of overall queries for a given participant) for each fundamental query type are summarized in the box plots presented in Figure 6.11. This aspect explores if the focus on a certain fundamental query type was significanty different for successful subjects as compared to unsuccessful subjects. The goal was to identify if developers could be encouraged to focus on certain query types to improve debugging effectiveness. However, despite the significant difference in completion rate for several fundamental query types, no significant difference was found for the prevalence of any fundamental query type. This aspect was investigated in the same manner as previous aspects, using a two-tailed t-test with $\alpha = 0.05$. As a result, it can be concluded that successful subjects did not demonstrate a significant difference in focus on any specific fundamental query type.

6.5 Conclusion

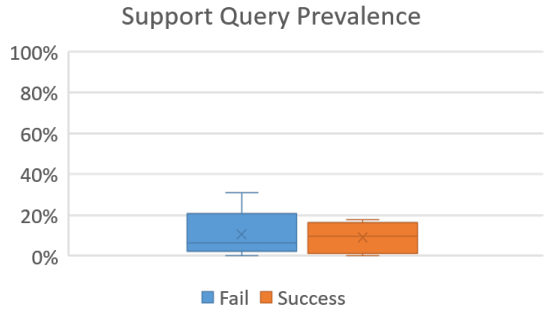
This chapter has presented the design and results of an exploratory empirical study exploring the formation and use of queries during the debugging process of entry level developers working in the Eclipse environment on two defects in the industrial quality OO system, ANT. The results presented in this chapter support that queries are a natural part of a developer's debugging and maintenance process. A set of 7 fundamental query types covering 29 specific query types



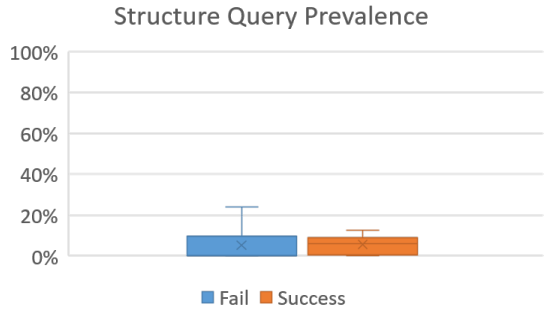
(a) Behavior Query Prevalence



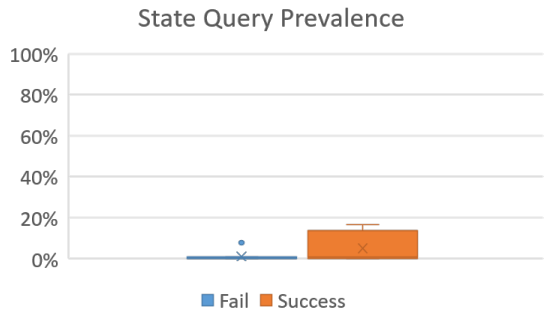
(b) Relevance Query Prevalence



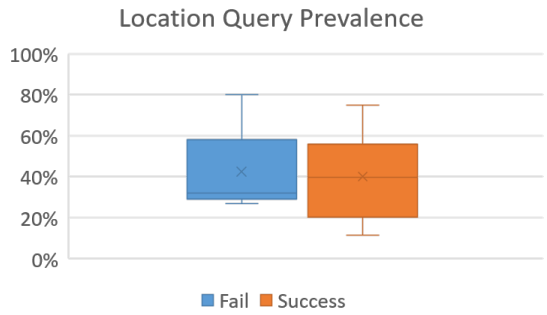
(c) Support Query Prevalence



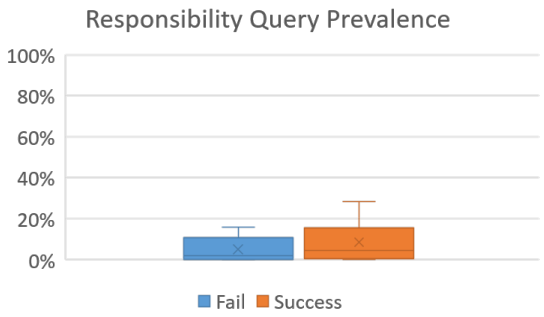
(d) Structure Query Prevalence



(e) State Query Prevalence



(f) Location Query Prevalence



(g) Responsibility Query Prevalence

Figure 6.11: Fundamental Query Prevalence Distribution for Participants that Successfully Completed Task 1 (Success) or Did Not Complete Task 1 (Fail)

were identified using a grounded theory approach to analyze the type of information the queries, presented by participants in the study during their debugging process, were seeking. The query type categorization provides insight into the type of information developers seek during a debugging task, and can be used to support both development of new tools and review of existing tools support. Additionally, a set of query generation factors and query relationships were identified to explain why developers formed new queries. The exploration of these query generation factors and query relationships indicated that developers form individual queries as part of a larger investigation process. This finding seems to support the prior work of Lawrance *et al.* that investigated the Information Foraging model as a model of the process used by developers during the debugging process. However, further analysis outside the scope of the work presented in this chapter is necessary to confirm this finding. Finally, the analysis indicate that overall query completion rate and the query completion rate specifically for Behavior queries, Relevance queries, and Support queries are significant factors indicating successful task completion.

CHAPTER 7

CONCLUSION & FUTURE WORK

MDE has provided a new paradigm for software development focused on the use of software models. However, traditional SE concerns such as maintenance and evolution are still common and significant concerns for practicioners of MDE [29]. As discussed in Chapter 1, researchers have noted a lack of research concerning the significant software maintenance task of debugging in an MDE context [16]. While many debugging techniques have been explored in a GPL context (*e.g.,* omniscient debugging [6–10] and QBD [5, 11–15]), relatvely little work has explored debugging techniques in an MDE context.

Over the course of the work presented in this dissertation, I have made several contributions to the MDE community and the general SE community. I have contributed to the MDE community through the investigation of omniscient debugging for MTs and xDSMLs. This investigation has resulted in a prototype supporting omniscient debugging of MTs in AToMPM and a prototype supporting omniscient debugging of xDSMLs in GEMOC Studio. The investigation into omniscient debugging of MTs contributed a new scalable technique to support efficient omniscient traversals, and the collaborative investigation of omniscient debugging for xDSMLs contributed a novel set of multi-dimensional omniscient traversal features and a novel interactive visualization of execution history to support omniscient debugging. I have also contributed to the MDE community through identifying a set of collaboration scenarios for MDE and investigating a live, cloud-based solution to support these scenarios. Finally, I have contributed to the general SE community through

an exploratory empirical study regarding the formation, use, and impact of queries presented by developers while debugging an OO system. The contributions of this project include identifying a categorization of queries according to the information the query was seeking, a general set of factors that led developers to generate new queries, and a set of general relationships describing how newly formed queries related to prior queries presented during the debugging process. Furthermore, the exploration of queries during the debugging process presented a set of aspects found to be linked to successful completion of the observed debugging task.

7.1 Collaborative Modeling

As the first contribution of my dissertation work, I presented the results of a project that explored the challenges and some initial solutions to supporting collaborative modeling processes in a distributed environment. The growing trend for collaborative environments was discussed, and the needs of domain experts to work in a coordinated manner to model complex systems were motivated The exploration of collaborative modeling identified a set of 4 collaboration scenarios that describe how modelers could collaborate in a modeling environment. These scenarios ranged from simple sharing of elements to more complicated scenarios involving interdependent models. The cloud-based architecture of AToMPM supporting collaborative modeling was presented, . Discussion then detailed how the architecture ensures consistency and synchronization among artifacts produced by each stakeholder across the identified collaboration scenarios. Preliminary results evaluating the scalability and performance of the architecture reveal that the Controllers portion of the architecture is efficient with regards to the scaling of number of concurrent users, but distributing the low level processing and storage of models remains an open challenge (see Chapter 3 for more details).

7.2 Omniscient Debugging for MTs and xDSMLs

The primary contribution of the work presented in this dissertation investigated the application of omniscient debugging, a traditional approach to identify and resolve software errors drawn from existing GPL literature, to debugging MDE artifacts. The investigation focused primarily on the application of omniscient debugging to MTs with additional collaborative work extending this investigation to xDSMLs.

The investigation of omniscient debugging for MTs culminated in a prototype in the AToMPM environment, AODB, and an empirical evaluation of the technique regarding performance in terms of execution time and scalability in terms of memory usage. The technique supported efficient omniscient traversal through the use of two novel algorithms designed to construct a `MacroStep` (*i.e.,* a construct used to traverse multiple incremental prior states of execution) without replicating unnecessary change operations. The primary results of the empirical evaluation indicated that the technique performs similarly to a standard stepwise execution debugger in the same environment and that the addition of omniscient debugging features had only a modest impact on scalability. Additionally, the evaluation explored the comparative performance of the two `MacroStep` building algorithms identifying results that indicate how to utilize the algorithms together most efficiently, as well as exploring the efficiency of using the omniscient traversal features as compared to restarting the transformation system (see Chapter 4 for more details).

While the investigation into omniscient debugging for MTs focused on basic omniscient traversal features, the investigation regarding xDMSLs introduced multi-dimensional omniscient traversal features and a novel interactive visualization of execution history to support omniscient debugging. The investigation into omniscient debugging of xDSMLs resulted in a technique sup-

porting generic omniscient debugging (including both basic and multi-dimensional omniscient features) for the various DSMLs typical of xDSMLs in the GEMOC Studio environment. The technique relied on the generation of domain-specific trace management facilities to support arbitrary xDSMLs. Similar to AODB, a prototype was constructed in the GEMOC Studio environment and evaluated with regards to performance and scalability. The results indicated that the generative technique supporting omniscient debugging performed similarly to a stepwise execution debugger in terms of execution time, and the generative technique both performed (in terms of execution time) and scaled (in terms of memory usage) more efficiently than a clone-based omniscient debugger (see Chapter 5 for more details).

7.3 Exploring Query Formation and Impact

Finally, the debugging process used by developers was investigated with special emphasis on the impact and formation of queries through an exploratory empirical study. The results presented in Chapter 6 found that queries are a natural part of a developer's debugging and maintenance process, and identified a set of query types based upon the type of information the observed queries were seeking. Additionally, a set of query generation factors and query relationships were identified to explain why developers formed new queries. The exploration of these query generation factors and query relationships indicated support for the prior work of Lawrance *et al.* that investigated the Information Foraging model to describe the process used by developers during the debugging process [112]. Finally, the results presented indicated several aspects related to query completion that were significantly linked to successful completion of the debugging tasks (see Chapter 6 for more details).

7.4 Perspectives on Future Research

As I look forward to continuing my research beyond the work presented in this dissertation, I have identified several prospects to further my research in the area of debugging and MDE.

7.4.1 User Study of Omniscient Debugging for MTs

I plan to pursue a user study to validate the omniscient debugging technique presented in Chapter 4. The technique has been evaluated through analyzing the performance and scalability of a prototype implemented in AToMPM. However, the technique should also be evaluated when used in practice by developers to identify impact of the tool when used in a practical debugging scenario. A study design and relevant materials have already been prepared in preparation for this work, and I need only identify a set of participants to perform this study. This study will exlore the impact of omniscient debugging when applied to the debugging process in terms of efficiency and effectivenes, as well as investigating the perceptions and opinions of developers concerning the technique. These perceptions must be understood as they will directly impact the adoption of the technique by practitioners.

7.4.2 Exploring Query Formation and Impact for MDE

Following the results of the study of query formation and impact in a GPL context as presented in Chapter 6. I plan to pursue empirical research concerning queries and the debugging process in an MDE context. The new paradigm of MDE pushes developers closer to a domain view of software development through the use of software models. As a result there is fundamental shift from more traditional software development focused on the use of GPLs and MDE. However, many traditional SE concerns such as debugging are still relevant in an MDE context. Thus, a replication study with modification would be warranted to evaluate the concerns discussed in Chapter 6 within

this new context. This study would be designed to both explore the formation and impact of queries in the context of MDE, as well as draw conclusions comparing the observations directly with those from a GPL context (drawn from the results presented in Chapter 6).

7.4.3 Exploring Tool Support and the Debugging Process

I also plan to further explore the debugging process used by developers to improve the pairing of tool support and developer exploration. I intend to perform a comprehensive review of existing modern tool support. The resulting set of tools will be mapped to the fundamental and specific query types discussed in Chapter 6 to identify any gaps and to review the quality of coverage provided by existing modern SE tools. Additionally, this work will further explore the process used by developers seeking to confirm or deny existing proposed theories (*e.g.,* information foraging theory [112]). The results of this exploration will aim to guide the development of future tool support through identifying how tool support can assist and improve the debugging process. Furthermore, exploring the debugging process through observing and analyzing the processes used by developers will enable future researchers to identify best practices that can be used to educate new generations of practicioners.

7.4.4 Efficient, Distributed Storage and Processing of Models

I have also identified, as a result of the work presented in Chapter 3, the need for improved solutions to the challenge of collaborative modeling. The key concern for this continued work is how to efficiently distribute storage and low-level processing of models. The significant challenges will be to create a low-level structure that is aware of the unique relationships and processes specific to models. Such concerns include the ability to properly coordinate metamodels and models across storage solutions to enable direct processing, and to support advanced processing such as MTs and conformance checking.

REFERENCES

[1] E. Syriani, J. Kienzle, and H. Vangheluwe, "Exceptional transformations," in *Proceedings of the 3rd International Conference on Model Transformation*, June 2010, pp. 199–214.

[2] H. Ergin and E. Syriani, "AToMPM solution for the IMDB case study," in *Proceedings of the 7th Transformation Tool Contest at STAF*, July 2014, pp. 134–138.

[3] J. Gray, J.-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle, "Domain-specific modeling," *Handbook of Dynamic System Modeling*, vol. 7, pp. 7–1, 2007.

[4] M. Seifert and S. Katscher, "Debugging triple graph grammar-based model transformations," in *Proceedings of 6th International Fujaba Days*, Dresden, Germany, 2008.

[5] A. J. Ko and B. A. Myers, "Designing the whyline: A debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, April 2004, pp. 151–158.

[6] B. Lewis, "Debugging backwards in time," in *Proceedings of the Fifth International Workshop on Automated Debugging*, October 2003.

[7] A. Lienhard, J. Fierz, and O. Nierstrasz, "Flow-centric, back-in-time debugging," in *Proceedings of Objects, Components, Models and Patterns*, July 2009, pp. 272–288.

[8] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proceedings of 22nd European Conference on Object-Oriented Programing*, July 2008, pp. 592–615.

[9] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, October 2007, pp. 535–552.

[10] G. Pothier and E. Tanter, "Back to the future: Omniscient debugging," *IEEE Software*, vol. 26, no. 6, pp. 78–85, November 2009.

[11] A. J. Ko, "Debugging by asking questions about program output," in *Proceedings of the 28th International Conference on Software Engineering*, May 2006, pp. 989–992.

[12] A. J. Ko and B. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, May 2008, pp. 301–310.

[13] R. Lencevicius, U. Hölzle, and A. K. Singh, "Query-based debugging of object-oriented programs," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1997, pp. 304–317.

[14] R. Lencevicius, U. Holzle, and A. K. Singh, "Dynamic query-based debugging of object-oriented programs," *Automated Software Engineering*, vol. 10, pp. 39–74, October 2003.

[15] R. Lencevicius, "Query-based debugging," 1999, dissertation, Accessed: 09-02-2015. [Online]. Available: http://www.cs.ucsb.edu/research/tech-reports/1999-27

[16] R. Mannadiar and H. Vangheluwe, "Debugging in domain-specific modelling," in *Proceedings of the 4th International Conference on Software Language Engineering*, July 2011, pp. 276–285.

[17] J. Schoenboeck, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Catch me if you can – debugging support for model transformations," in *Proceedings of the 12th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2009, pp. 5–20.

[18] J. de Lara and E. Guerra, "Formal support for QVT-relations with coloured Petri nets," in *Proceedings of the 12th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2009, pp. 256–270.

[19] M. Wimmer, G. Kappel, J. Schoenboeck, A. Kusel, W. Retschitzegger, and W. Schwinger, "A Petri net based debugging environment for QVT relations," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, November 2009, pp. 3–14.

[20] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai, "The graph rewriting and transformation language: GReAT," in *Proceedings of the 3rd International Workshop on Graph Based Tools*, September 2006.

[21] F. Jouault, "Loosely coupled traceability for ATL," in *Proceedings of the European Conference on Model-Driven Architecture - Foundations and Applications Workshop on Traceability*, November 2005, pp. 29–37.

[22] M. Lawley and J. Steel, "Practical declarative model transformation with Tefkat," in *Satellite Events at the 8th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2006, pp. 139–150.

[23] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "AToMPM: A web-based modeling environment," in *Joint Proceedings of the MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, October 2013.

[24] J. Corley and E. Syriani, "A cloud architecture for an extensible multi-paradigm modeling environment," in *Joint Proceedings of 8th Internations Conference on Model-Driven Engineering, Languages, and Systems Poster Session and the ACM Student Research Competition*, October 2014, pp. 6–10.

[25] J. Corley, H. Ergin, S. Van Mierlo, and E. Syriani, *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global, January 2016, ch. Cloud-based Multi-View Modeling Environments.

[26] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, "Incremental pattern matching in the viatra model transformation system," in *Proceedings of the 3rd International Workshop on Graph and Model Transformations*, May 2008, pp. 25–32.

[27] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *Proceedings of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance*, September 2003, pp. 446–453.

[28] L. Geiger and A. Zündorf, "Graph based debugging with Fujaba," in *Proccedings of the 1st International Workshop on Graph-based Tools*, no. 2, October 2002, p. 112.

[29] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, September 2003.

[30] B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, and J. Gray, "Globalizing modeling languages," *IEEE Computer*, vol. 47, no. 6, pp. 10–13, June 2014.

[31] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the Satellite Events at the 8th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2006, pp. 128–138.

[32] J. Corley, "Exploring omniscient debugging for model transformations," in *Joint Proceedings of 8th Internations Conference on Model-Driven Engineering, Languages, and Systems Poster Session and the ACM Student Research Competition*, October 2014, pp. 63–68.

[33] J. Corley, B. P. Eddy, and J. Gray, "Towards efficient and scalable omniscient debugging for model transformations," in *14th Workshop on Domain-Specific Modeling*, October 2014, pp. 13–18.

[34] J. Corley, B. P. Eddy, E. Syriani, and J. Gray, "Efficient and scalable omniscient debugging for model transformations," *Software Quality Journal*, pp. 1–42, 2016.

[35] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xDSMLs," in *Proceedings of the 8th International Conference on Software Language Engineering*, October 2015, pp. 137–148.

[36] B. P. Eddy and J. Corley, "Searching for answers: An exploratory study of the formation, use, and impact of queries during debugging," in *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, October 2014, pp. 51–52.

[37] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, March 2006.

[38] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, May 2005.

[39] J. Bézivin and O. Gerbé, "Towards a precise definition of the OMG/MDA framework," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, November 2001, p. 273.

[40] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, December 2006.

[41] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal - Model-Driven Software Development*, vol. 45, no. 3, pp. 621–645, July 2006.

[42] "MDA guide version 1.0.1," 2003, accessed: 09-02-2015. [Online]. Available: http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf

[43] "Unified modeling language (OMG UML) infrastructure version 2.4.1," 2013, accessed: 09-02-2015. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

[44] "Unified modeling language (OMG UML) superstructure version 2.4.1," 2013, accessed: 09-02-2015. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

[45] "Meta object facility (MOF) version 2.4.1," 2013, accessed: 09-02-2015. [Online]. Available: http://www.omg.org/spec/MOF/2.4.1/PDF

[46] C. Atkinson and T. Kühne, "Profiles in a strict metamodeling framework," *Science of Computer Programming*, vol. 44, no. 1, pp. 5–22, July 2002.

[47] "Meta object facility (mof) 2.0 query/view/transformation (QVT) version 1.2," 2015, accessed: 09-02-2015. [Online]. Available: http://www.omg.org/spec/QVT/1.2

[48] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, June 2008.

[49] E. Syriani and H. Vangheluwe, "A modular timed model transformation language," *Software & Systems Modeling*, vol. 12, no. 2, pp. 387–414, May 2013.

[50] P. Stevens, "Bidirectional model transformations in QVT: semantic issues and open questions," *Software & Systems Modeling*, vol. 9, no. 7, pp. 7–20, January 2010.

[51] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet, "Mashup of metalanguages and its implementation in the Kermeta language workbench," *Software & Systems Modeling*, vol. 14, no. 2, pp. 905–920, May 2015.

[52] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable dsmls based on fUML," in *Proceedings of the6th International Conference on Software Language Engineering*, October 2013, pp. 56–75.

[53] M. Hibberd, M. Lawley, and K. Raymond, "Forensic debugging of model transformations," in *Proceedings of the 10th International Conference on Model-Driven Engineering Languages and Systems*, 2007, pp. 589–604.

[54] M. V. Zelkowitz, "Reversible execution," *Communications of the ACM*, vol. 16, no. 9, p. 566, September 1973.

[55] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Requirements traceability: Theory and practice," *Annals of Software Engineering*, vol. 3, no. 1, pp. 397–415, January 1997.

[56] M. Ducasse, "Coca: an automated debugger for C," in *Proceedings of the 21st International Conference on Software Engineering*, May 1999, pp. 504–513.

[57] "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[58] J. Engblom, "A review of reverse debugging," in *Proceedings of the 4th System, Software, SoC and Silicon Debug Conference*, September 2012, pp. 1–6.

[59] A. Potanin, J. Noble, and R. Biddle, "Snapshot query-based debugging," in *Proceedings of the 6th Australian Software Engineering Conference*, April 2004, p. 251.

[60] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in Eclipse," in *Proceedings of the 5th Workshop on Eclipse Technology eXchange at OOPSLA*, October 2007, pp. 31–35.

[61] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, October, 2005, pp. 365–383.

[62] M. F. van Amstel, M. G. J. van den Brand, and A. Serebrenik, "Traceability visualization in model transformations with TraceVis," in *Proceedings of the 5th International Conference on Model Transformation*, May 2012, pp. 152–159.

[63] P. Dhoolia, S. Mani, V. S. Sinha, and S. Sinha, "Debugging model-transformation failures using dynamic tainting," in *Proceedings of the 24th European Conference on Object-Oriented Programming*, June 2010, pp. 26–51.

[64] M. Seifert, "Opportunities and challenges for traceable graph rewriting systems," in *Proceedings of the 3rd International Workshop on Graph and Model Transformations at ICSE*, May 2008, pp. 33–36.

[65] J.-r. Falleri, M. Huchard, and C. Nebut, "Towards a traceability framework for model transformations in Kermeta," in *Proceedings of the 2nd ECMDA Traceability Workshop at ECMDA-FA*, July 2006, pp. 26–35.

[66] K. Zeng, Y. Guo, and C. K. Angelov, "Graphical model debugger framework for embedded systems," in *Proceedings of the 13th Conference on Design, Automation and Test in Europe*, March 2010, pp. 87–92.

[67] S. V. Mierlo, "Explicit modeling of model debugging and experimentation," in *Proceedings of Doctoral Symposium co-located with 17th International Conference on Model-Driven Engineering Languages and Systems*, October 2014, pp. 1–8.

[68] D. Lucrédio, R. P. de M. Fortes, and J. Whittle, "Moogle: A model search engine," in *Proceedings of the 11th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2008, pp. 296–310.

[69] D. H. Akehurst and B. Bordbar, "On querying UML data models with OCL," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, October 2001, pp. 91–103.

[70] G. Bergmann, A. Hegedüs, A. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "Integrating efficient model queries in state-of-the-art EMF tools," in *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, C. A. Furia and S. Nanz, Eds., 2012, pp. 1–8.

[71] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *Proceedings of the 4th International Conference on Model Transformation*, June 2011, pp. 167–182.

[72] C. Hobatr and B. A. Malloy, "Using OCL-queries for debugging C++," in *Proceedings of the 23rd International Conference on Software Engineering*, May 2001, pp. 839–840.

[73] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and A. Lédeczi, "Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure," in *8th International Workshop on Multi-Paradigm Modeling at MODELS*, October 2014, pp. 41–60.

[74] M. Voelter, "Md* best practices," *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, September 2009.

[75] J. Corley and E. Syriani, "Modeling as a service: Scalability and performance of the cloud architecture of atompm," in *Proceedings of 4th Internations Conference on Model-Driven Engineering and Software Development*, February 2016.

[76] S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne, "Multi-level modelling in the modelverse," in *8th International Workshop on Multi-Paradigm Modeling at MODELS*, October 2014, pp. 83–92.

[77] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Addison Wesley Professional, December 2008.

[78] S. Kelly, K. Lyytinen, and M. Rossi, "MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment," in *Proceedings of the 8th International Conference on Advanced Information Systems Engineering*, May 1996, pp. 1–21.

[79] S. Hiya, K. Hisazumi, A. Fukuda, and T. Nakanishi, "CLOOCA : Web-based tool for domain specific modeling," in *Join Proceedings of the MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, October 2013, pp. 31–35.

[80] B. Combemale, J. Deantoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, and R. France, "Reifying concurrency for executable metamodeling," in *Proceedings of the 6th International Conference on Software Language Engineering*, October 2013, pp. 365–384.

[81] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *Journal of Network and Computer Applications Special Issue on Trusted Computing and Communications*, vol. 35, no. 3, pp. 1086–1105, May 2012.

[82] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "MDEForge: An extensible web-based modeling platform," in *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud at MoDELS*, September 2014, pp. 66–75.

[83] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer Science & Business Media, 2011.

[84] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. K. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software & Systems Modeling*, pp. 1–38, 2014.

[85] "IEEE 610-12.1990 IEEE standard glossary of software engineering terminology," https://standards.ieee.org/findstds/standard/610.12-1990.html.

[86] Z. Ujhelyi, A. Horváth, and D. Varró, "Dynamic backward slicing of model transformations," in *Proceedings of the 5th International Conference on Verification & Validation*, April 2012, pp. 1–10.

[87] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, "State-based model slicing: A survey," *ACM Computing Surveys*, vol. 45, no. 4, pp. 53:1–53:36, August 2013.

[88] D. Di Ruscio, D. S. Kolovos, and N. Matragkas, "Scalability in model-driven engineering: BigMDE '13 workshop summary," in *Proceedings of the 1st International Workshop on Scalability in Model-Driven Engineering at STAF*, June 2013, pp. 1:1–1:2.

[89] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, "Parallel in-place model transformations with LinTra," in *Proceedings of the 3rd International Workshop on Scalability in Model-Driven Engineering at STAF*, July 2015, pp. 52–62.

[90] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró, "Incquery-d: A distributed incremental model query framework in the cloud," in *Proceedings of the 17th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2014, pp. 653–669.

[91] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, "A research roadmap towards achieving scalability in model-driven engineering," in *Proceedings of the 1st International Workshop on Scalability in Model-Driven Engineering at STAF*, June 2013, pp. 2:1–2:10.

[92] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry, "A generative approach to define rich domain-specific trace metamodels," in *Proceedings of the 11th European Conference on Modelling Foundations and Applications*, July 2015, pp. 45–61.

[93] E. Syriani, H. Vangheluwe, and B. LaShomb, "T-core: A framework for custom-built model transformation engines," *Software & Systems Modeling*, vol. 14, no. 3, pp. 1215–1243, July 2015.

[94] T. Horn, C. Krause, and M. Tichy, "The transformation tool contest 2014 movie database case," in *Proceedings of the 7th Transformation Tool Contest at STAF*, July 2014, p. 93.

[95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[96] B. Combemale, X. Crégut, and M. Pantel, "A design pattern to build executable DSMLs and associated V&V tools," in *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, December 2012, pp. 282–287.

[97] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier, "Formalizing execution semantics of UML profiles with fUML models," in *Proceedings of the 17th International Conference on Model Driven Engineering, Languages, & Systems*, October 2014.

[98] "Semantics of a foundational subset for executable UML models, v 1.1," Object Management Group, August 2013, last accessed: 5/17/2016. [Online]. Available: http://www.omg.org/spec/FUML/1.1/

[99] S. Maoz, J. O. Ringert, and B. Rumpe, "Addiff: Semantic differencing for activity diagrams," in *Joint Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13th European Software Engineering Conference*, September 2011, pp. 179–189.

[100] S. Maoz and D. Harel, "On tracing reactive systems," *Software & Systems Modeling*, vol. 10, no. 4, pp. 447–468, October 2011.

[101] Á. Hegedüs, I. Ráth, and D. Varró, "Replaying execution trace models for dynamic modeling languages," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 56, no. 3, pp. 71–82, March 2012.

[102] E. A. Aboussoror, I. Ober, and I. Ober, "Seeing errors: Model-driven simulation trace visualization," in *Proceedings of the 15th International Conference on Model-Driven Engineering, Languages, and Systems*, October 2012, pp. 480–496.

[103] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: a framework for developing domain-specific debuggers," in *Proceedings of the 7th International Conference on Software Language Engineering*, October 2014, pp. 102–121.

[104] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer, "Promobox: A framework for generating domain-specific property languages," in *Proceedings of the 7th International Conference on Software Language Engineering*, October 2014, pp. 1–20.

[105] F. Hilken, L. Hamann, and M. Gogolla, "Transformation of UML and OCL models into filmstrip models," in *Proceedings of the 7th International Conference on Model Transformations*, October 2014, pp. 170–185.

[106] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[107] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July 2008.

[108] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp. 185–194.

[109] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory: strategies for qualitative research*, 1st ed. Aldine Publishing Company, 1967.

[110] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," *Software & Systems Modeling*, pp. 1–19, 2015.

[111] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec 2006.

[112] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, Feb 2013.

[113] T. Boren and J. Ramey, "Thinking aloud: Reconciling theory and practice," *IEEE Transactions on Professional Communication*, vol. 43, no. 3, pp. 261–278, Sep 2000.

Appendices

APPENDIX A
IRB CERTIFICATES

Office for Research

Institutional Review Board for the
Protection of Human Subjects

THE UNIVERSITY OF
ALABAMA
RESEARCH

June 28, 2013

Jonathan Corley
Department of Computer Science
College of Engineering
Box 870290

Re: IRB # 13-OR-227, "How Developers Debug: An Exploratory Study"

Dear Mr. Corley:

The University of Alabama Institutional Review Board has granted approval for your proposed research.

Your application has been given expedited approval according to 45 CFR part 46. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your application will expire on June 27, 2014. If your research will continue beyond this date, please complete the relevant portions of the IRB Renewal Application. If you wish to modify the application, please complete the Modification of an Approved Protocol Form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, please complete the Request for Study Closure Form.

Please use reproductions of the IRB approved stamped consent forms to obtain consent from your participants.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,

[redacted]

Carpantato T. Myles, MSM, CIM
Director & Research Compliance Officer
Office for Research Compliance
The University of Alabama

358 Rose Administration Building
Box 870127
Tuscaloosa, Alabama 35487-0127
(205) 348-8461
FAX (205) 348-7189
TOLL FREE (877) 820-3066