End-to-end Fine-grained Traceability Analysis in Model Transformations and Transformation Chains

by

Victor Guana Garces

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science
University of Alberta

Abstract

Over the last few decades, *model-driven code generation* has been the flagship paradigm used to promote adoption of model-driven engineering among the general software-engineering community. *Model-driven code generators* integrate *model-to-model* and *model-to-text* transformations to build applications that systematically differ from each other. Typically, generators use multi-step transformation chains to translate high-level application specifications, captured using domain-specific languages, into executable artifacts such as code and deployment scripts.

A key challenge in the construction of development environments for model transformations involves the analysis and visualization of traceability information. Access to fine-grained traceability links enables developers to assess evolutionary scenarios in transformation ecosystems, to effectively debug complex binding expressions, and to accurately determine the metamodel coverage of transformation chains.

Unfortunately, current traceability-analysis techniques do not consider *implicit bindings* when collecting traceability information from complex transformation expressions. *Implicit bindings* manipulate, constrain, or navigate the structure of a metamodel in order to realize the final intent of a transformation expression. Furthermore, they do not conceive *model-to-model* and *model-to-text* transformations as equal constituent elements of a unified model-driven engineering toolbox. This effectively limits their usability in the context of non-trivial *model-driven code generators*. To the best of our knowledge, the effectiveness of current traceability analysis, and the development environments built on top of them, has not been validated in empirical studies with real developers.

In this work, we address these shortcomings. We propose an end-to-end fine-grained traceability-analysis technique for individual *model-to-model* and *model-to-text* transformations, as well as model-transformation chains combining the two. Our analysis technique is based on a traceability framework that considers traceability links as symbolic dependencies between metamodels, transformation expressions, and generated artifacts. Furthermore, we introduce ChainTracker, a traceability-analysis environment. We evaluated the completeness of our traceability-analysis technique using 25 *model-to-model* and 18 *model-to-text* transformations from the *ATLZoo* and the *Acceleo Example Repository*. Our analysis technique achieved an overall fine-grained traceability coverage of 91% and 85%, respectively. Furthermore, we evaluated the usability of ChainTracker in an empirical study in which 25 developers completed traceability-driven tasks in two *model-driven code generators* of different complexity. We found statistically significant evidence that ChainTracker improves the accuracy and efficiency of developers by between 22% and 900%.

Preface

This thesis is an original work by Victor Guana. The empirical study, which this thesis is a part, received research ethics approval from the University of Alberta Research Ethics Board on October 27, 2014. Project name *ChainTracker Usability Evaluation* No. Pro00051612,

The literature review presented in Chapter 2.4, and the usability evaluation of ChainTracker in Chapters 6 and 7 have been published in:

- **Guana, V.**, Eleni, S. End-to-end Model-transformation Comprehension Through Fine-grained Traceability Information. International Journal on Software and Systems Modeling (SoSYM), 2017.
- **Guana, V.**, Stroulia, E. How Do Developers Solve Software-engineering Tasks on Model-based Code Generators? An Empirical Study Design. First International Workshop on Human Factors in Modeling. Sep. 27-28, 2015. Ottawa, Canada.

ChainTracker was built with the assistance of Kelsey Gaboriau. Kelsey contributed in the construction of ChainTracker's visualization canvas. Some portions of the description of ChainTracker in Chapter 6 have been published in:

- **Guana, V**., Stroulia, E. Reflecting on Model-based Code Generators Using Traceability Information. 18th International Conference on Model Driven Engineering Languages and Systems (MODELS) - Tool Demo. Sep. 27-28, 2015. Ottawa, Canada.
- **Guana, V.**, Gaboriau, K. Stroulia, E. ChainTracker: Towards a Comprehensive Tool for Building Code-generation Environments. 30th IEEE International Conference on Software Maintenance and Evolution (ICSM) - Tool Demo. Sep 27 - October 3, 2014. Victoria, Canada.

Information included in Chapter 2.1 was published in:

- **Guana, V.** Supporting Maintenance Tasks on Transformational Code Generation Environments. Doctoral Symposium, 35th International Conference on Software Engineering (ICSE). May 18-26, 2013. San Francisco. USA.

The literature review presented in Chapter 2.2, as well as the conceptual framework presented in Chapter 4, and the evaluation of our traceability-analysis technique in Chapter 5 were published in:

- **Guana, V.**, Eleni, S. Traceability Analysis in Model-to-Model and Model-to-Text Transformations and Transformation Chains. International Journal on Software and Systems Modeling (SoSYM), 2017 *(submitted)*
- **Guana, V.** Stroulia, E. Backward Propagation of Code Refinements on Transformational Code Generation Environments. 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). May 18, 2013. San Francisco, California. USA.
- **Guana, V.**, Stroulia, E.: ChainTracker, a Model-transformation Trace Analysis Tool for Code-generation Environments. 7th International Conference on Model Transformation (ICMT). July 21-25. 2014. York, UK.

PhyDSL (Chapter 3) was built with the assistance of Vina Nguyen. Vina contributed with the refactoring of model transformations, as well as during the release of its corresponding Eclipse plugin. Vina assisted during the evaluation of our traceability-analysis technique (Chapter 5). Chapter 3 has been published in:

- **Guana, V.**, Stroulia, E., Nguyen, V. Building a Game Engine: A Tale of Modern Model-Driven Engineering. 4th International Workshop on Games and Software Engineering (GAS15). May 18, 2015. Florence, Italy,
- **Guana, V.**, Stroulia, E. PhyDSL: A Code-generation Environment for 2D Physics-based Games. IEEE Games, Entertainment, and Media Conference (IEEE GEM). October 22-24, 2014. Toronto. Canada.

Finally, other research work completed during the development of this thesis, but not discussed in this manuscript, include:

- Wallace, B., Knoefel, F., Goubran, R., Masson, P., Baker, A., Allard, B., Stroulia, E., **Guana, V**. Monitoring Cognitive Ability in Patients with Moderate Dementia Using a Modified "Whack-a-Mole. International Symposium on Medical Measurements and Applications. IEEE. May 7, 2017. Rochester. USA.
- Tong, T., **Guana, V.**, Jovanovic, A., Tran, F., Mozafari, G., Chingell, M., and Stroulia, E. Rapid Deployment and Evaluation of Mobile Serious Games: A Cognitive Assessment Case Study. 7th International Conference on Advances in Information Technology. Nov 22-25, 2015. Thailand.

- **Guana, V.**, Xiang, T., Zhang, H., Schepens, E., Stroulia, E. UnderControl an Educational Serious-Game for Reproductive Health. ACM SIGCHI Annual Symposium on Computer-Human Interaction in Play (CHIPLAY'14). October 19-22, 2014. Toronto. ON. Canada.
- Tsantalis, N., **Guana, V.**, Stroulia, E., Hindle, A. A Multidimensional Empirical Study on Refactoring Activity. CASCON. November 18th -20th, 2013. Markham, ON. Canada.
- Cheung, T., **Guana, V.**, Labas, M., Lock, A., Liu, L., Stroulia, E. . Computerized Tablet-based Cancellation Assessment for Spatial Inattention. Proceedings of the Canadian Association of Occupational Therapists Annual Conference, May 29-June 1, 2013. Victoria, Canada.
- **Guana, V.**, Rocha, F., Hindle, A., Stroulia, E., 2012. Do Stars Align? Multidimensional Analysis of Android's Layered Architecture. 9th Conference on Mining Software Repositories (MSR). Zurich, Switzerland. *(Best Challenge Paper Award)*

Information Boxes:

We use *light bulb boxes* to highlight important information, including section summaries and terminology clarifications.

We use *fire boxes* to highlight the main take home messages of this work, including insights and observations obtained from statistical analyses.

Contents

List of Tables

List of Figures

1

Introduction

The main goal of model-driven engineering (MDE) is to boost the productivity of developers by enabling them to work at a higher level of abstraction [1, 2]. In principle, MDE promotes the use of domain-specific languages (DSLs) ruled by metamodels, with concepts closer to the domain of the desired application, rather than the ones offered by a general-purpose programing language [3].

Over the last few decades, *model-driven code generation* has been the flagship paradigm used to promote adoption of model-driven engineering among the general software-engineering community. *Model-driven code generators* integrate *model-to-model* (M2M) and *model-to-text* (M2T) transformations to build applications that systematically differ from each other. Typically, generators use multi-step transformation chains (MTCs) that gradually translate high-level application specifications into executable artifacts, such as code and deployment scripts. In the transformation process, M2M transformations are used to split, merge, or augment the information provided in the initial specification, potentially producing multiple intermediate models that capture different system concerns [4]. In turn, M2T transformations take the intermediate models, and produce executable artifacts based on templates that have been previously engineered for reuse.

Although *model-driven code generators* offer multiple opportunities for the development of software systems, including faster time-to-market, higher code quality, and a more inclusive development experience, adoption of the paradigm continues to be slow [2]. Empirical studies have reported that although *model-driven code generators* increase the productivity of developers by between 20% and 800% [1, 2], in most cases, their maintenance costs penalize developers with 27% of reduced productivity [2, 5].

Like any software system, *model-driven code generators* are bound to evolve. Evolutionary changes in *model-driven code generators* can be classified in two scenarios of evolution: *metamodel evolution* and *platform evolution* [6, 7]. In the metamodel-evolution scenario, changes to the gen-

erator's underlying languages are required to improve their expressiveness, and to better capture information relevant to the to-be-constructed systems. In the platform-evolution scenario, changes to generated artifacts are required to meet new requirements not captured by the generation infrastructure. Such modifications also include code refactoring for design improvement, performance tuning for mission-critical systems, energy consumption optimization, and bug fixes [8]. In both scenarios of evolution, M2M and M2T transformations potentially need to be modified in order to reflect changes in a systematic way. Due to the numerous dependent artifacts in a transformation ecosystem, supporting its evolution is a challenging and error-prone task [9, 10, 11]

1.1 Problem Statement

In order to increase the adoption of MDE practices, we need development environments specifically tailored to support the construction and maintenance of model transformations [5, 11, 12]. This is particularly relevant in the context of modern software developers who continuously experiment with development technologies and quickly abandon tools with no evident economic return [13, 14, 5]. Unfortunately, current development environments for model transformations do not consider M2M and M2T transformation as a part of a unified model-driven engineering toolbox, effectively limiting their usability in the construction of non-trivial *model-driven code generators*. In fact, empirical studies conducted with industrial practitioners, have shown that 30% of developers consider development environments as one of the main barriers to adoption [5].

A key challenge in the construction of development environments for model transformations involves the analysis of traceability information [9]. In this thesis, we are interested in providing a transformation-analysis technique, and corresponding traceability-analysis environment, to enable developers to find *the right code to look at* during the construction and maintenance of *model-driven code generators*. Traceability information is fundamental to enable developers answer traceability-driven questions in the following activities.

- *Supporting change impact analysis in evolving metamodels, transformations, and generated codebases*: End-to-end traceability links enable developers to assess the impact of metamodel and platform evolution in individual transformations, as well as in transformation chains. [15, 16, 17]. This activity is based on the following questions.
  - What metamodel elements are derived using a given metamodel element or property? *i.e., metamodel-downstream dependency analysis.*
  - What binding expressions does a given metamodel element or property depend on? *i.e., metamodel-upstream dependency analysis.*

  - What metamodel elements and properties does a given template line depend on? *i.e., template-upstream dependency analysis.*
  - What transformation bindings intervene in the generation of a given line of code? *i.e., code-upstream dependency analysis.*

- *Debugging transformations during development and maintenance*: Fine-grained traceability links enable developers to evaluate the correctness of transformations whether in isolation or in transformation chains [18]. This activity motivates additional questions, such as:

  - What metamodel elements and attributes are used in a given binding expression?
  - What underlying dependency relationships exist due to binding expressions that use *helper rules*?

- *Evaluating the design of a transformation composition to improve its qualities*: Fine-grained traceability links can be used to visualize the execution mechanics of complex transformation ecosystems. This enables developers to efficiently evaluate metamodel design alternatives, and transformation-modularization strategies [19, 20, 21, 10]. This activity involves questions, such as:

  - What is the order of precedence for the correct execution of the transformations in my ecosystem? Are there interdependent branches in my transformation chain?
  - How are individual binding expressions linked throughout my entire transformation chain? *i.e., binding-expression dependency analysis.*

- *Supporting the assessment of metamodel coverage and the analysis of orphan metamodel elements*: Precise coverage measurements help developers to prune evolving metamodels, and to design transformation oracles [22, 23, 24]. This task relies on the ability of developers to answer questions, such as:

  - Are there unused binding expressions or transformations rules in the transformations of my ecosystem? *i.e., transformation refactoring.*
  - How well is the information captured by the metamodels in my ecosystem used by my transformations? Are there uncovered elements or attributes? *i.e., fine-grained metamodel coverage analysis.*

Unfortunately, the term *traceability* does not have a universally accepted definition in the MDE community [17]. This has caused a research landscape in which the usability and completeness of traceability-analysis approaches can not be precisely assessed. Surveys [25, 17, 26] have compiled

the existing work towards collecting traceability links from M2M and M2T transformations. They conceive traceability links as dependency relationships between a variety of transformation artifacts, at different levels of granularity and abstraction. One line of research [27, 28, 29, 30, 31, 32] considers traceability links as dependency relationships between the models used by a transformation and those produced after its execution, i.e., *traceability at the model level*. Yet another [20, 33, 21] conceives traceability links as the symbolic dependencies between the metamodels used by a transformation, and its corresponding binding expressions, *i.e., traceability at the metamodel level. Traceability at the model level* is mainly concerned in supporting model co-evolution; it enables developers to synchronize models with evolving metamodels and vice versa. On the other hand, *traceability at the metamodel level* focuses on supporting developers maintaining transformations and metamodels as generation specifications change over time [34].

Current traceability-analysis techniques, and the analysis environments built on them, suffer from multiple limitations. They are unable to analyze fine-grained end-to-end traceability information in M2M and M2T transformation chains. Furthermore, they do not consider *implicit bindings* when collecting traceability information from complex transformation expressions. *Implicit bindings* manipulate, constrain, or navigate the structure of a metamodel in order to realize the final intent of a transformation expression. This limits their usability in the context of transformation ecosystems, whose main purpose is to generate textual/executable artifacts from high-level specifications. More importantly, the evaluation of current traceability-analysis techniques is often based on simplified or conceptual transformation examples. Typically, these evaluations only include one example to illustrate the mechanics of the analysis, rather than their technical completeness, i.e., the binding expressions they are able to analyse, and the granularity level of their traceability links. Indeed, the practical limitations of current analysis techniques are yet to be explored. Finally, researchers claim that current transformation-analysis environments make developers more efficient and effective at maintaining *model-driven code generators*. However, to the best of our knowledge, their usability has not been validated in empirical studies.

In view of these shortcomings, this thesis presents a metamodel-level end-to-end traceability framework and analysis technique for model transformations. Our traceability-analysis technique has been implemented in ChainTracker [35, 36, 37] an integrated analysis environment for ATL [38] and Acceleo [39] transformation technologies. Our traceability framework is generalizable to transformation languages that use OCL [40] as their underlying model manipulation language. It conceives traceability information in individual M2M and M2T transformations, as well as in transformation chains combining the two. We have evaluated the completeness of our analysis technique in the context of 14 individual transformation projects from the *ATLZoo*[1], and 5 code-

[1] `https://www.eclipse.org/atl/atlTransformations/`

generation projects from the *Acceleo Repository*[2]. Furthermore, we have evaluated the usability of ChainTracker in an empirical study with 25 developers completing traceability-driven tasks in two non-trivial *model-driven code generators*, i.e., PhyDSL and ScreenFlow.

1.2 Thesis Contributions

In summary, the contributions of this thesis are three: (C1) a traceability conceptual framework and model-transformation analysis technique to gather fine-grained metamodel-level traceability links from M2M and M2T transformations; (C2) a curated traceability evaluation dataset with 25 ATL (M2M) transformations and 18 Acceleo (M2T) transformations, and two fully-featured *model-driven code generators* in the context of physics-based video games, and mobile graphic user interfaces; and finally, (C3) an integrated end-to-end analysis environment for individual model transformations, and model transformation chains, namely ChainTracker.

C1: A Traceability Framework and Analysis Technique for End-to-end Traceability

Our traceability framework considers model transformations individually, and in transformation chains. We propose a formal *traceability link* definition in the context of M2M and M2T transformations. We formally characterize *traceability links* as dependency relationships derived from the execution semantics of *explicit* and *implicit binding* expressions. Furthermore, we formally introduce *dependent traceability links* and *end-to-end traceability links*, as semantic elements to represent the transitive dependency relationships that arise between the diverse artifacts of a model-transformation chain (Chapter 4).

We present a collection of static-analysis algorithms based on our traceability conceptual framework. They (i) summarize the metamodel-navigation paths used by M2M and M2T binding expressions, and (ii) collect their corresponding fine-grained traceability links. Our analysis technique has been instantiated in the context of ATL and Acceleo M2M and M2T transformations, respectively. However, it can be generalized to other transformation languages as long as they use OCL or OCL-like formalisms as underlying model-manipulation languages. In our evaluation dataset (C2) our analysis technique achieved a fine-grained traceability coverage of 91% and 85% for M2M and M2T transformations, respectively (Chapter 5).

C2: A Traceability Analysis and Usability Evaluation Dataset

The evaluation of our traceability-analysis technique was based on 25 M2M transformations, and 18 M2T transformations. They correspond to 14 individual transformation projects from the *ATLZoo*, and 5 individual code-generation projects from the *Acceleo Example Repository*. We contribute a

[2] `https://github.com/eclipse/acceleo/tree/master/examples`

characterization of the binding expressions and existing traceability links for all the transformations in our evaluation dataset. The characterization process was conducted manually by two software-engineering researchers with with 6 and 2 years of experience in model-transformation technologies (Chapter 5.4).

In order to increase our understanding on the challenges and opportunities of model-driven engineering in the construction of software systems, we have built two fully-featured *model-driven code generators*, i.e., PhyDSL and ScreenFlow (Chapter 3). We have used their underlying transformation ecosystems in the usability evaluation of ChainTracker (C3).

PhyDSL [41, 42] is a game engine and authoring environment for mobile 2D physics-based games. It consists of a textual domain-specific language for gameplay design, and a multi-branched transformation chain that takes high-level gameplay specifications and translates them into executable code for mobile devices. PhyDSL is currently used by the *Faculty of Rehabilitation Medicine at the University of Alberta*, the *Knowledge Media Design Institute at the University of Toronto*, and the *Sapporo Medical University in Japan*, to create cost-effective mobile games for rehabilitation therapy. ScreenFlow is a design environment for mobile application storyboards. It enables developers to quickly translate user-interface sketches into application skeletons, including interface navigation logic. It consists of a textual domain-specific language, and a linear model-transformation chain. ScreenFlow is designed for novice Android application developers and for rapid software prototyping environments, such as hackathons. PhyDSL and ScreenFlow are publicly available in `https://guana.github.io/phydsl/` and `https://guana.github.io/screenflow/`.

C3: The ChainTracker Integrated Traceability Analysis Environment

In order to make traceability information available to model-transformation developers, we present ChainTracker, an integrated traceability analysis environment for M2M and M2T transformations, and transformation chains combining the two. ChainTracker includes interactive traceability visualizations inspired by parallel-coordinate visualizations. Furthermore, it includes projectional code editors that enable developers to explore information obtained from transformation visualizations onto transformation editors, and vice versa. ChainTracker provides developer-oriented features such as binding filters, contextual tables, and transformation highlighters (Chapter 6).

Considering that most developers are used to the execution semantics of imperative programing languages [43], e.g., Java and C++, ChainTracker is designed to lower the cognitive challenges that developers face when first introduced to the declarative semantics of relational transformation languages, e.g., ATL and ETL [44]. ChainTracker's features enable developers to more efficiently complete traceability-driven tasks such as, assessing the impact of metamodel changes, and debug-

ging non-trivial transformation chains. ChainTarcker is currently being released in private beta at `https://guana.github.io/chaintracker/`

Finally, we contribute an empirical study that investigates the performance of developers when reflecting on the execution semantics of M2M and M2T transformations. We measured the accuracy and efficiency of developers when asked to identify dependency relationships between transformation artifacts using ChainTracker. Furthermore, we compared their performance with that of developers using Eclipse Modeling (with ATL and Acceleo plugins) as the industry baseline. In this empirical study, we investigated two research questions:

- **RQ1**: Do developers using ChainTracker identify metamodel and generation dependencies in transformation ecosystems more accurately and efficiently than those using Eclipse Modeling?
- **RQ2**: Do the size and complexity of transformation ecosystems affect the effectiveness of ChainTracker in helping developers identify their metamodel and generation dependencies?

We found that when using Eclipse Modeling most developers could not effectively identify metamodel dependencies defined in non-trivial M2M transformations. Furthermore, we observed that developers were unable to precisely pinpoint dependencies between metamodels and generated textual artifacts in the context of M2T transformations. Our study also revealed that developers often fail to identify chained upstream and downstream metamodel dependencies in both linear and multi-branched model-transformation chains. We found statistically significant evidence that ChainTracker improves the accuracy and efficiency of developers by between 22% and 900% in five families of traceability-driven tasks (Chapter 7).

1.3 Thesis Outline

The content of this thesis is structured as follows. Chapter 2 describes the conceptual foundations of MDE and clarifies the terminology used throughout this thesis. Furthermore, it presents a literature review of the research relevant to our work, including traceability analysis and visualization approaches. Chapter 3 introduces the PhyDSL and ScreenFlow *model-driven code generators*. Chapter 4 presents our formal conceptual framework for end-to-end traceability at the metamodel level. Moreover, Chapter 4 introduces a model-transformation chain running example, which is used throughout the next two chapters of this thesis. Chapter 5 presents our traceability-analysis technique for M2M and M2T transformation, including its corresponding analysis algorithms. Chapter 5 also discusses the evaluation of our analysis technique. Chapter 6 presents ChainTracker and its development-oriented features. Chapter 7 presents the usability evaluation of ChainTracker and highlights its main take home messages. Finally, Chapter 5 revisits our contributions, and summarizes our future avenues of research.

2

Related Work

In this chapter, we discuss the work related to our research. Section 2.1, introduces domain-specific languages, Section 2.2 reviews the fundamentals of MDE. Section 2.6 presets the definition of traceability that our research adheres to. Section 2.7 categorizes the existing work towards collecting traceability information in MDE. Section 2.9 discusses current strategies to visualize traceability in *model-driven code generators*.

2.1 Domain-specific Languages

Domain-specific languages (DSLs) are languages tailored to a specific application domain. [45, 46]. They implement graphical or textual concrete syntaxes that offer substantial gains in expressiveness and ease of use when compared to general-purpose programing languages (GPLs), such as Java or C, in their application domain. Typically, the abstract syntax of a DSL is determined by a metamodel that defines its concepts and their relationships. DSLs provide a simplified development interface, with constructs that simplify –or completely abstract– the execution semantics of an application. According to Fowler [47], DSLs can be classified in three categories; namely *external DSLs, internal DSLs,* and *language workbenches*. An *external DSL* is a language that is completely separate from a general-purpose programming language. It consists of a self-contained abstract and concrete syntax suitable for expressing an application specification from a concern-specific or high-level perspective. An *internal DSL* is a particular way of using a GPL. They use a subset of the language constructs included in a GPL to handle one small aspect of the to-be constructed system. Finally, a language workbench is a specialized integrated development environment (IDE) for defining DSLs. There are two main approaches to build execution engines for DSLs [46]: *translation* (i.e. *generation*) and *interpretation*. The *translation* approach focuses in the transformation of a DSL program into a

language for which an execution engine exists. Typically, this is code in a GPL. In the *interpretation* approach, a customized execution engine is built in order to interpret and execute DSLs programs directly. In this thesis, we focus on *external DSLs* supported by *translational* execution approaches.

2.2 Model-driven Engineering Foundations

Model-driven engineering [48] (MDE) is conceived as an alternative methodology for building software systems in which models are an integral part of the design, construction, and maintenance of an application, during its entire life-cycle. Although MDE can be used to tackle numerous software-engineering tasks, over the last few decades *model-driven code generation* has been the flagship paradigm used to promote its adoption among the general software-engineering community.

In MDE, models capture the structural, functional, and behavioral properties of a software system. They are typically defined using a textual and/or graphical *concrete syntax* associated with a domain-specific language (DSL), and then transformed into executable or deployable artifacts using M2M and M2T transformations. Using MDE techniques, a self-contained domain language can be designed to, for example, describe the security concerns of an application, e.g., authorization mechanisms, and data-access policies. Previously engineered transformations translate such definitions into executable or deployable artifacts such as source-code text, configuration files, and deployment scripts.

It is worth noticing that MDE is surrounded by terminological predicaments. Model-Driven Architecture (MDA) [49, 50] is an OMG standard of an early MDE vision. It prescribes a collection of abstraction layers required in the construction of a *model-driven code generator*. On the other hand, proposals like [51] conceive a more general incarnation of MDE in which models, defined in using formalisms without prescribed levels of abstraction, can be used to completely –or partially– describe a software system. Furthermore, proposals like Model-driven Software Product Lines (MD-SPL) [52] combine MDE concepts, together with the asset management techniques of software product lines, to automate the production of families of software systems.

In this thesis, we focus in the concept of *model-driven code generators* as a general incarnation of MDE principles. A *model-driven code generator* is understood to involve a (number of) domain language(s), used to describe a family of software systems that systematically differ across well-defined dimensions. Moreover, they involve transformation ecosystems, relying on a composition of model transformations, capable of translating system specifications captured using domain-specific languages into executable and/or deployment artifacts, including, but not limited to, source-code, and deployment and testing scripts.

2.3 Model Transformations

Model transformations can be classified in two major types: *model-to-model* and *model-to-text* transformations [53].

Model-to-Model (M2M) Transformations

M2M transformations translate information captured in a source model into a target model. These models can be instances of the same or different metamodels, i.e., endogenous and exogenous transformations [54], respectively. Model transformations can be categorized into three main approaches, namely *direct-manipulation approaches*, *graph-based approaches*, and *relational approaches* [53].

Direct-manipulation approaches are often implemented using object-oriented programming languages, e.g., C++ and Java. They rely on developers encoding transformation algorithms using imperative instructions [3]. An advantage of using a *direct manipulation approach* is that developers generally do not need additional training to write transformations. However, encoding transformation algorithms in imperative languages can be time-consuming, error prone, and the transformation algorithms may be difficult to maintain [3].

Graph-based approaches rely on the theoretical work on graph transformations [55]. A graph-based transformation rule consists of two graphs, a left-hand side (LHS) and a right-hand side (RHS) graph. If a LHS is found in the input model, the rule is triggered, causing the matched sub-graph to be replaced with its corresponding RHS. Concretely, this type of transformations operate on typed, attributed, labeled graphs, which are suitable to represent UML-like models [53]. Transformation technologies that follow this approach include VIATRA [56], PROGRES [57], GReAT [58], AToM3 [59],and Henshin [60]. GReAT provides a visual language to specify a rule's LHS and RHS, and a separate language to describe the rule execution order. Unfortunately, its

non-deterministic nature, and the complexity of its composition constructs, has drastically limited its adoption [3]. Other *graph-based* languages such as Maude [61], are built on logic-based languages in which metamodels are treated as theories, and transformations behave as logic-rewriting rules. Considering that most developers have little experience on logic-based programming languages, its readability and understandability can be limited.

Relational approaches use declarative languages based on mathematical relations and mapping rules. In their most basic form, *relational approaches* use predicates to define correspondence relationships between source and target metamodel elements. Declarative languages can hide the complexity of non-trivial transformation algorithms behind a simple syntax, e.g., algorithms that require depth-first traversals and backtracking strategies in front of deep hierarchical models. *Relational approaches* are often based on the principle of defining *what to do* instead of *how to do it*, which has boosted their popularity among the MDE community. Examples of *relational-transformation approaches* include ATL [38], RubyTL [62], and ETL [63]. Most of the technologies in this category use the Object Constraint Language (OCL) [40], or OCL-like formalisms such as the Epsilon Object Language (EOL) [64], as underlying model-manipulation languages. Even though *relational approaches* are effective at hiding the complexity of non-trivial transformation algorithms, their maintenance and debugging are challenging since visualizing their execution is a not a straight forward process [43, 54].

Finally, transformations may be unidirectional or bidirectional [53, 65]. Declarative transformation rules can be applied in reverse direction in order to obtain the input model used to produce a given output model. This is highly useful in the context of synchronization between models, e.g., system architecture views that can be modified independently. However, since different inputs might produce the same output, a bidirectional transformation might produce multiple solutions. This not only depends on the invertibility of transformation rules, but also on the invertibility of its execution order [53]. Most modern transformation languages do not provide bidirectionality.

Model-to-Text (M2T) Transformations

M2T transformations can be categorized in two main categories: *visitor-based approaches* and *template-based approaches* [53]. *Visitor-based approaches* implement imperative mechanisms to traverse the tree-based internal representation of a model, and generate textual artifacts for each one of its elements [66]. Transformation technologies in this category include Jamda [67], and Melange [68].

Template-based approaches rely on templates that capture the text shared by all instances of the transformation process. Furthermore, templates include binding expressions that inject variable snippets of code using information captured in models. Compared to a visitor-based transformation,

the structure of a template resembles more closely the textual artifacts to be generated [66]. This facilitates their iterative development as they can be derived easily from instances of the to-be-constructed systems. Modern examples of template-based transformation languages include the Epsilon Generation Language (EGL) [63] and Acceleo [39].

Given their popularity and predominant adoption, we focus on *relational and rule-based model-to-model* transformations and *template-based model-to-text* transformations. We use the term *transformation ecosystem* to denote the set of artifacts that comprise one or multiple transformation chains that work cooperatively in a model-driven software-engineering tool [10].

2.4 Model-Transformation Compositions

Transformations can be composed using *internal* or *external composition* strategies to accomplish complex software-engineering tasks [4, 69]. In a model-transformation composition, each transformation encapsulates a set of binding expressions that deal with a specific step of the transformation process. *External composition* allows the integration of transformations developed using (potentially) multiple transformation languages. They are usually specified in a pipeline architecture, where the output of a transformation serves as the input for the next one, resulting in a "chain" of *model-to-model* and/or *model-to-text* transformations [70]. *Internal composition* is characterized by the "compilation" of multiple transformation rules into a single transformation unit, which is often implemented using a single transformation language [69, 71]. From the developer's perspective, using an *external composition* strategy implies that the execution of a transformation step is semantically isolated from other steps in the transformation chain.

A basic activity of the software design process is the modularization of the software specification into a collection of modules that, together, satisfy a set of functional and non-functional requirements [72]. The theoretical criteria for software modularization include increased software cohesion, reduced coupling, and information hiding [73, 74]. In *model-driven code generators*, two composition strategies support transformation modularity. *Vertical modularization* is used to gradually reduce the semantic gap between a system specification, and its corresponding executable artifacts. Thus, intermediate metamodels and transformations focus on the separation of different abstraction layers that isolate the high-level specifications from implementation-specific constructs. *Horizontal modularization* is used to isolate the a collection of language concepts in individual

language implementations. In this context, DSLs can be modularized to describe different system concerns, architectural views, and application abstraction levels. Thus, individual transformation ecosystems can be developed to target smaller languages in a self-contained fashion. This is the basis for meta-model extensibility through generator extensibility [4].

As a concrete example, a domain-specific language can be defined to capture the security policies of a web application. These policies might be sufficiently concise so its corresponding *model-driven code generator* can be implemented using a single M2T transformation. However, in the case of a domain-specific language for the construction of video games, its potentially broad semantics can be divided into sub-domains. This modularizes its corresponding *model-driven code generator* into multiple transformation branches, e.g., the artificial intelligence of its gameplay entities, their corresponding control mechanics, as well as their graphical rendering properties.

We focus on *model-driven code generators* implemented using externally-composed transformations, modularized using both vertical and horizontal modularization strategies (Figure 2.1).

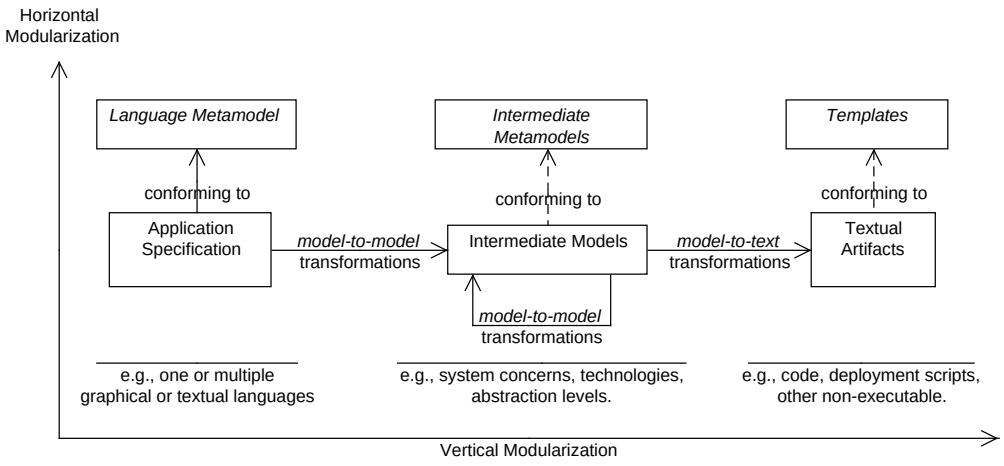


Figure 2.1: Model-driven Code Generator - Transformation Chain

2.5 Model-driven Code Generator Life-Cycle

The role of an *environment developer* is to design, build and maintain a DSL and its underlying transformation ecosystem. This category of developers includes requirements engineers, domain analysts, and other software engineers managing artifacts such as, context-free grammars, model transformations, and code templates, as well as the generator's underlying development environ-

ments i.e., graphical and syntax-directed editors. In turn, the role of an *application developer* is to utilize a domain-specific language, and its underlying transformation engine, to create an application specification and derive its corresponding executable artifacts. *Application developers* may be able to augment or maintain a generated application instance. In this case, a *model-driven code generator* is used as a one-time bootstrapping tool [75], rather than a continuous development environment. Figure 2.2 summarizes the activities that *environment developers* and *application developers* perform during the life-cycle of a *model-driven code generator*.

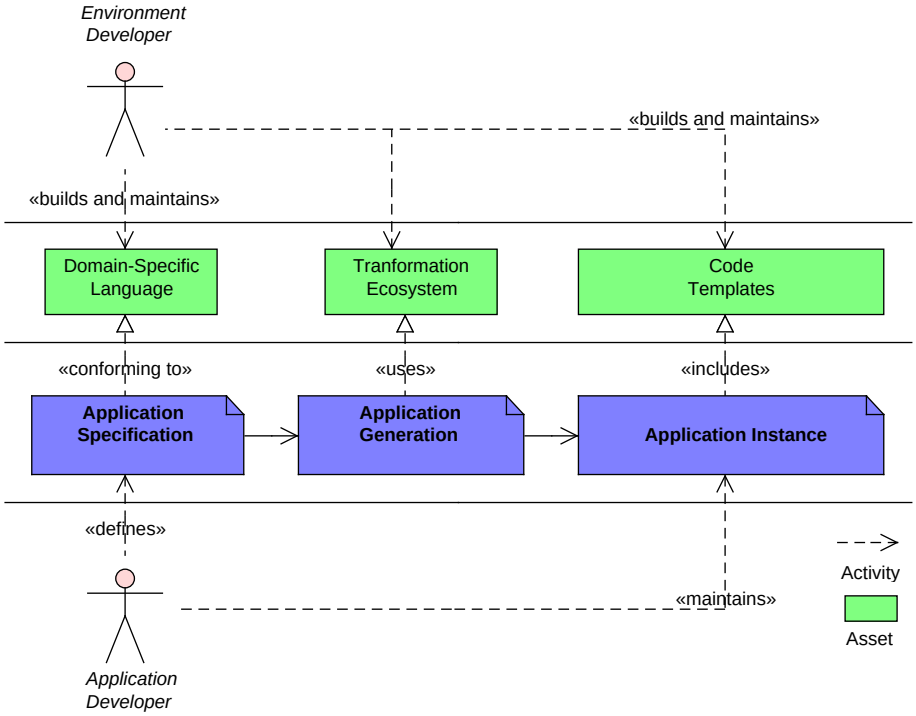


Figure 2.2: Model-driven Code Generator - Developer Activities

In this thesis, we focus on supporting *environment developers* answering traceability driven questions while completing construction and maintenance activities in *model-driven code generators*.

2.6 Traceability

Software traceability is understood as the ability to relate heterogeneous development artifacts created during the construction and maintenance of a software system, and used to describe a system from different perspectives and levels of abstraction [76, 77]. These artifacts are typically created

by potentially multiple stakeholders to capture functional requirements, bug reports, deployment metrics, quality attributes, graphic user interface designs, and codebases, among many others, [76]. According to Ramesh et al. [78], traceability relations capture *overlap, dependency, evolution, satisfiability, conflict, generalization* and *refinement* associations between diverse software artifacts. Maintaining traceability links between software artifacts improves software maintainability, and helps developers assessing the impact of changes in front of evolution [79]. Unfortunately, manually maintaining traceability links among multiple, and individually evolving, artifacts is extremely impractical [79]. Numerous automatic traceability approaches have been proposed to address this issue in the requirements engineering field of study, e.g. *recovering traceability links between code and documentation* [80, 81, 82, 83, 84], *detection of logical coupling based on release history* [85], *recovering traceability links as software artifacts evolve in repositories* [86, 87].

Classic traceability analysis approaches have to analyse semantically disjunct artifacts in order to identify their relationships. Thus, most of them are based on information retrieval (IR) techniques [76, 88, 89]. Conversely, the artifacts involved in the construction of a model-driven software system are captured in models described using well-formed languages (DSLs). Furthermore, these models are explicitly linked by means of transformations with the purpose of isolating the levels of abstraction and the views of a system while, at the same time, enabling their automatic synchronization and generation. Traceability analysis approaches in MDE are mostly concerned about understanding the execution semantics of transformation codebases and the artifact relationships they capture. In this context, a variety of traceability-analysis techniques have been proposed to support the development and maintenance of *model-driven code generators*. These include, but are not limited to, assessing the impact of metamodel evolution [15, 16, 20], testing transformations [90, 91, 22, 30, 92], optimizing transformations to improve their qualities [93, 94, 95], and visualizing transformation execution [96, 97, 21, 98].

Unfortunately, the term *traceability* is not consistently used by authors in the MDE community [17]. We elaborate on the traceability definition proposed by Winkler et al. [17] in which traceability is conceived as "*the ability* to collect traceability links from a set of transformation-binding expressions." Traceability links are understood as the *dependency relations* between a set of artifacts in a transformation ecosystem, including a transformation's codebase, its source and target metamodels, corresponding model instances, and potential generated artifacts. Moreover, a traceability link is understood as a "dependency relationship between two artifacts *a1* and *a2*, in which *a2* relies on the existence of *a1*, or that changes in *a1* potentially result in changes in *a2*" [17, 99, 100].

2.7 Model-to-Model Traceability Analysis

Given the loose definition of *traceability* in the field of MDE, categorizing the existing work on traceability analysis is a challenging task. Authors such as in [27, 28, 29, 30, 31] have proposed traceability-analysis techniques for M2M transformations. They conceive traceability links as dependency relationships between the models used by a transformation and those produced after its execution, i.e., *traceability at the model level*. Yet other researchers, such as in [20, 21, 33], conceive traceability links as the symbolic dependencies between the metamodels used by a transformation, and its corresponding binding expressions, i.e., *traceability at the metamodel level*. Most of the aforementioned techniques identify dependency relationships by means of instrumenting and executing the transformations under analysis, thereby obtaining traceability information as a byproduct of a transformation execution itself. Other techniques, however, identify dependency relationships by comparing the input and output models of a transformation in order to infer its execution mechanics. Traceability techniques such as in [101, 102, 103, 104] investigate how to collect traceability links in M2T transformations. They conceive traceability links as dependency relationships between metamodels, code templates, and generated textual artifacts.

Another dimension where one can distinguish traceability techniques is in terms of their level of granularity. Analysis techniques such as in [27, 28, 98], consider traceability links as dependency relationships between the metamodel elements of an ecosystem, i.e., *coarse-grained traceability*. Other techniques, such as in [105, 20, 20, 104, 29] include their constituent attributes, in addition to their top-tier elements, i.e., *fine-grained traceability*. To the best of our knowledge, there are no proposals that provide a unified traceability analysis technique for heterogeneous transformation compositions, i.e., transformation chains that combine both M2M and M2T transformations. None of the existing techniques provide analysis capabilities to identify end-to-end fine-grained traceability links.

The execution of a model transformation is defined using binding expressions. A binding expression consists of one (or multiple) individual binding statement(s) that derive a target attribute, based on potentially multiple source attributes. In our research, we categorize bindings in two main types, namely *explicit bindings*, and *implicit bindings* [36, 106]. *Implicit bindings* manipulate, constrain, and navigate the structure of a metamodel in order to realize the intent of a transformation. *Explicit bindings* effectively assign the value of a source attribute into a target attribute. Traceability links represent dependency relationships given by the execution semantics of *explicit* and *implicit* bindings. In Chapter 4, we formally define these concepts. Figure 2.3 portrays an example of *explicit* and *implicit bindings* in the context of UML2Java [107], a popular M2M transformation from the *ATLZoo*. In this case, the Operation2Method (O2M) transformation rule contains a binding expression with multiple binding statements. These statements define dependency relationships

between the *(return) type* of a Java Method, and multiple elements (and attributes) of the UML metamodel, including the *parameters* of an Operation and its corresponding *type* and *kind* attributes.

```
rule O2M {
   from e : UML!Operation
   to out : JAVA!Method (
        ...
        type <- e.parameter->
                select(x|x.kind=#pdk_return)->
                asSequence()->first().type,
        ...
   )
}
```

(implicit bindings) (explicit binding)

Figure 2.3: Operation2Method (O2M) - Implicit and Explicit Binding Example

Collecting complete and fine-grained traceability information is a fundamental requirement for the construction of pragmatic model-driven software engineering tools. However, to the best of our knowledge, current traceability-analysis techniques do not consider *implicit bindings* in their analysis. Table 2.1 summarizes the traceability-analysis techniques most relevant to our research. Our summary classifies the techniques according to the type of transformations they analyze, the level of abstraction and granularity of their traceability links, and whether they have been included in developer-oriented tools. Table 2.1 also includes information on whether their evaluation is based on non-trivial transformations, or simple conceptual examples. Moreover, Table 2.1 presents whether the reviewed techniques propose traceability-visualization strategies[1] (Section 2.9). Let us now briefly discuss the reviewed traceability-analysis techniques.

2.7.1 Metamodel-Level Traceability Analysis

Van Amstel et al. [21] present a traceability-analysis technique for ATL (M2M) transformations. It conceives traceability links as dependency relationships between the binding expressions of a transformation, and its corresponding source and target metamodels. It analyzes the abstract-syntax tree of individual transformations in order to collect coarse-grained traceability links. It does not consider *implicit bindings* as a part of its analysis. This work is evaluated using a research case study in the context of concurrency management systems. In [98], Van Amstel et al. present an extended version of their work with the purpose of collecting traceability information at the model level of abstraction. The latter version is in turn evaluated using a minimal pedagogical example. Van Amstel et al. export traceability information in a textual file that can be interpreted by TraceVis [113], a generic tool proposed to visualize the interactions of software artifacts.

In [20], Di Rocco et al., propose a traceability-analysis framework for M2M transformations. This technique uses weaving models [114] in order to specify and manipulate correspondences between evolving transformation artifacts. It identifies fine-grained traceability links at the meta-

[1](GB) Graph Based, (CR) Cross-Reference Based, (MB) Matrix Based

Table 2.1: Traceability Analysis and Visualization Techniques in Model-driven Engineering

		Transformation Type			Granularity		Evaluation		Developer Support	
	Technique	**M2M**	**M2T**	**MTC**	**Coarse**	**Fine**	**Non-Trivial**	**Conceptual**	**Tooling**	**Visualization**
Meta Level	Van Amstel et al. [21]	ATL		■	■		■	■		GB
	Di Rocco et al. [20]	ATL				■		■	■	GB
	Di Ruscio et al. [33]	ATL				■		■		
	Olsen et al. [103]		MOFScript			■		■	■	CR
	Glitia et al. [105]	Gaspard		■		■				
	Guana et al. [35, 36, 37, 10]	ATL	Acceleo	■	■	■	■		■	GB
Model Level	Drivalos et al. [108]	NA	NA		■		■			
	Falleri et al. [27]	Kermeta		■	■			■		GB
	Van Amstel et al. [98]	ATL		■	■			■		GB
	Von Pilgrim et al. [29]	ATL, MTF		■		■		■	■	GB
	Santiago et al. [31]	ATL				■	■		■	GB
	Jouault [28]	ATL			■			■		
	Matragkas et al. [30]	ETL			■			■	■	CR
	Kolovos et al. [109]	NA	NA		■			■	■	CR
	Grammel et al. [32]	ATL				■	■			
	Santiago et al. [104]	ATL			■				■	MB
	Oldevik et al. [101]		MOFScript			■		■		
	Grammel et al. [110]	QVTo	Xpand			■		■		
	Garcia et al. [102]		MOFScript			■	■		■	CR
	ATL Eclipse Plugin [111]	ATL			■	■			■	CR
	Acceleo Eclipse Plugin [112]		Acceleo		■	■			■	CR

model level. However, it does not identify traceability links from *implicit bindings*. Similarly to Van Amstel et al. [21], Di Rocco et al. export traceability information to be visualized with TraceVis. This framework has been evaluated using the *PetriNet2PNML* example from the *ATLZoo*. In [115], Di Rocco's et al. include their traceability analysis framework in MDEForge, a community-based modeling environment.

Di Ruscio et al. [33] present a methodology to build textual editors for metamodel definitions. It includes a technique to support the propagation of changes between metamodels and their corresponding textual representations. In order to support their change-propagation mechanism, Di Ruscio et al. propose a metamodel-level traceability-analysis technique. It uses the TCS domain-specific language [116] to define mappings between metamodels and their corresponding syntactical elements. This technique is comprised of three main steps: (a) identifying the dependencies between a given metamodel and its concrete syntax definition, (b) classifying the identified changes according to their impact, and (c) defining syntax adaptations to restore its consistency. Di Ruscio's work uses the *PetriNet2PNML* example from *ATLZoo* as a running example.

In [105], Glita et al. present a conceptual traceability framework for the Gaspard Modeling Environment [117]. The analysis framework conceives traceability links as fine-grained metamodel dependencies in M2M transformations. Although the authors argue that they have included traceabil-

ity capabilities in the Gaspard transformation engine, details of its implementation are not formally introduced.

In [108], Drivalos et al. present the Traceability Metamodeling Language (TML), a language designed to manually describe metamodel-level coarse-grained traceability links. Similarly to Di Ruscio et al. [33], this technique is limited by the ability of developers to manually define traceability constructs. This is a time consuming and error prone task. The expressibility of the language is evaluated a using the i* and the KAOS ecosystems [118] from the requirements engineering domain.

2.7.2 Model-Level Traceability Analysis

Jouault [28] presents a strategy to collect model-level coarse-grained traceability links from ATL (M2M) transformations. This technique relies on developers manually augmenting model transformations, in order to derive traceability information as a byproduct of its execution. However, given that transformations are polluted with additional binding expressions, the resulting codebases are harder to build and maintain [105]. This technique is one of the first traceability-analysis approaches in the MDE literature, and its evaluation is limited to conceptual examples.

Similarly to Jouault [28], Falleri, et al. [27] propose a traceability-analysis framework that relies on developers augmenting transformations to obtain model-level traceability links. The proposed mechanism is comprised of an imperative language designed to augment M2M transformations, and a metamodel designed to capture traceability information. This framework is targeted at M2M transformations defined in Kermeta [97], a transformation language developed by the same authors. Falleri, et al. use GraphViz [119] to graphically represent traceability information in model-transformation chains. The usability of the framework is discussed in terms of a simplified database translation example.

In [29], Von Pilgrim et al. present a traceability visualization framework for ATL and MTF model-transformation languages. In [120], Von Pilgrim et al. introduce $_{UNI}TI$, an Eclipse plugin to visualize model-level fine-grained traceability links. Similar to Jouault [28] and Falleri [27], this technique relies on developers augmenting transformations with expressions that derive traceability information at runtime. This technique does not consider *implicit bindings*. The visualization framework is evaluated using the Class2ER transformation from the *ATLZoo*.

In [109], Kolovos et al. present the Epsilon Merging Language (EML). The EML can be used as a mechanism to gather model-level traceability from M2M transformations. This technique includes a family of metamodel stereotypes, and a collection of transformations that compare metamodels with stereotyped elements. The evaluation of this work is based on UML models that have been manually stereotyped to make their traceability analysis possible. More recently, Grammel et al. [32] present a similar model-matching technique to collect model-level fine-rained traceability

links in ATL. The evaluation of this work includes large-scale business transformations, as well as transformations from the ATLZoo.

In [31], Santiago et al. introduce iTrace, a model-level traceability-analysis environment for ATL. iTrace follows a transformation augmentation strategy similar to Falleri's in [27]. In this case, the augmentation process is supported by High Order Transformations (HOT) based on transformation-rule signatures. In effect, this technique does not consider *implicit binding* expressions. In [104], Santiago et al. present an extension of their traceability visualizations for M2T transformations. However, a M2T traceability-analysis technique is not formally introduced.

Matragkas et al. [30] preset a traceability-driven approach for M2T transformation verification. The authors propose an extension of the Epsilon Transformation Language (ETL) [44] to automatically collect model-level traceability information. Unfortunately, Matragkas et al. do not provide details on their traceability-analysis technique. Most of their work focuses on the definition of traceability contracts, and an Eclipse plugin that notifies developers when contracts are violated.

Grammel et al. [110] propose a traceability framework that enables developers to manually extend transformation engines with traceability-analysis capabilities. The framework includes Trace-DSL, a domain-specific language to capture traceability in a well-formed fashion, and a programming interface with traceability-analysis services. Trace-DSL considers traceability links at the model level. It conceives four types of traceability links: *create links, update links, delete links,* and *query links*. In [110], two traceability implementations are presented as a proof-of-concept of the framework, including an interface for the Xpand[2] M2M transformation language, and an interface for the QVT-o[3] M2T transformation language. The Xpand implementation collects fine-grained traceability, while the QVT-o implementation collects coarse-grained traceability.

2.8 Model-to-Text Traceability Analysis

Olsen et al. [103] introduce a traceability-analysis technique for MOFScript M2T transformations. It uses template annotations to define traceable segments of code. It considers traceability links as generation dependencies between generated text files and metamodel properties. This technique only considers *explicit bindings* when analyzing traceability information. Olsen et al. present an Eclipse plugin to explore M2T traceability links in a tabular fashion.

In [102], Garcia et al. describe HandyMOF, a web-based testing facility for M2T transformations. HandyMOF measures the coverage of transformation test suites by analyzing the binding expressions they execute. HandyMOF uses an analysis technique based on annotations similar to Olsen et al. [103]. It conceives traceability links as dependency relationships between metamodels,

[2] `http://www.eclipse.org/modeling/m2t/?project=xpand`
[3] `http://wiki.eclipse.org/QVTo`

transformation codebases, and generated files. HandyMOF's evaluation is based on a web-based application for geolocation services. This technique only considers traceability analysis from *explicit binding* expressions.

In summary, current traceability-analysis techniques do not consider *implicit bindings* when collecting traceability information. With the exception of Grammel et al. [32], their evaluation is often based on a single transformation example, thus their practical limitations are not thoroughly explored. Most of the analysis techniques describe domain-specific languages to capture traceability links in a well-formed fashion. However, only few of them present a detailed description of their traceability-analysis process. Furthermore, current analysis techniques do not consider M2M and M2T transformation as a part of a unified model-driven engineering toolbox. This effectively limits their usability in the context heterogeneous model-transformation chains, and *model-driven code generators*.

2.9 Traceability Visualization

Multiple techniques have been proposed to diagrammatically depict traceability information in software systems. Most of these techniques have been developed in the field of requirements engineering [17]. They have inspired little, but precious work on visualizing traceability information in the context of MDE. According to Wieringa [121], traceability visualizations can be categorized in three main groups: matrices (MB), cross-references (CR), and graph-based representations (GB) (Table 2.1). Let us briefly discuss each one of them, and provide examples of traceability visualizations that follow their design guidelines.

2.9.1 Matrix Representations

Traceability matrices portray traceability links between a two-dimensional set of software artifacts. They follow a grid-based layout in which rows and columns capture information about two families of related entities. Primitive traceability matrices represent the existence of a dependency relationship between two artifacts by placing a mark, such as a black box, in their corresponding intersecting cell [17]. Almeida et al. in [96] use a matrix-based representation to study the conformance relationships between the implementation of a M2M transformation and its application domain (Figure 2.4). Traceability matrices provide developers with little information about the type of relationship that a traceability link represents. However, enhancements can be made to matrices in order to enrich the information that they convey [121]. For example, matrices can be made interactive as to allow navigation to specific linked artifacts, such as using pop ups [122] or color encoded properties [123].

	M1	a_{TSA}	M2	P3	M3
AR1	✓		[✓] TSA		[✓]TSB
AR2	✓		[✓] TSA		[✓]TSB
AR3	✓		[✓] TSA		[✓]TSB
AR4	✓		[✓] TSA		[✓]TSB
AR5	✓		[✓] TSA		[✓]TSB
AR6	✓		[✓] TSA		[✓]TSB
AR7		✓	[✓] a_{TSA}		[✓]TSB
AR8		✓	[✓] a_{TSA}		[✓]TSB
AR9		✓	[✓] a_{TSA}		[✓]TSB
AR10		✓	[✓] a_{TSA}		[✓]TSB
AR11				✓	✓
AR12				✓	✓

Figure 2.4: Traceability cross-table used in [96] to relate the models, application requirements, and transformations scripts of an ecosystem.

Traceability matrices are easy to understand by expert and novice developers. However, they have several limitations when used to represent the traceability links in a transformation chain. A matrix representing the symbolic dependencies between the source and target metamodels of a transformation can be extremely cluttered and overwhelmingly large. Furthermore, the size of such

a traceability matrix will depend on the size of each underlying metamodel, and the complexity of the transformations under analysis. Research has shown that large traceability matrices become unreadable very quickly [124, 125], and that their two-dimensional nature makes them unsuitable to represent n-ary traceability links, or links between hierarchical artifacts [17]. This is a common scenario in the context of transformation ecosystems, in which artifacts such as metamodels and transformations have hierarchal structures.

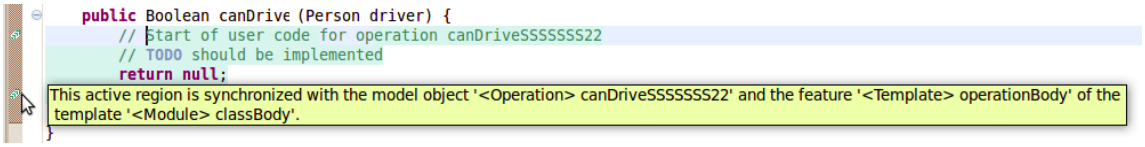


Figure 2.5: Aceeleo Eclipse Plug-in: In-line Cross-referencing Editor [112]

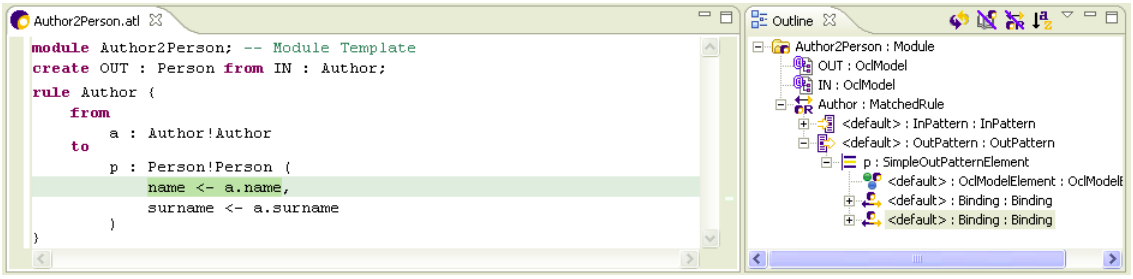


Figure 2.6: ATL Eclipse Plug-in: Cross-referencing Multi-panel Editor [111]

2.9.2 Cross-reference Representations

Traceability links can be expressed as cross-references embedded in the artifacts of a software system [126]. Cross-references can be represented using natural language, or using interactive referencing features such as hyperlinks. In their most simple from, cross-references can be found as in-line documentation in source code and design documents. In the context of rule-based transformation languages such as ATL, RubyTL and ETL, it is a widespread practice to document the source code of every transformation rule with cross-referencing notes, e.g., *"This rule transforms element A into element B"* or *"This rule uses helper X."*

Hyperlinks enable developers to navigate through the traceability links of a given artifact, in order to switch between their different contexts. A limitation of this approach is that hyperlinks can only reveal localized outgoing and upcoming traces between two artifacts [17]. Cross-referencing is very common in modern integrated development environments. For example, the Eclipse plug-ins for transformation technologies, such as ATL and Acceleo, allow developers to use interactive

code editors, and navigate through their execution dependencies via cross-referencing hyperlinks. Developers can click on the procedural calls between transformation rules in order to have access to their definition from outlying segments of code (Figures 2.5 and 2.6). Similarly, techniques such as in [103, 30, 102] enable developers to explore detailed information about individual metamodel elements, by means of hyperlinks that open views with detailed listings describing their relationships with other transformation artifacts.

Even though cross-reference representations allow the navigation of interdependent traceability links, they do so at the cost of limiting the visible scope to one single artifact at a time. This makes cross-referencing a poor alternative to portray global dependency views between artifacts in model-transformation chains. Furthermore, using cross reference representations to visualize n-ary links is highly impractical [17]. Representing n-ary traceability links is a fundamental requirement in model-transformation ecosystems. Complex ecosystems usually involve multiple fine-grained artifacts with multiple outgoing and upcoming dependency relationships. As a concrete example, consider the dependency relationships between a M2T transformation and a generated segment of code: a metamodel element can be used in the generation of multiple lines of code, and a line of code may be the result of querying multiple metamodel elements in a single binding expression [102].

2.9.3 Graph-based Representations

Most artifacts in model-driven software engineering tools are represented using both graphical and textual concrete syntax, e.g., a metamodel can be studied in its textual structured form, or as a class diagram that captures its elements and properties. The dual nature of artifacts in transformation ecosystems makes diagrams and general graph-based representations the most common mechanism to represent their traceability information [17]. Let us now review current graph-based approaches to represent traceability information in transformation ecosystems.

Falleri et al. [27] represent traceability information in Kermeta as a bipartite graph in which nodes represent individual model elements, and edges their dependency relationships. Falleri et al. use Graphviz [119] in order to create a simple visual representation of their trace graph. It is important to mention that due to the static nature of the visualization, no additional information can be obtained by means of interacting with it.

In [29], Von Pilgrim et al. present a traceability-visualization strategy based one GEF3D [120]. This strategy visualizes a collection of overlapped 2D class diagrams linked by edges in a 3D space (Figure 2.7). Each layer of the visualization captures a diagram corresponding to the models resulting from the execution of each step of a transformation chain. The 3D visualizations presented in this tool have numerous scalability issues. Considering that model instances may contain several elements, handling the visualization of large class diagrams is challenging in terms of memory

space [127]. Furthermore, in terms of usability, research has shown that large class diagrams pose significant cognitive challenges to developers when filtering, isolating, and summarizing information [128, 129], which is exacerbated by the 3D overlapping nature of the proposal.

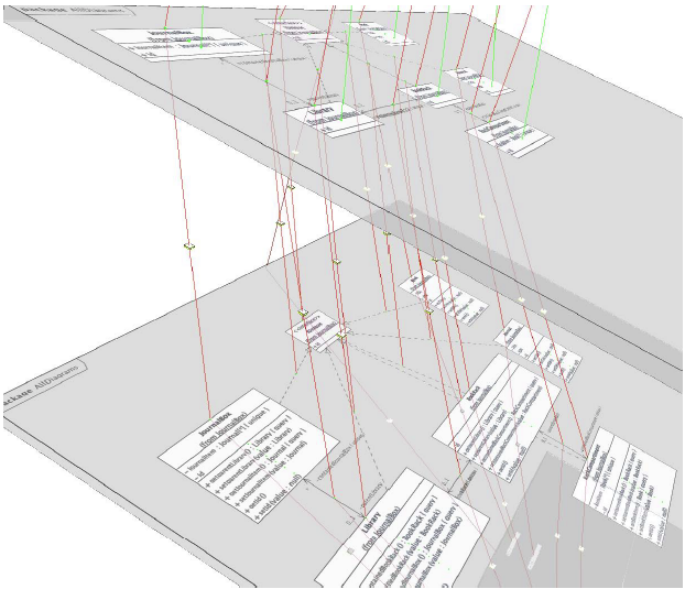


Figure 2.7: Traceability 3D visualization created using GEF3D [29]

Van Amstel et al. [21, 98] use TraceVis [130] to visualize traceability information in ATL (M2M) transformations. The visualization highlights the hierarchical structure of the transformation expressions that determine the presence of symbolic dependencies between source and target metamodel elements, e.g., grouping them in *helpers, matched rules, lazy matched rules, unique lazy matched rules*, and *called rules*. TraceVis supports hierarchical edge bundling which makes both tools highly scalable in front of large ecosystems [131]. A similar visualization approach based on TaceVis is presented by Di Rocco et al. in [20].

Santiago et al. introduce iTrace [31], a framework for the management and analysis of traceability information in MDE. iTrace offers two visualization dashboards to the end user, namely the overview dashboard, and workload dashboard. The overview dashboard presents a tabular view that summarizes the metamodel elements used by a transformation. The workload dashboard presents information about a transformation's runtime behavior, including the number of elements processed by each of its transformation rules. In [104], iTrace was extended to support the visualization of M2T transformations using a multi-panel editor. The editor includes information such as the model elements used by a M2T transformation, and the textual artifacts derived from its execution. Unfortunately, iTrace has not been designed to support the visualization of model-transformation chains; it considers traceability links in M2M and M2T separately, and in different levels of abstrac-

tion. Furthermore, iTrace portrays traceability links using interactive two-dimensional tables, thus suffering from the limitations of matrix-based representations discussed in Section 2.9.1.

None of the reviewed techniques proposes a traceability visualization for M2T transformations as a part of a model-transformation chain. More importantly, even though most proposals claim that their traceability collection and visualization techniques help developers to build and maintain transformation ecosystems in a more effective or efficient fashion, none of them has been empirically validated in controlled experiments with real developers. Moreover, none of the tools reviewed in this section is publicly available for download to be studied or compared by other research teams. To the best of our knowledge, ChainTracker (Chapter 6) is the first traceability collection and visualization technique to be formally evaluated with real developers using non-trivial case studies.

3

PhyDSL and ScreenFlow

In this chapter, we present PhyDSL and ScreenFlow, two model-driven code generators in the context of video game and mobile application development, respectively. The motivation behind the construction of PhyDSL and ScreenFlow has been to increase our understanding on the challenges and opportunities of model-driven engineering in the construction of complex software systems. We have used their underlying transformation ecosystems in usability evaluation of ChainTracker (Chapter 7).

3.1 PhyDSL

PhyDSL [41, 42] is a game engine and authoring environment for mobile 2D physics-based games. It consists of a textual domain-specific language for gameplay design and a multi-branched transformation chain that takes high-level gameplay specifications and translates them into executable code for mobile (Android) devices. PhyDSL's transformation chain includes four M2M transformations implemented using ATL, and four template-based M2T transformations written in Acceleo (Figure 3.1). PhyDSL is currently used by the Faculty of Rehabilitation Medicine at the University of Alberta, the Knowledge Media Design Institute at the University of Toronto, and the Sapporo Medical University in Japan, to create cost-effective mobile games for rehabilitation therapy [132, 133]. The PhyDSL official website can be found at `https://guana.github.io/phydsl/`.

As mobile devices become intrinsic to people's everyday lives, so does mobile gaming [134]. Currently, mobile gaming accounts for 42% of the gaming market, with an estimated total revenue of $46 billion [135]. The challenges of mobile-game development include, short times-to-market, deploying in evolving platforms, and answering to a heterogeneous population of game consumers [136]. These challenges are exacerbated by the diversity of the video-game development teams,

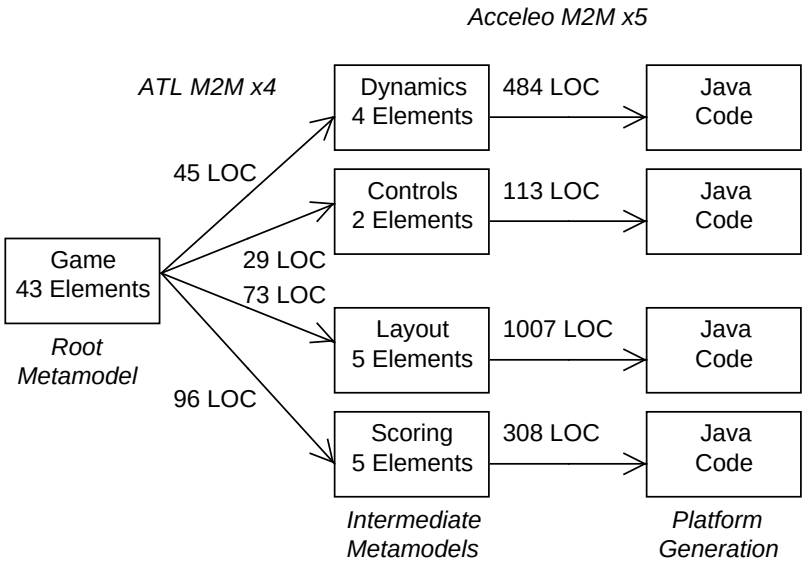


Figure 3.1: PhyDSL's multi-branch model-transformation chain: 4 (M2M) ATL transformations comprised of 14 matched rules and 2 helpers, and 4 (M2T) Acceleo transformations containing 172 binding expressions.

and the highly iterative nature of video-game design, i.e., several gameplay designs have to be prototyped and evaluated before deciding which one to implement fully.

Video games are designed by teams with diverse skills. These teams include storytellers, artists, graphic designers, and software engineers [136]. However, more often than not, video-game authoring environments are designed for software developers, supporting programming tasks agnostic of the needs of the non-computer experts on the team. Callele et al. [137] identify two main stages of video-game development: *pre-production* and *production*: in the former stage developers reduce design uncertainty by defining the game story, characters, and visual effects; the latter stage involves the formalization of the game's requirements, architecture, and implementation details. Furthermore, considering the roles of the two major groups participating in both of these stages, *game designers* produce the documents that contain a game concept and gameplay structure, and the *software engineers* formalize the game requirements and implement the designers' vision. According to Callele et al., the transition process between the *pre-production* and *production* stages is challenging due to the absence of generally accepted practices that facilitate the interaction between the two teams; *game designers* do not necessarily understand the limitations of the implementation technologies and *software engineers* often limit the designers' creative vision. Acknowledging these challenges, Tang and Hanneghan [138] highlight the need of game-authoring environments to facilitate the rapid prototyping of games by non-computer experts.

Video game development is a fundamentally iterative process. In the early stages of video game prototyping, variable versions of a game need to be implemented and analyzed in order to improve its overall design [139]. Among many others, perceived gameplay difficulty, visual aesthetics, actor mechanics, and goal cohesion, are some gameplay features that need to be experimented with in order to shape games to their target audiences [140].

In [141], Johnson-Laird presented mental models as the main concept behind human reasoning. Johnson-Laird argued that mental models are key to capturing human perception, imagination, and structural understanding of reality. In the context of video-game development, game developers implement gameplay designs using mostly general-purpose programming languages. In this process, game developers have to translate their gameplay mental models into the operational semantics of a programming language. This task is cognitively challenging and often frustrating for developers, since general programming languages are not designed to capture the gameplay mechanics in the developers' vision of the game.

The semantic gap between the developers' mental model of a game and the implementation artifacts that make this model executable, poses significant challenges to non-programming experts when reflecting about a gameplay design. In the context of video-game design, Tang et al. [142] studied the benefits of using model-driven engineering techniques as a means to abstract the implementation details of video games, and allow non-programming experts to prototype, and efficiently create gameplay designs.

In our work, we have focused on physics-based games that represent a large segment of the casual-games and rehabilitation-games markets, including *platform*, *shoot 'em up*, *puzzle* and *maze* games. This segment includes popular titles such as Angry Birds[1], and platformers such as Rayman Fiesta Run[2]. We have proposed PhyDSL as a model-driven prototyping environment for this broad class of games. PhyDSL enables non-programming experts to sketch out gameplay designs using high-level textual language, including their visual layout, interaction alternatives, and feedback mechanisms, and to automatically obtain their corresponding executable codebase based on an architecture designed for reuse. PhyDSL enables developers to quickly perform design changes, and to create alternative gameplay designs for agile gameplay testing.

[1] https://www.angrybirds.com/

[2] https://www.ubisoft.com/en-us/game/rayman-fiesta-run

3.1.1 Background

Although numerous tools exist to support the flexible prototyping of video games, relatively few academic articles have been published describing their technical implementation, or evaluating their ease of use. Let us now review some of them.

Furtado and Santos [143] present SharpLudus, a code-generation environment for stand-alone action-adventure games. SharpLudus includes two graphic domain-specific languages to model game layouts and character behaviors, and a generation engine targeted at C# built-on Microsoft's DirectX. SharpLudus is specifically tailored for the creation of top-down room-based gameplay designs. It enables developers to specify characters with health counters, and non-playable characters (NPCs) with primitive artificial intelligence (AI) behaviors. Although it is not clear whether SharpLudus implements model-driven engineering techniques, it gives developers access to the generated codebases for post-generation edition and refinement.

Reyno et al. [144] present a game-authoring prototype based on model-driven engineering techniques. The tool produces C++ code from UML models extended with game-specific stereotypes. It allows developers to define the structure and behavior of standalone platformers. Although concrete examples of game specifications are provided, the underlying language is not described in detail. More recently, Palmer [145] introduced Fictitious, a domain-specific language to specify textual interactive stories in which players make decisions by navigating through a world with immersive narratives.

In [146], Robenalt proposed a model-driven engineering framework to develop multiplatform 3D games. The proposal envisions the usage of the Eclipse Modeling Framework (EMF) together with model-transformation technologies, in order to describe platform-independent models that capture the geometry, textures, lighting, animations, and sounds of a game. These high-level specifications provide the basis for automatically generating code targeted at 3D game engines. Karamanos et al. [147] describe a 2D graphical authoring environment to create 3D action role-playing games. It enables developers with limited technical background to specify 3D spaces using 2D projections. The environment allows game developers to specify the location and attributes of the gameplay elements. It creates a rendering client capable of materializing the high-level definitions using the OpenGL-ES API.

Finally, proprietary authoring environments such as GameSalad[3], GameMaker[4], Stencyl[5], and Construct2[6] provide multi-platform game construction mechanisms integrated with visual editors that enable the creation of video games through drag-and-drop operations, action events, and

[3] http://gamesalad.com/
[4] https://www.yoyogames.com/studio
[5] http://www.stencyl.com/
[6] https://www.scirra.com/construct2

advanced scripting. Although proprietary environments are highly flexible, and provide powerful generation capabilities, they often present developers with a steep learning curve. More often than not, proprietary environments do not allow developers to have access to the code that they produce.

PhyDSL enables developers to understand dynamic gameplay interactions that otherwise are not easily represented using graphical syntaxes. This includes collision events and corresponding event actions, time-based activities, physical interactions, and game control mechanics. PhyDSL combines a succinct textual syntax with declarative constructs that are easy to understand, along with a graphical interpreter that analyzes gameplay specifications, and diagrammatically portrays their visual layout. This helps developers to quickly envision the organization of the graphical assets of a game while, at the same time, interpreting its dynamic properties without advanced training. Moreover, PhyDSL enables developers to access generated codebases, so prototypes can be used as boilerplates for further development.

3.1.2 The PhyDSL Language

Considering the three major design components for gameplay specification presented by Hunicke et al. in [148], namely *Mechanics*, *Dynamics*, and *Aesthetics*, PhyDSL consists of five gameplay definition sections: (i) *mobile and static actor definition*, (ii) *environment and layout definition*, (iii) *activities definition*, (iv) *scoring rules definition*, and (v) *controllers definition*. PhyDSL has been implemented as an Eclipse-based syntax-directed editor using Xtext[7]. In this section, we explore the PhyDSL language through the specification of *Alien Miner* (Figure 3.2), a physics-based platformer with similar gameplay characteristics to Asteroids (Atari, 1979). The main actor of the *Alien Miner* is an adventurous alien whose primary mission is to collect precious gems while exploring the galaxy.

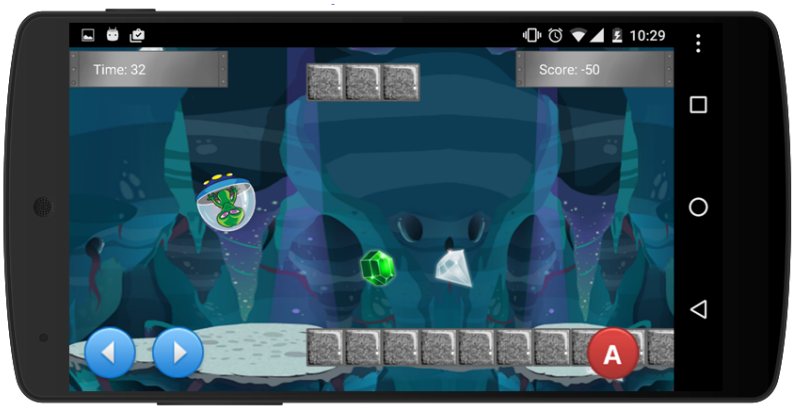


Figure 3.2: Alien Miner - Gameplay Example

[7]https://eclipse.org/Xtext/

In this version of *Alien Miner*, the main actor of the game is found inside a cave with hidden treasures and dangerous meteorites. The goal of the game is to guide the alien through the cave while collecting its precious treasures. For every emerald and diamond collected, the player will receive 30 and 20 points, respectively. Emeralds are collected when the alien collides with them, and diamonds are collected when the player taps on their screen position. If the alien gets hit by a meteorite, the player loses 20 points. The game has a total duration of 80 seconds. However, if the player reaches a teleportation portal at the end of the level, the game ends.

Using The Type System

PhyDSL offers an enumeration-based type system that allows developers to specify concrete values for a wide range of variables to be used throughout a gameplay design (Figure 3.3).

```
Types:

// Resources
resource alien: "alien"
resource meteorite: "asteroid_fire"
resource emerald: "emerald"
resource brick: "granite"
resource planet: "alien_planet"
resource crash: "glass_koenig"
resource upBtn: "arrow_up"
resource downBtn: "arrow_down"
resource rightBtn: "arrow_right"
resource leftBtn: "arrow_left"
resource aBtn: "a_button"
resource diamond: "diamond"
resource star: "level_end"

// Elasticity
elasticity zero: 0.0
elasticity rock: 0.05
elasticity wood: 0.5
elasticity tennisBall: 2.0
elasticity beachBall: 4.0

//Friction (us)
friction zero: 0.0
friction teflon: 0.04
friction ice: 0.1
friction brick: 0.4
friction rough: 0.7
friction infinite: 100.0

// Density g/cm^3
density zero: 0.00
density cork: 1.12
density rubber: 1.2
density stone: 2.3
density granite: 2.6
density glass: 2.7
density lean: 11.3

// Linear speed
linear speed fast: (40.0, 40.0)
linear speed slow: (3.0, 3.0)

// Angular Velocity
angular velocity fast: 40.0
angular velocity slow: 1.0

// Gravity
gravity moon: 4.0
gravity earth: 9.8

// Actor Sizes
size tiny: 0.2
size teensy: 0.3
size short: 0.4
size medium: 0.45
size small: 2.0
size huge: 8.0

// Actions
force up: (0.0, -5.0)
force down: (0.0, 5.0)
force right: (3.0, 0.0)
force left: (-3.0, 0.0)
```

Figure 3.3: PhyDSL - Type System

The type system includes physical properties that will be used in the creation of the game actors, such as “elasticity”, “friction”, “density”, and “size”. Furthermore, vector and scalar variables can be declared such as “linear speed”, “angular velocity”, “gravity” and “acceleration”. They are used to describe the properties of a game’s environment, as well as its control patterns, and non-player activity properties. In turn, “resources” are used to create references to the graphical and sound assets of a game.

Defining The Game Actors

Using PhyDSL’s *mobile and static actor definition* section, game designers specify the game actors and their physical properties. An actor is defined using eight different properties (Figure 3.4). The first three, namely “density”, “elasticity” and “friction” define the way an actor behave in the physics simulation of the game, e.g., collisions and gravity forces. Conversely, the properties “image”, “size”

and “shape” determine the actor’s look and feel. All the properties assume concrete values specified in the type section of a gameplay definition.

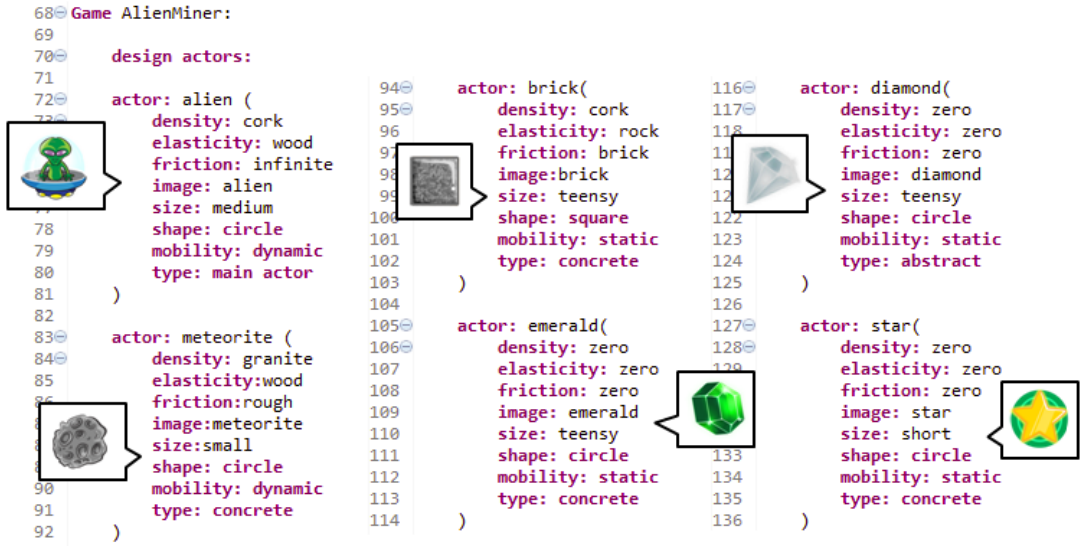


Figure 3.4: PhyDSL - Actor Definition

The “shape” property can assume two values, i.e. *circle* and *square*. This property allows PhyDSL to precisely calculate the effects of collision events in the simulation of a game.

PhyDSL distinguishes between mobile and static actors; while mobile actors can be affected by collisions and the game’s environmental forces, such as gravity or acceleration vectors, static actors are not affected by any force. The “mobility” property can be defined by two concrete values, i.e., *dynamic* and *static*. In *Alien Miner*, mobile actors can be used to model the game’s meteorites, along with its main actor. In turn, static actors can be used to define layout elements, such as the individual bricks that comprise a platform, or gameplay elements, such as the diamonds and emeralds that the player must collect.

Finally, the “type” property specifies whether an actor is an “abstract”, “concrete” or a “main actor”. Abstract actors do not interact with any other actor in the game. They can be used as “immutable” graphical elements to enrich the environment of a game. As a concrete example, if a developer wants to add a tree as a part of the background of the game, she can do so using an abstract actor defined using the image of a tree. In effect, the physical properties of an abstract actor are ignored by the physics engine, and no events are generated upon collision with other actors. It is important to mention that even though abstract actors are not considered in the physics simulation of a game, they can trigger touch events, e.g., checkpoints. In the case of *Alien Miner*, diamonds are abstract actors. The idea behind this design decision is to make diamonds “immutable” for the physics engine, yet allowing the player to collect them using touchscreen events.

Concrete actors are the most commonly used in classic gameplay designs. They observe all the physical interactions supported by the physics engine, such as collisions and gravity forces. Finally, labeling an actor as a *main actor* enables PhyDSL to provide complex camera behaviors, and on-screen controls. Below, we present the different camera behaviors that can be defined in PhyDSL. All camera tracking patterns are inherently linked to the main actor of the game. Furthermore, we explore how developers can design on-screen controls i.e., buttons and keypads, or interactive game elements that manipulate the position of its main actor. It is important to mention that not all games need a main actor; games like Candy Crush[8] and Bejeweled[9] do not have one.

Defining The Game Layout and Environment

PhyDSL uses a floating point 2D grid-based coordinate system to manage the location of the actors and non-player activities in a gameplay design. Using the coordinate system, developers can create different layouts and place actors within the game's canvas to create engaging and challenging experiences. As mentioned before, PhyDSL offers a graphical interpreter that analyzes textual gameplay specifications, and diagrammatically portrays their visual layout[10].

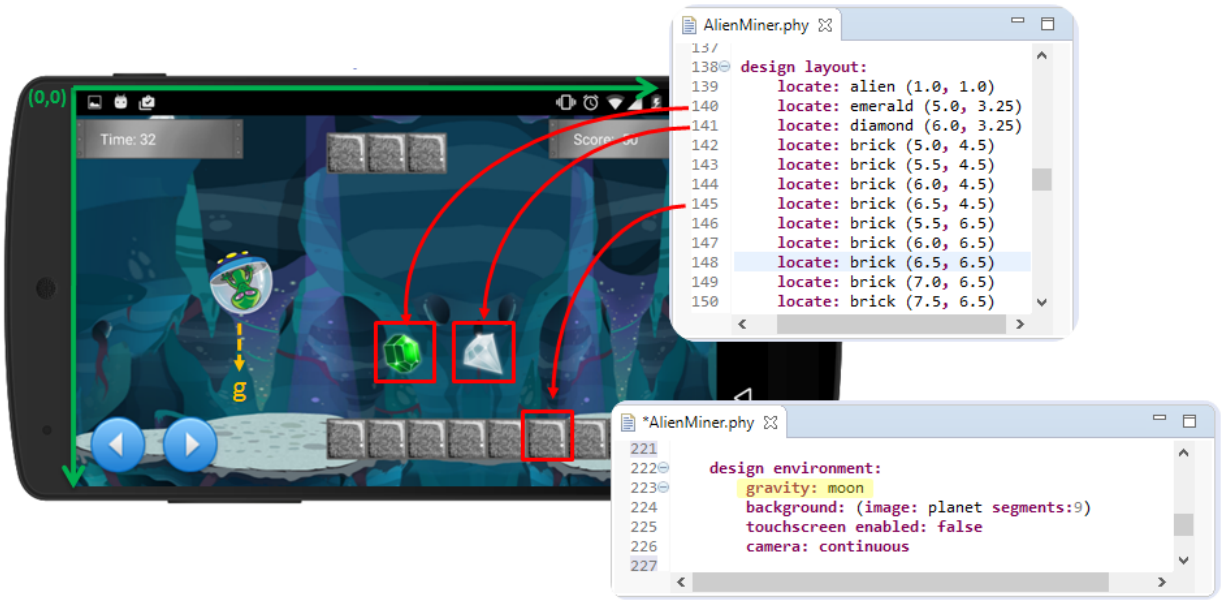


Figure 3.5: PhyDSL - Layout and Environment Definition

In PhyDSL, the origin of the coordinate system is located in the upper-left corner of the world. Furthermore, the size of the world is determined by the size of its background. In *Alien Miner*, we are interested in building a variety of platforms using "brick" actors. Furthermore, we would like to

[8] https://king.com/game/candycrush
[9] http://www.bejeweled.com/
[10] A video demo of this feature can be found at `https://youtu.be/9HZ637bplGw`

place assorted gems in different locations of the world to provide positive feedback to the players (Figure 3.5). The main actor of the game, i.e., "alien", has an initial position corresponding to the coordinate (x=1.0, y=1.0). Since the "alien" actor is mobile, it will be affected by collisions and the gravitational forces of the game simulation as soon as the game starts. The gravity of the game can be defined in the environment section of the gameplay design using the property "gravity". In *Alien Miner*, the gravity of the planet where the game takes place is equivalent to that of Earth's moon.

The *environment and layout definition* section also contains the property "background" which defines the background image of the game (Figure 3.5). Since *Alien Miner* is a platformer that takes place inside of an extraterrestrial cave, a large horizontal background is needed to provide a deep sense of immersion for the player. This is common in modern platformer games such as Jetpack Joyride[11], Super Meat Boy[12], and Sword of Xolan[13]. In order to specify the background of a game, two variables are used, i.e., "image" and "segments". The "image" variable specifies the name of the graphical assets that comprise the background. The "segments" variable indicates the number of individual slices that comprise a background image. A single background image can be comprised by multiple background slices identified by consecutive numerical identifiers. PhyDSL horizontally assembles the slices provided by the developer in execution time (Figure 3.6).

It is worth noticing that developers may use a single-slice strategy in order to manage the background assets of a game. However, we have observed that this causes an unnecessary memory overhead for the rendering engine of PhyDSL. Indeed, single-slice backgrounds need to be completely loaded during the initialization of the game, which causes jittery gameplay experiences.

In the environment section, the property "touch screen" can be set to *true* or *false* in order to enable players interacting with the game actors using touchscreen gestures. In *Alien Miner*, this variable is set to *false*. Below, we explore the definition of on-screen controls as the main interaction mechanism of this particular game.

Defining The Game Camera

The *environment and layout definition* section also includes the "camera" property. This property captures the camera-tracking behaviors of the game. The "camera" property can be set in three possible ways, i.e., *continuous*, *discrete* or *none*.

If the camera property is set to *none*, the first background slice loaded by the game is presented to the player. This setting is appropriate for single scene games that conform to gameplay designs such as in Candy Crush and other board-based games. A *continuous* camera follows the main actor of the game while keeping it at the center of the screen. This type of camera provides a continuous flow

[11] https://halfbrick.com/our-games/jetpack-joyride/
[12] http://supermeatboy.com/
[13] https://www.swordofxolan.com

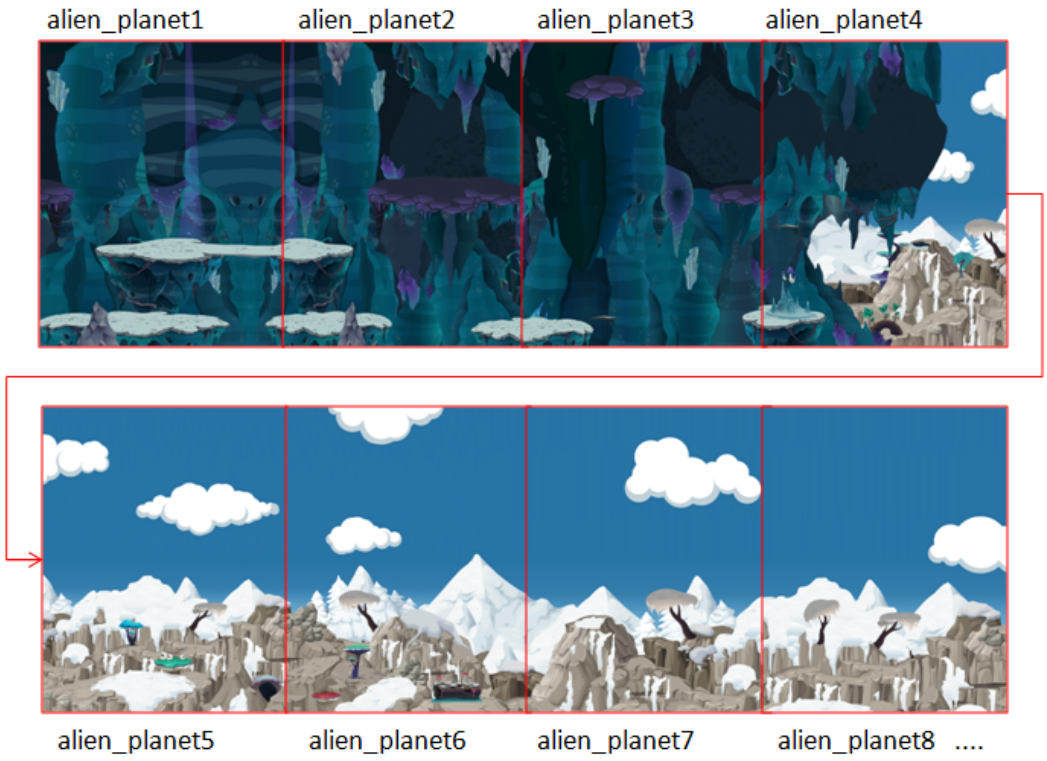


Figure 3.6: Alien Miner Background Slices

of movement through the world of the game (Figure 3.7-A). We have chosen a continuous camera behavior for *Alien Miner*. Finally, a *discrete* camera will swap background scenes as the main actor approaches the boundaries of the screen (Figure 3.7-B). A discrete camera is suitable for games with high-density backgrounds and for games with large screen-size to actor-size ratio. In both cases, the scene swapping will not be frequent, thus minimizing potential continuity disruptions.

Defining The Game Activities

Let us now introduce PhyDSL's *activity* concept, which allows game developers to add interactive elements in their gameplay design. The *activity* concept is used to define the interactive gameplay elements dictated by time, such as the *appearance* and *movement* of mobile actors. They are modeled as Event - Condition - Action rules, associated with a given actor. Specifically, two types of rules can be modeled in the form of *"When <timer condition> actor <actor ID> moves <MoveProperties>"* and *"When <TimerCondition> actor <actor ID> appears <AppearanceProperties>"*.

Appear Activities describe the iterative appearance of an actor dictated by time. *Appear Activities* are comprised by a name along with five different properties, i.e., the "actor" of interest, the appearance "frequency", its "angular velocity" and "linear speed", and the "position" of the

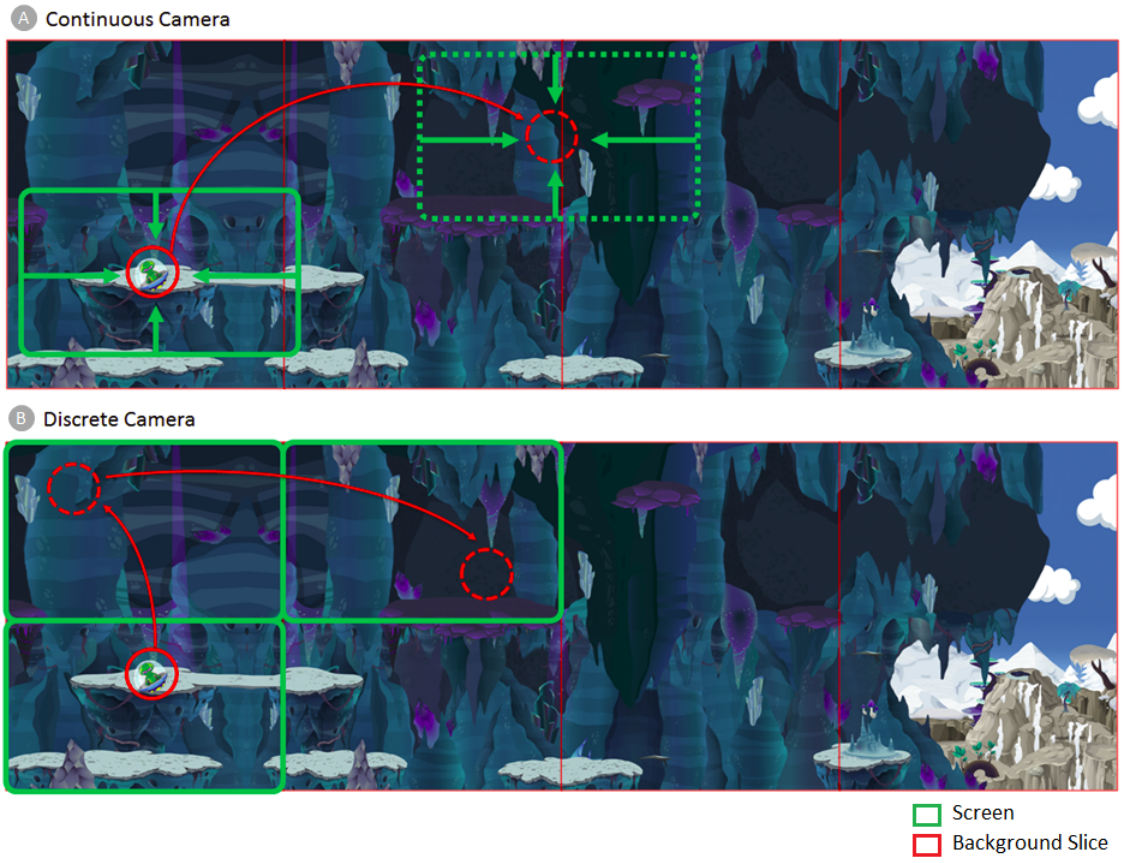


Figure 3.7: PhyDSL - Camera Behaviors

appearance event. In *Alien Miner*, two *appearance activities* are defined to model meteorites that spawn every 4 and 8 seconds (Figure 3.8).

Since appearance events are typically linked to mobile actors, their position determines the initial location of their corresponding actor in the coordinate system. Gameplay designs that include elements such as cannons, particle sprays, and shooters, can be modeled with this type of events. Furthermore, static elements can be used with appear activities to model scenarios such as opening and closing doors. In fact, the usefulness of activities to model gameplay elements is limited only by the designer's imagination.

Defining The Game Scoring Rules

A big component of any gameplay design is the ability to define scoring rules that provide feedback and motivate the player [149]. Similarly to *activities*, PhyDSL's *scoring rules* use an Event - Condition - Action structure. *Scoring rules* can be triggered by *time events*, *collision events*, or *touch-screen events*. Furthermore, each *scoring rule* can result in four possible actions: (i) the point count of the game changes positively or negatively; (ii) the game comes to an end; (iii) the player receives auditory or haptic feedback, and (iv) one of the associated actors disappear.

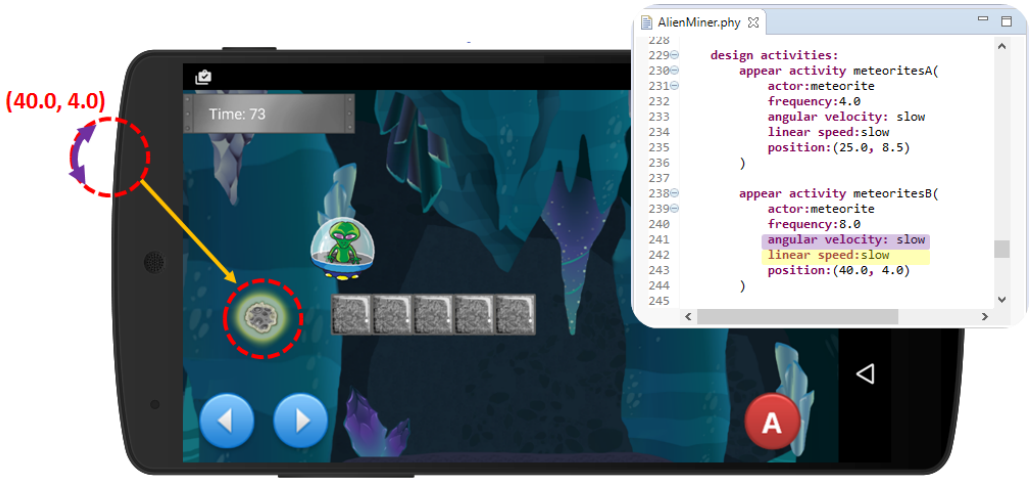


Figure 3.8: PhyDSL - Activities

The signature of a *scoring rule* depends on its type. *Collision Rules* include the name of the actors that, upon collision, will activate or trigger the scoring rule (Figure 3.9-A and B). *Touch Rules* are defined by specifying the name of the actor that will trigger the rule if touched by the player (Figure 3.9-C). Finally, *Time rules* are specified by setting a countdown timer which zero-event will trigger the rule (Figure 3.9-D).

All *scoring rules* include three mandatory properties regardless of their type, i.e., "points", "game ends", and "haptic feedback". The property "points" can positively or negatively affect the point count of the game. The property "game ends" indicates whether the rule will end the game immediately after the rule is triggered. Furthermore, the property "haptic feedback" indicates whether the device will vibrate if the rule is triggered.

Scoring rules include two optional properties, namely "actor disappears" and "sound feedback". The former indicates whether there is an actor that disappears if the rule is triggered. In *Alien Miner*, meteorites will disappear upon collision with the alien. Moreover, gems will disappear if they are collected (either by collision with the alien or by touchscreen event). Finally, the "sound feedback" property specifies whether the game will use a sound asset to provide auditory feedback to the player once a rule is triggered. In *Alien Miner*, a crashing sound will provide feedback when a diamond is collected.

Defining The Game Controls

The final step on the creation of a gameplay design is to optionally specify a set of controls for the main actor of the game. This section is only available when the game has a main actor, and it is particularly suitable for games in which touchscreen events on actors are disabled. Control elements

Figure 3.9: PhyDSL - Scoring Rules - Collision- Touch- and Time-based

are on-screen touch-enabled artifacts defined using a "name" that specifies its unique identifier, an “image” and a “position” which determine the rendering properties of the control element, and a “moves” property which specifies the vectorial force that will be applied to the main actor when the control is touched. In *Alien Miner*, three controls are defined to manipulate its main actor: left (left arrow), right (right arrow), and upwards (“A” button) (Figure 3.10).

Figure 3.10: PhyDSL - Controls

3.1.3 The PhyDSL Game Catalogue

A catalogue of community-submitted games built using PhyDSL can be found at `https://guana.github.io/phydsl/catalogue.html`. Currently, the catalogue includes a platformer, i.e., *Snowy the Penguin* (Figure 3.11), a top-down view game, i.e., *Volcanic Maze* (Figure 3.11), and a reaction-based game inspired by the widely popular Flappy Bird[14], i.e., *Castle Barrage* (Figure 3.13).

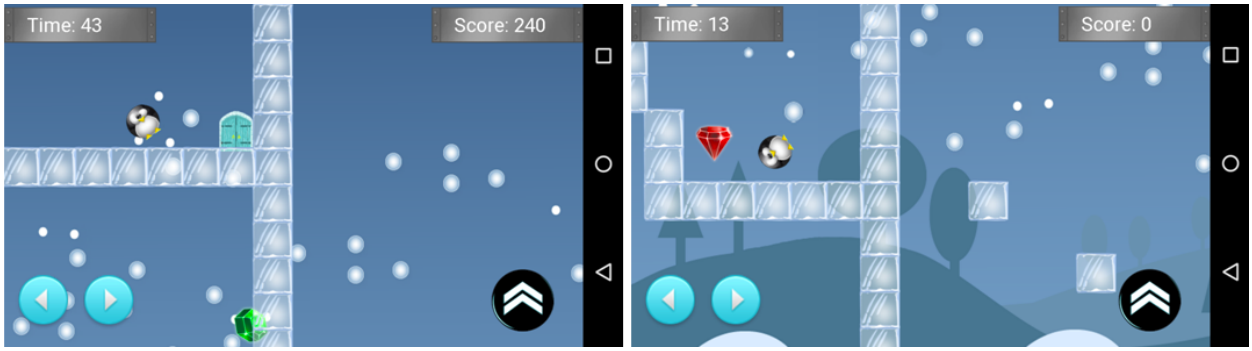


Figure 3.11: PhyDSL - Snowy The Penguin

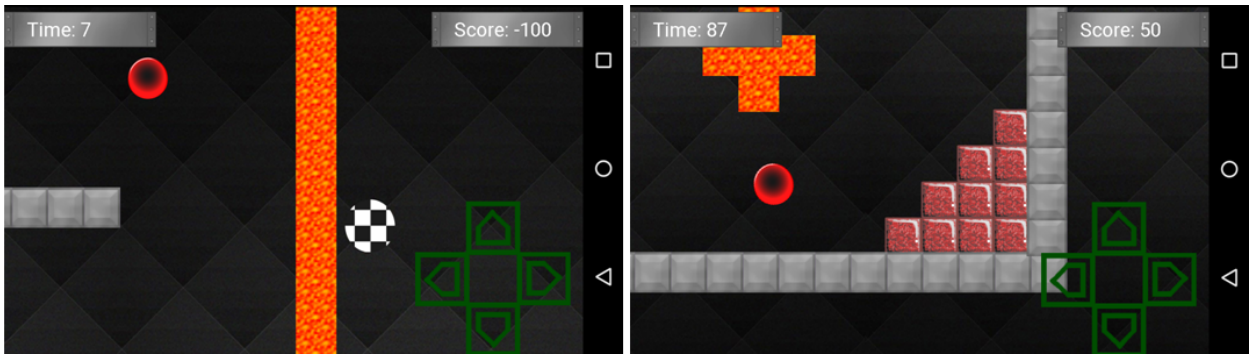


Figure 3.12: PhyDSL - Volcanic Maze

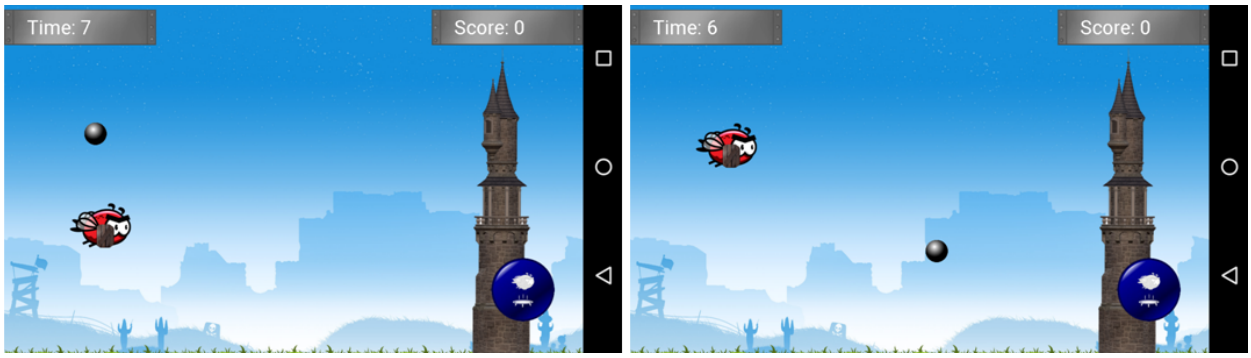


Figure 3.13: PhyDSL - Castle Barrage

[14] https://flappybird.io/

3.2 ScreenFlow

ScreenFlow is a design environment for mobile application storyboards. It enables developers to quickly translate user-interface sketches into application skeletons, including interface navigation logic. ScreenFlow consists of a textual domain-specific language, and a linear model-transformation chain that includes one M2M transformation, and two M2T transformations written in ATL and Acceleo, respectively (Figure 3.14). ScreenFlow is designed for novice Android application developers and for rapid software prototyping environments, such as hackathons. A complete description and demo video of ScreenFlow can be found at `https://guana.github.io/screenflow`.

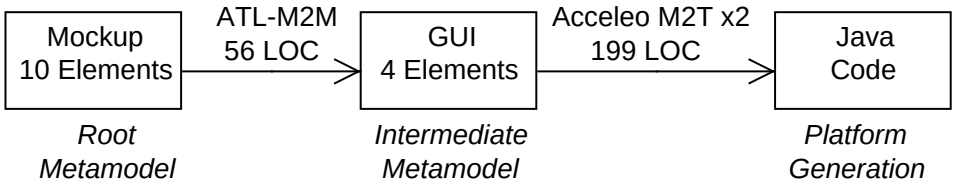


Figure 3.14: ScreenFlow's linear model-transformation chain: 1 (M2M) ATL transformation comprised by 4 matched rules and 1 helper, and 2 (M2T) Acceleo transformations containing 43 binding expressions.

3.2.1 The ScreenFlow Language

In the early stages of a mobile application's design, storyboards give developers a way of visualizing its navigation patterns and execution flow. Furthermore, storyboards highlight triggers, such as buttons and drop-down menus, that make an application transition between different screens. Mobile development platforms, such as Android and PhoneGap, implement a Model-View-Controller (MVC) architecture in their applications. More often than not, these platforms use structured file descriptors to define an application's layout and event handlers, i.e., isolating the application's *control* infrastructure in self-contained units of code. Initializing and synchronizing the diverse development artifacts in a mobile application is a challenging and error prone task [150]. ScreenFlow enables developers to textually describe an application's *screens*, *triggers*, *screen transitions*, and *hardware permissions* in order to generate an extensible and synchronized MVC boilerplate. Similarly to PhyDSL, ScreenFlow has been implemented as an Eclipse-based syntax-directed editor using Xtext.

Figure 3.15, presents a brief example of ScreenFlow in the context of a sound recording application inspired by SoundCloud[15]. In this example, the application consists of five screens, i.e., *login, playlist, subscribe, recording* and *settings*. The *login* screen is the main/landing screen of the application. Moreover, four triggers can be found in the application definition, i.e., *loginButton, subscribeButton, recordButton*, and *settingsButton*. The former two are regular buttons; the later are hidden in the application's menu. In this example, four transitions define the navigation paths available between screens, i.e., *toPlaylist* and *toSubscribe* which transition the application from the *login* screen to the *playlist* and *subscription* screens, and *toRecord* and *toSettings*, which transition the application from the *playlist* screen to the *recording* and *settings* screens, respectively. Each trigger is included in an application screen or menu if a transition has been defined with it. The final location of a trigger is determined by its originating transition screen. If the same trigger is used to transition from two screens, a unique identifier is assigned to each of them during the generation. Finally, the application is set to allow *networking* and *storage* access permissions.

```
define application: MySoundCloud

  package: "edu.ualberta.mysoundcloud"

screens:
     screen Login is main:true
     screen Playlist
     screen Subscribe
     screen Recording
     screen Settings

triggers:
     trigger loginButton
     trigger subscibeButton
     trigger recordButton is menu:true
     trigger settingsButton is menu:true

transitions:
     transition toPlayList from:Login to:Playlist trigger:loginButton
     transition toSubscribe from:Login to:Subscribe trigger:subscibeButton
     transition toRecord from:Playlist to:Recording trigger:recordButton
     transition toSettings from:Playlist to:Settings trigger:settingsButton

permissions:
     geolocation:false
     networking:true
     camera:false
     storage:true
```

ScreenFlow
Mobile Storyboards, In a Nutshell.

Figure 3.15: The ScreenFlow Language

[15] https://soundcloud.com/ - a fully featured video demo of ScreenFlow can be found at `https://guana.github.io/screenflow`

4

A Traceability Conceptual Framework

In this chapter, we present a formal conceptual framework for end-to-end traceability at the metamodel level. Section 4.1 presents *Library to Anonymous Index*, a pedagogical model-transformation chain that we will use to illustrate our traceability conceptual framework. Sections 4.2 briefly discusses binding expressions in the context of OCL. Section 4.3 and 4.4 define traceability links in the context of M2M and M2T transformations. Section 4.5 formally describes our conceptual framework in model-transformation chains.

4.1 A Model Transformation Chain Example

In this section, we present *Library to Anonymous Index*, a simple model-transformation chain inspired by the popular Book2Publication transformation example [151]. It consists of one ATL (M2M) transformation and one Acceleo (M2T) transformation, Book2Publication (Listing 4.1) and Publication2HTML (Listing 4.2), correspondingly.

Book2Publication is a simple (M2M) transformation in which a model describing a library is transformed into a simpler anonymous publication database (Figure 4.1). The source metamodel of the transformation is Book, which consists of four metamodel elements: Library, Book, Chapter, and Summary. The root element of the metamodel is Library which contains a collection of *books*[1]. The element Book contains a set of *chapters*. A Chapter contains a *digest* of its contents in the form of a Summary. The target metamodel, i.e., *Publication*, is simpler. It contains two metamodel elements: Database and Publication. The Publication element represents an anonymous-database entry, consisting of a book's *title*, *prologue*, and total number of pages, i.e., *nbPages*.

[1] We use *italics* to denote metamodel element attributes. We use the terms *attribute* and *property* interchangeably.

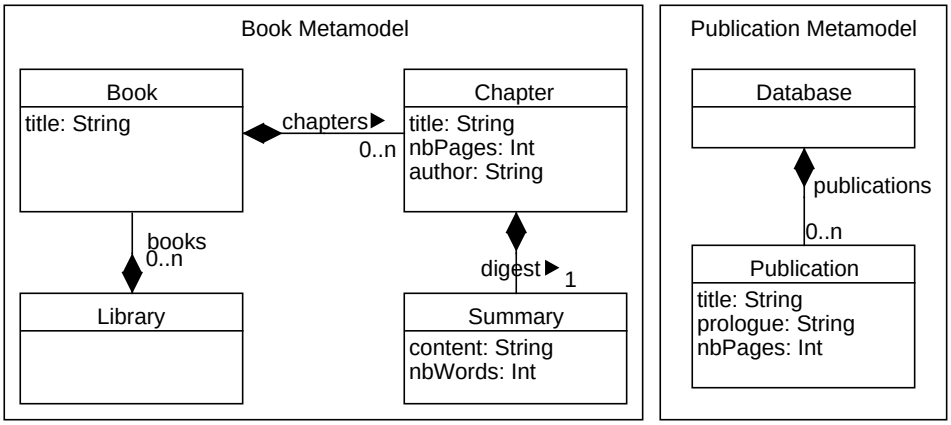


Figure 4.1: Book and Publication Metamodels

```
1  module Book2Publication;
2  create OUT : Publication from IN : Book;
3
4  -- Main matched rule.
5  rule Main{
6       from
7         l : Book!Library
8       to
9         db : Publication !Database(
10                publications <- l.books
11        )
12 }
13
14 -- Transforms a book into an anonymus publication entry
15 rule Book2Publication{
16      from
17        b : Book!Book
18      to
19        p : Publication ! Publication (
20              title <- b. title ,
21              prologue <- b.chapters . first (). digest . content ,
22              nbPages <- b.getSumPages()
23        )
24 }
25
26 -- Computes the sum of all pages given a Book
27 helper context Book!Book def : getSumPages() : Integer =
28       self . chapters ->collect( f | f . nbPages ).sum()
29 ;
```

Listing 4.1: Book2Publication Model-to-Model Transformation

The Book2Publication transformation consists of three transformation rules, namely `Main`, `Book2Publication` and `getSumPages()` (Listing 4.1, lines 5-13, 15-24, and 27-29, respectively). `Main` is the entry point of the transformation, while `Book2Publication` flattens the contents of a Book into an anonymous-publication entry. In turn, `getSumPages()` serves as a helper rule that calculates the total length of a book. In summary, the transformation considers all the chapters of a book to calculate the total number of pages of its corresponding publication. Furthermore, the summary of the first chapter of a book is used as the prologue of its corresponding publication entry.

```
1 [module generateIndex('http://ualberta.ssrg.publication')]
2 [template public generateIndex(aDatabase : Database)]
3 [comment @main/]
4 [file ('index.html', false, 'UTF-8')]
5 <h2> Publication Database </h2>
6  <table border="" style="with:50%">
7     [for (p : Publication | publications)]
8     <tr>
9       <td>
10          <p><h4>[p.title/]</h4></p>
11          <p>[p.prologue/]</p>
12          <p>Total Pages: [p.nbPages/]</p>
13      </td>
14    </tr>
15    [/for]
16  </table>
17 [/file]
18 [/template]
```

Listing 4.2: Publication2HTML - Model-to-Text Transformation

The Publication2HTML (M2T) transformation generates an HTML index with the anonymous entries contained in a publication Database. An HTML table is populated using an iterative binding expression (Listing 4.2, line 7). Each row in the table contains a publication's *content, prologue,* and total number of pages, i.e., *nbPages* (Listing 4.2, lines 10, 11, and 12, respectively).

4.2 Binding Expressions

In rule-based M2M transformations, binding expressions determine the mappings between source and target metamodel elements. The role of transformation rules is to group binding expressions according to their metamodel-root context. In template-based M2T transformations, binding expressions, defined in expansion rules, derive snippets of text using templates designed for reuse [53]. We present a traceability conceptual framework for M2M and M2T binding expressions defined in OCL. A binding expression is defined in terms of four main elements [40]:

1. a collection of variables that define the metamodel-root context of the expression (i.e., a source metamodel element);

2. a collection of variables that define the target of the expression (i.e., a target metamodel element in a *model-to-model* transformation, or a target template position in a *model-to-text* transformation);

3. a collection of metamodel element properties (i.e., the source attributes manipulated by the expression, and in the case of *model-to-model* transformations, the expression's target attribute);

4. a collection of operations (e.g., arithmetic, string concatenation, iterator) that manipulate one or more properties (e.g., concatenating two attributes, or selecting elements form a collection); and set of keywords (e.g., if, then, else, and, or, not) used to constrain the execution of the expression.

Our traceability conceptual framework considers fine-grained traceability links as dependency relationships created by the semantics of a transformation's binding expressions. A binding expression navigates a source metamodel to manipulate one (or multiple) source attribute(s) in order to derive a target attribute. The starting point of the metamodel path followed by an expression is determined by its metamodel-root context. This path is known as the *metamodel footprints* of a binding expression [152]. It is worth noticing that since a binding expression might navigate multiple source attributes, it might define multiple bindings between a target element and several source elements. Therefore, numerous traceability links can be derived from the execution semantics of a single binding expression.

We consider M2M traceability links as dependency relationships between a transformation's source and target metamodel elements and properties. We conceive M2T traceability links as dependency relationships between a transformation's source metamodel elements and properties, and its template lines. Let us briefly discuss two binding expressions defined in *Library to Anonymous Index*.

In the Book2Publication (M2M) transformation, we observe the `prologue<-b.chapters.first().digest.content` binding expression (Listing 4.1, line 21). It binds the source attribute *content* of Summary, to the target attribute *prologue* of Publication. Furthermore, it uses the variable *b* to gain access to the expression metamodel-root context, i.e., Book (Listing 4.1, line 17). The expression uses a collection of statements and operations, e.g., `first()`, to navigate from its metamodel-root context, to the source attribute of interest. The *metamodel footprints* of this binding expression can be characterized as *chapters.digest.content*. The navigation path followed by this binding expression, not only defines a dependency relationship between the source attribute *content* and the target attribute *prologue*, but also with the source attributes *chapters* and *digest*.

The Publication2HTML (M2T) transformation contains the `[p.title/]` binding expression (Listing 4.2, 10). This expression binds the attribute *tile* of Publication, to the line 10 of the transformation template. It uses the variable *p* in order to access the metamodel-root context of the expression, i.e., Publication. In this case, the root-context element is not determined by a transformation rule, but by a preceding –iterative– binding expression (Listing 4.2, 7). We discuss in

detail the different types of binding expressions in M2M and M2T transformations in Chapter 5. As mentioned before, we distinguish between two different types of bindings, namely *explicit bindings* and *implicit bindings*. In the remainder of this chapter we formalize our traceability-link definition in the context of M2M and M2T transformations, and explore the definitions of *explicit* and *implicit bindings*.

4.3 A Formal Traceability Framework for M2M Transformations

Let us now formalize our traceability conceptual framework for model-to-model transformations.

- Let $M2M : \{SE, TE, E\}$ be a model-to-model transformation.
- Let $SE : \{s_1, s_2, ..., s_i\}$ be the set of all the source-metamodel elements in $M2M$. A source-metamodel element contains a collection of source attributes $s_i : \{s_i sa_1, ..., s_i sa_j\}$.
- Let $TE : \{t_1, t_2, ..., t_r\}$ be the set of all the target-metamodel elements in $M2M$. A target-metamodel element contains a collection of target attributes $t_r : \{t_r ta_1, ..., t_r ta_k\}$.
- Let $E : \{e_1, e_2, ..., e_n\}$ be the collection of binding expressions in $M2M$. A binding expression consists of the collection of binding tuples in its metamodel-navigation path $e_n : \{e_n b_1, e_n b_2, ..., e_n b_w\}$. A binding tuple $e_n b_w : \{t_r ta_k, s_i sa_j\}$ defines a dependency relationship between a source attribute $s_i sa_j$ and a target attribute $t_r ta_k$.
- Let $< e_n, \preccurlyeq >$ be a partially ordered set, then $\{e_n b_a, e_n b_b\} \in < e_n, \preccurlyeq >$ if the source attribute in $e_n b_a$ is required to reach the source attribute in $e_n b_b$ from the metamodel-root context of e_n.
- Let $e_n b_x \in e_n$ be an explicit-binding tuple if there is no binding tuple $e_n b_y \in e_n$ that satisfies $e_n eb_x \preccurlyeq e_n b_y$.
- Let $e_n b_y \in e_n$ be an implicit-binding tuple if there is a binding tuple $e_n b_x \in e_n$ that satisfies $e_n b_y \preccurlyeq be_n b_x$.
- Finally, let $TLM\{tlm_1, tlm_2, ..., tlm_p\}$ be the set of $M2M$ traceability links, consisting of fine-grained traceability links $tlm : \{s_i, t_r, s_i sa_j, t_r ta_k, e_n\}$, where

$$(\forall\, tlm | tlm \in TLM : (\exists e \in E, \exists b \in e : tlm_{s_i sa_j} = b_{s_i sa_j} \wedge tlm_{t_r ta_k} = b_{t_r ta_k} \wedge tlm_{e_n} = e)$$

A M2M *traceability link*, $tlm : \{s_i, t_r, s_i sa_j, t_r ta_k, e_n\}$ represents a dependency between a source element s_i and a target element t_r, if there is a binding tuple b defined in the *footprints* of

a binding expression e, whose source and target attributes are $s_i sa_j$ and $t_r ta_k$. It is important to note that in our framework a *traceability link* exists whether its associated binding tuple is *implicit* or *explicit*. Let us now discuss our traceability link definition for M2M transformations using the Book2Publication transformation (Listing 4.1).

			Publication Metamodel (TE)			
			Database (t_1)	Publication (t_2)		
			publications ($t_1 ta_1$)	*title* ($t_2 ta_1$)	*prologue* ($t_2 ta_2$)	*nbPages* ($t_2 ta_3$)
Book Metamodel (SE)	Library (s_1)	*books* ($s_1 sa_1$)	$tlm_1 : e_1 b_1$	⌀	⌀	⌀
	Book (s_2)	*title* ($s_2 sa_1$)	⌀	$tlm_2 : e_2 b_1$	⌀	⌀
		chapters ($s_2 sa_2$)	⌀	⌀	$tlm_3 : e_3 b_1 *$	$tlm_6 : e_4 b_1 *$
	Chapter (s_3)	*title* ($s_3 sa_1$)	⌀	⌀	⌀	⌀
		nbPages ($s_3 sa_2$)	⌀	⌀	⌀	$tlm_7 : e_4 b_2$
		author ($s_3 sa_3$)	⌀	⌀	⌀	⌀
		digest ($s_3 sa_4$)	⌀	⌀	$tlm_4 : e_3 b_2 *$	⌀
	Summary (s_4)	*content* ($s_4 sa_8$)	⌀	⌀	$tlm_5 : e_3 b_3$	⌀
		nbWords ($s_4 sa_9$)	⌀	⌀	⌀	⌀

Table 4.1: Fine-grained M2M Traceability Links in Book2Publication (*) implicit binding, (⌀) no binding.

Table 4.1 summarizes the fine-grained M2M traceability links (tlm) in Book2Publication. Its `main` transformation rule contains one binding expression, e_1 : `publication<- l.books` (Listing 4.1, line 10) with one explicit-binding tuple, $e_1 b_1 : \{publications, books\}$. In turn, its `Book2Publication` rule contains three binding expressions, i.e., e_2 : `title<-b.title`, e_3 : `prologue <- b.chapters.first().digest.content`, and e_4 : `nbPages<-b.getSum Pages()` (Listing 4.1, lines 20, 21, and 22, respectively). The expression e_2 : `title<- b.title` contains one explicit-binding tuple, $e_2 b_1 : \{title, title\}$. The expression e_3 : `prologue<- b.chapters.first().digest.content` is more complex; it contains two implicit-binding tuples, i.e., $e_3 b_1 : \{prologue, chapters\}$ and $e_3 b_2 : \{prologue, digest\}$, and one explicit-binding tuple, $e_3 b_3 : \{prologue, content\}$. Finally, the expression e_4 : `nbPages<-b.getSum Pages()` uses a helper rule. In ATL, helper rules enable developers to factorize commonly used binding expressions; this is a common feature in modern transformation languages. We can in-line the bindings defined in e_4 : `getSumPages()` as follows, `nbPages<-b.chapters ->collect(f|f.nbPages) .sum()`. This expression contains one implicit-binding tuple, $e_4 b_1 : \{nbPages, chapters\}$ and one explicit-binding tuple, $e_4 b_2 : \{nbPages, nbPages\}$. In summary, Book2Publication contains four binding expressions, i.e., e_{1-4}, with seven fine-grained traceability links, i.e., tlm_{1-7}; four of them are due to *explicit bindings*, i.e., $tlm_1, tlm_2, tlm_5, tlm_7$, while the remaining three to *implicit bindings*, i.e., tlm_3, tlm_4, tlm_6.

4.4 A Formal Traceability Framework for M2T Transformations

Let us now extend our conceptual framework to include model-to-text transformations.

- Let $M2T : \{SE, E\}$ be a model-to-text transformation.
- Let $SE : \{s_1, s_2, ..., s_i\}$ be the set of all the source-metamodel elements in $M2T$. A source-metamodel element contains a collection of source attributes $s_i : \{s_i sa_1, ..., s_i sa_j\}$.
- Let $E : \{e_1, e_2, ..., e_n\}$ be the collection of binding expressions in $M2T$. A binding expression consists of the collection of binding tuples in its metamodel-navigation path $e_n : \{e_n b_1, e_n b_2, ..., e_n b_w\}$. A binding tuple $e_n b_w : \{si_s a_j, tl_{k-p}\}$ defines a dependency relationship between a source attribute $s_i sa_j$ and the template line(s) tl_{k-p} where e_n is defined.
- Let $< e_n, \preccurlyeq >$ be a partially ordered set, then $\{b_a, b_b\} \in < e_n, \preccurlyeq >$ if the source attribute in b_a is required to reach the source attribute in b_b from the root context of e_n.
- Let $e_n b_x \in e_n$ be an explicit-binding tuple if there is no binding tuple $e_n b_y \in e_n$ that satisfies $e_n e b_x \preccurlyeq e_n b_y$.
- Let $e_n b_y \in e_n$ be an implicit-binding tuple if there is a binding tuple $e_n b_x \in e_n$ that satisfies $e_n b_y \preccurlyeq b e_n b_x$.
- Finally, let $TLT\{tlt_1, tlt_2, ..., tlt_q\}$ be the set of M2T traceability links, consisting of fine-grained traceability links $tlt : \{s, s_i sa_j, tl_g, e_n\}$, where

$$(\forall\, tlt | tlt \in TLT : (\exists e \in E, \exists b \in e : tlt_{s_i sa_j} = b_{s_i sa_j} \wedge tlt_{tl_g} <= b_{tl_p} \wedge tlt_{tl_g} >= b_{tl_k} \wedge tlt_{e_n} = e)$$

An M2T traceability link $tlt : \{s_i sa_j, tl_k, e_n\}$ represents a dependency relationship between a source element s_i and a template line tl_k, if there is a binding tuple b in an expression e defined in a template line tl_k, whose source attribute is $s_i sa_j$. Similarly to traceability links in M2M transformations, a M2T traceability link exists whether its underlying binding tuple is *implicit* or *explicit*. Let us now discuss our traceability link definition for M2T transformations in the context of Publication2HTML (Listing 4.2).

Table 4.2 summarizes the fine-grained M2T traceability links (ttl) in Publication2HTML. The expressions defined in template lines tl_{10-12} contain three explicit bindings, i.e., $e_1 b_1 : \{title, 10\}$ in e_1 : `[p.title/]`, $e_2 b_1 : \{prologue, 11\}$ in e_2 :`[p.prologue/]`, and $e_3 b_1 : \{nbPages, 12\}$ in e_3 :`[p.nbPages/]`. Iterative expressions such as in e_4 : `[for (p : Publication |`

`publications)]` (Listing 4.2, line 7) determine multiple traceability links. There is a dependency relationship between the metamodel elements used to define iterative or conditional invariants, and their contained template lines. In this case, nine explicit bindings can be identified, i.e., e_4b_{1-9} $\{publications, 7-15\}$. In Chapter 5, we explore the multi-line binding pattern of iterative and conditional binding expressions in template-based M2T transformations.

			Publication Metamodel (SE)			
			Database (s_5)	Publication (s_6)		
			publications (s_5sa_1)	*title* (s_6sa_1)	*prologue* (s_6sa_2)	*nbPages* (s_6sa_3)
Publication2HTML	Template Lines (tl_{7-15})	*7*	$tlt_4 : e_4b_1$	⌀	⌀	⌀
		8	$tlt_5 : e_4b_2$	⌀	⌀	⌀
		9	$tlt_6 : e_4b_3$	⌀	⌀	⌀
		10	$tlt_7 : e_4b_4$	$tlt_1 : e_1b_1$	⌀	⌀
		11	$tlt_8 : e_4b_5$	⌀	$tlt_2 : e_2b_1$	⌀
		12	$tlt_9 : e_4b_6$	⌀	⌀	$tlt_3 : e_3b_1$
		13	$tlt_{10} : e_4b_7$	⌀	⌀	⌀
		14	$tlt_{11} : e_4b_8$	⌀	⌀	⌀
		15	$tlt_{12} : e_4b_9$	⌀	⌀	⌀

Table 4.2: Fine-grained M2T Traceability Links in Publication2HTML (⌀) no binding.

4.5 A Formal Traceability Framework for MTCs

So far we have considered M2M and M2T transformations individually. However, as we mentioned before, a key challenge in the construction model-driven software engineering tools involves the analysis of traceability information in heterogeneous model-transformation chains.

Model-transformation chains (MTCs) are specified in a pipeline architecture, where the output of a transformation serves as the input for the next one. Thus, transitive dependency relationships arise between the metamodels and transformations that comprise a transformation chain. Let us now formally define the concepts of *dependent traceability links* and *end-to-end traceability links*.

- Let $MTC : \{M2M_1, M2T_1, ..., M2M_i, M2T_j\}$ be a model-transformation chain, consisting of multiple *model-to-model* and *model-to-text* transformations.
- Let $TLC : \{tlc_1, tlc_2, ..., tlc_k\}$ be the set of traceability links in the transformations of a model-transformation chain regardless of their type:

$$(\forall\, tlc \in TLC : (\exists M2M \in MTC, \exists M2T \in MTC : tlc \in M2M_{TLM} \vee tlc \in M2T \in M2T_{TLT}))$$

- Let $< TLC, \Leftarrow >$ be a partially ordered set, then $(tlc_a, tl_b) \in < TLC, \Leftarrow >$ if tlc_b *is dependent of* tlc_a. A dependency relationship exists if the target attribute of a traceability link is the source attribute of another.

- Let $ete : \{tlc_1, tlc_2, ..., tlc_x\}$ be a tuple representing an *end-to-end traceability link* composed by individual traceability links, where

$$(\forall\, tlc_a \in ete : (\exists tlc_b \in ete : tlc_a \Leftarrow tlc_b \vee tlc_b \Leftarrow tlc_a)$$

  An *end-to-end traceability link* is a collection of *dependent traceability links* found in the transformations of a model-transformation chain.

As mentioned before, understanding end-to-end dependencies enable developers to interpret the execution semantics of complex transformation ecosystems. As a concrete example, let us examine the following question: *What metamodel elements, attributes, and binding expressions should be considered when debugging potential errors in the total number of pages reported for a publication?*

			Publication			
			Database (t_1, s_5)	Publication (t_2, s_6)		
			publications (t_1ta_1, s_5s_1)	*nbPages* (t_2ta_3, s_6sa_3)		
Book	Library (s_1)	*books* (s_1sa_1)	$ete_1 : \{tlt_8, tlm_1\}$	$\varnothing$	tl_{12}	**P2H.**
	Book (s_2)	*chapters* (s_2sa_2)	$\varnothing$	$ete_2 : \{tlt_3, tlm_6\}$		
	Chapter (s_3)	*nbPages* (s_3sa_2)	$\varnothing$	$ete_3 : \{tlt_3, tlm_7\}$		

Table 4.3: *Library to Anonymous Index* End-to-End Traceability Links Publication2HTML (12)

In order to address this question, developers need to identify the *end-to-end traceability links* of *Library to Anonymous Index* that involve line 12 of Publication2HTML, i.e., $ete1_1, ete_2, ete_3$ (Table 4.3). In summary, the template line 12 in Publication2HTML depends on five metamodel element attributes in five corresponding metamodel elements. These dependencies are defined in four *explicit* binding tuples, i.e., e_3b_1 and e_4b_6 in Publication2HTML, and e_1b_1 and e_4b_2 in Book2Publication, and one *implicit* binding tuple, i.e., e_4b_4 in Book2Publication (Tables 4.1 and 4.2). Furthermore, the related bindings are located in two M2M binding expressions e_1 and e_4 (Listing 4.2, lines 10 and 20-22), and two M2T binding expressions, i.e., e_4 and e_3 (Listing 4.2, lines 7 and 12). Determining the traceability links in *Library to Anonymous Index* is a relatively simple task. This is an illustrative example with simple binding expressions. However, as the complexity of the transformations and metamodels in an ecosystem increase, so does the difficulty of reflecting on its execution semantics [10].

5

A Traceability Analysis Technique

In this chapter, we present a metamodel-level traceability-analysis technique for M2M and M2T transformations. This analysis technique is based on the conceptual framework introduced in Chapter 4. This chapter is structured as follows: Section 5.1 presents a high-level overview of our analysis technique. Sections 5.2 and 5.3 present our traceability-analysis technique for M2M and M2T transformations, respectively. Section 5.4 discusses our evaluation and its corresponding threats to validity.

5.1 Overview

Modern transformation languages use different syntaxes to enable developers define binding expressions (Chapter 2.3). Our traceability-analysis technique has been instantiated in the context of ATL and Acceleo transformation languages. However, it can be generalized to other transformation languages based on OCL.

In ATL, the main goal of our technique is to analyze the abstract-syntax tree of a transformation in order to isolate the binding expressions defined in its transformation rules. Next, our technique analyzes the *footprints* of the binding expressions in order to individualize their constituent binding tuples. Finally, our technique identifies the metamodel elements corresponding to the attributes in each tuple, and exports their (M2M) traceability links (Section 5.2).

In Acceleo, our analysis technique follows a tokenization strategy that isolates binding expressions from a textual template. Once binding expressions have been isolated, their corresponding *footprints* are identified, along with their constituent binding tuples. Our technique identifies the metamodel elements to which the attributes in the binding tuples belong, and exports their corresponding (M2T) traceability links (Section 5.3).

In order to implement a robust traceability analysis for a transformation language, we need to consider its *concrete* and *abstract* syntax. For example, ATL enables developers to factorize binding expressions in *helper rules* that can be used by other expressions in outlying segments of code. Moreover, in ATL, the metamodel-root context of a binding expression is determined by the rule where the expression is located. The implementation of *helper rules*, and the mechanisms to access metamodel-root contexts vary across transformation languages. In the following sections we explore in detail the language abstractions of ATL and Acceleo, while discussing our traceability-analysis technique.

5.2 A Traceability Analysis for M2M Transformations (ATL)

Figure 5.1 diagrammatically depicts our traceability-analysis technique for ATL (M2M) transformations. It consists of three main steps; (i) *abstract syntax tree extraction*, (ii) *binding expression analysis*, and (iii) *binding tuple individualization*.

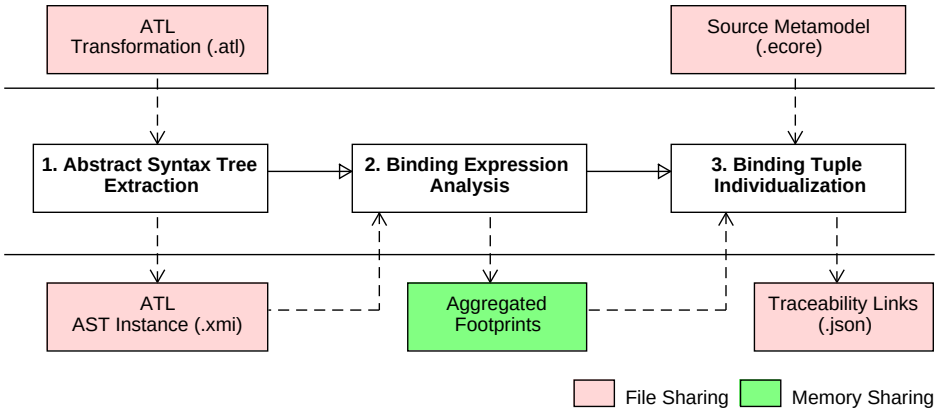


Figure 5.1: ATL Traceability Analysis Technique

5.2.1 Abstract Syntax Tree Extraction

In ATL, there are four types of transformation rules, i.e., *matched rules, lazy rules, called rules* and *helper rules*. In their canonical form, *matched rules* contain declarative expressions that bind to one source metamodel element, in order to derive one (or multiple) target element(s). A *matched rule* might include an action block where imperative statements can be defined. A *called rule* must be invoked from an action block in order to be executed [38]. As mentioned before, *helper rules* enable developers to factorize commonly used expressions. In this thesis, we focus on completely declarative model transformations, i.e., transformations without *lazy* or *called rules* [93].

Similarly to Van Amstel [98], we use the ATL Compiler to obtain the abstract-syntax tree of a transformation, given its textual representation. Figure 5.2 presents the concepts that comprise

the syntax tree of ATL's *matched* and *helper rules* (A and B, respectively). Figure 5.3 graphically portrays the mappings between the textual tokens of Book2Publication (Listing 4.1) and their corresponding syntax-tree nodes. We have removed some of the example's binding expressions and non-terminal symbols to improve its readability.

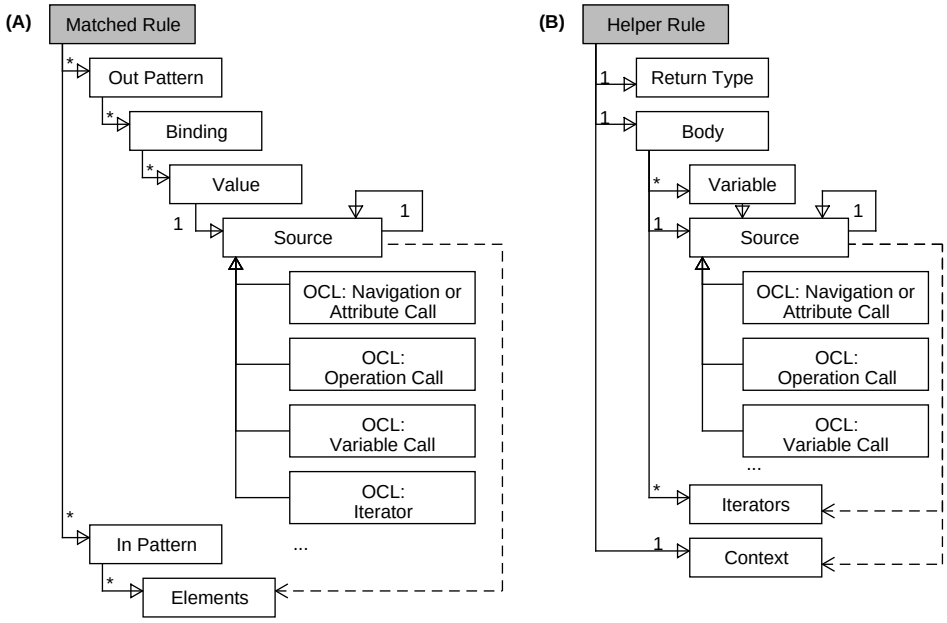


Figure 5.2: ATL - Matched and Helper Rules Simplified AST

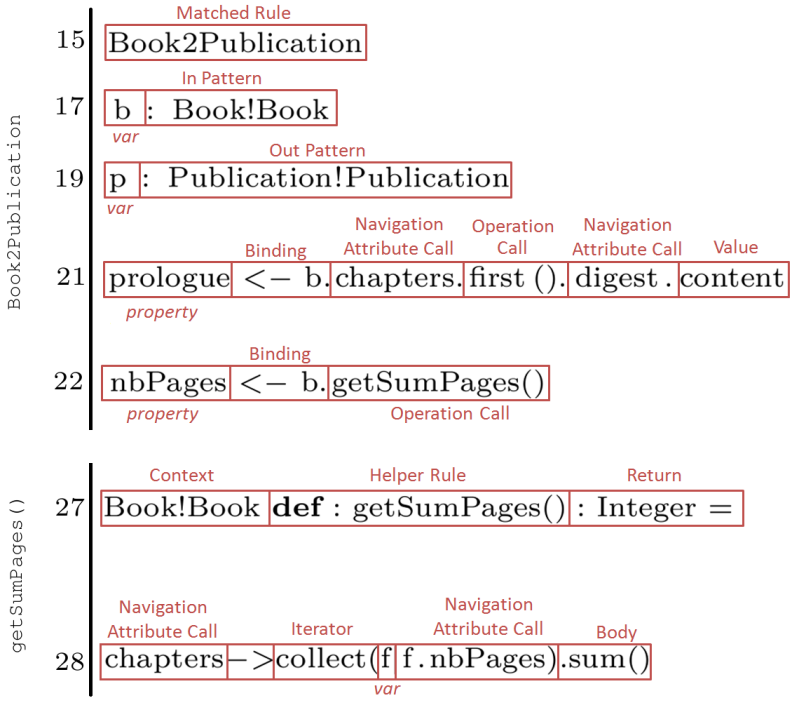


Figure 5.3: An Example of Book2Publication Tokens

The syntax tree of a *matched rule* consists of a collection of InPattern and OutPattern nodes that represent the source and target elements operated by the rule (Figure 5.3, lines 17 - 19). Binding expressions are captured by Binding nodes. They define the mappings between a transformation's InPattern and OutPattern properties. The target property of a Binding is captured by its corresponding Value. A binding expression is symbolized by a list of Source nodes contained by a Value node. A Source node symbolizes individual binding statements. It can be defined in four forms, i.e., NavigationOrAttributeCalls, OperationCalls, VariableCalls and Iterators (Figure 5.3, lines 21 and 22). A NavigationOrAttributeCall is used to symbolize the navigation of a source attribute; an OperationCall is used to symbolize access to OCL operations, e.g., `first()` and `sum()`, and calls to *helper rules*, e.g., `getSumPages()` (Figure 5.3, line 27); VariableCalls symbolize access to local variables, and Iterators access to iterator variables (Figure 5.3, line 28).

The syntax tree of a *helper rule* includes two constructs that determine the scope of its binding expressions, namely Return Type and Context. The former represents the expected output of the *helper rule*, while the latter has a role similar to that of InPattern in the syntax tree of a *matched rule*; symbolizing the helper's metamodel-root context. Moreover, the Body concept captures the binding expressions of the rule. It contains a collection of Source nodes with individual binding statements, including metamodel navigation statements, calls to other *helper rules*, and references to variables where intermediate binding expressions can be defined.

5.2.2 Binding-Expression Analysis (M2M)

The main goal of this step is to analyze the abstract-syntax tree of a M2M transformation looking for the *metamodel footprints* of its binding expressions. We capture the *footprints* of a transformation in an intermediate representation. We refer to this representation as the *aggregated footprints* of a transformation. We propose a static-analysis algorithm to effectively isolate a transformations' binding expressions based on its abstract-syntax tree. In the later stages of our analysis, we use a transformation's *aggregated footprints* to determine its individual binding tuples, and corresponding traceability links. Figure 5.4 diagrammatically depicts a simplified version of the metamodel used to capture the *aggregated footprints* of a M2M transformation.

The *aggregated footprints* of a transformation consist of a collection of *rules*. A *rule* contains a *source element*, and one or multiple *target elements*. A *target element* consists of one (or multiple) *expression(s)* used to derive its attributes. An *expression* captures the *metamodel footprints* of a binding expression starting from its metamodel-root context. Futhermore, an *expression* might be defined as a *call* concept, which represents the usage of a helper rule. An *expression* might contain references to one (or multiple) helper rule(s) in its *stack*.

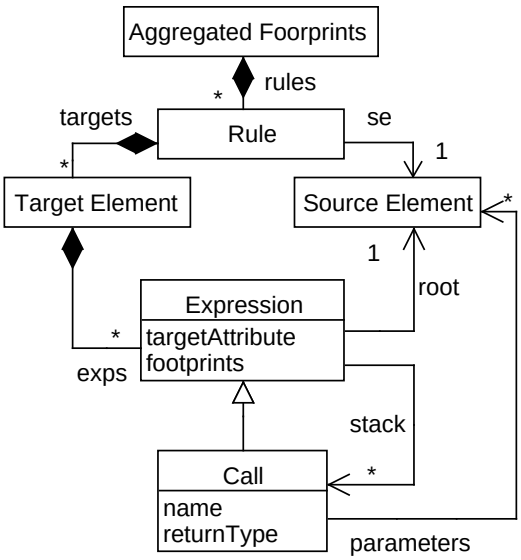


Figure 5.4: M2M Aggregated Footprints

Figure 5.5 presents the *aggregated footprints* of Book2Publication (Listing 4.1). The binding expressions in this example are simple in nature; they do not include binding expressions that use multiple *helper rules*, nor *helper rules* with inner calls. Let us now review some of the most relevant functions that comprise our transformation analysis algorithm.

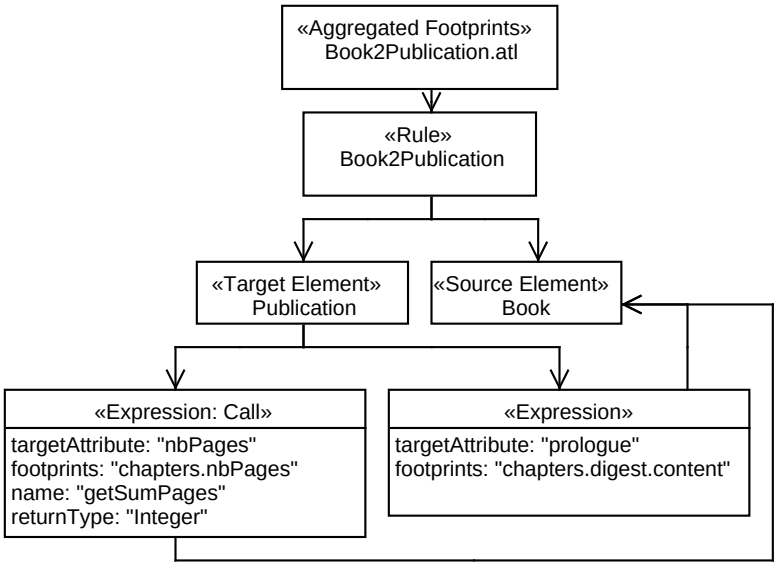


Figure 5.5: M2M Aggregated Footprints Instance - Book2Publication

We use the Document Object Model (DOM) Java API[1] in order to treat the syntax tree of a transformation as a collection of hierarchical nodes. In order to select, prune, and flatten concepts of the syntax tree we have implemented two utility functions, namely `getASTNodeByTag()` and `getASTNodeByType()`. These functions recursively extract concepts depending on their simple- and extended-types, respectively. The main function of our algorithm is `AnalyzeMatchedRules()` (Algorithm 1). It receives the root of a transformation syntax tree, and a list where its *aggregated footprints* will be stored. The functions described in this section use a non-return value strategy; they modify the value of one or more of their parameters to reflect the result of their execution.

Algorithm 1

```
1:  procedure ANALYZEMATCHEDRULES(transformationRoot, rules)
2:      matchedRules ← GETASTNODEBYTAG(transformationRoot, "MatchedRule")
3:      for all matched ∈ matchedRules do
4:          rule ← INIT(matched)
5:          se ← ANALYZESOURCEELEMENT(matched→getChildNodes())
6:          targets ← null
7:          ANALYZETARGETELEMENT(matched→getChildNodes(), targets)
8:          rule.se ← se
9:          rule.targets ← targets
10:         rules → add(rule)
```

Algorithm 2

```
1:  procedure ANALYZESOURCEELEMENT(ruleNodeList)
2:      inPattern ← GETASTNODEBYTAG(ruleNodeList, "InPattern")
3:      return INIT(inPattern)
4:
5:  procedure ANALYZETARGETELEMENT(ruleNodeList, targets, se)
6:      outPatterns ← GETASTNODEBYTAG(ruleNodeList, "OutPattern")
7:
8:      for all out ∈ outPatterns do
9:          te ← INIT(out)
10:         bindings ← GETASTNODEBYTAG(out → getChildNodes(), "Binding")
11:
12:         for all binding ∈ bindings do
13:             INIT(expression, binding)
14:             firstStatement ← GETASTNODEBYTAG(binding, "Value")
15:             expression.root ← se
16:             path ← empty
17:
18:             GETFOOTPRINTS(firstStatement, expression, path, null)
19:             te → add(expression)
20:         targets → add(expression)
```

[1] org.w3c.dom (Java Platform SE 7)

The function `AnalyzeMatchedRules()` traverses all the *matched rules* contained in a transformation's syntax tree. The `init()` function (Algorithm 1, line 4) is used to initialize a *rule* concept in its *aggregated footprints* representation, e.g., its signature and file location within a transformation file. Two functions are used to drive the analysis of its *source element* and potentially multiple *target elements* based on the InPattern and OutPattern nodes of the matched rule under analysis, i.e., `AnalyzeSourceElement()` and `AnalyzeTargetElement()` (Algorithm 2, lines 1 and 5, respectively).

The role of `AnalyzeSourceElement()` is to initialize the rule's *source element* in its *aggregated bindings* representation (Algorithm 2, line 2). The `AnalyzeTargetElement()` function is more complex; it navigates all the potential OutPattern nodes in the rule's syntax tree and builds their corresponding *target elements* (Algorithm 2, line 6). In order to initialize the *expressions* that derive the attributes of a *target element*, `AnalyzeTargetElement()` traverses all the Binding nodes in the OutPattern under analysis (Algorithm 2, line 13-19).

In order to identify the *source attributes* navigated by a binding expression to derive a *target attribute*, each Binding node in the AST is analyzed individually. A Binding node contains a list of linked Source nodes that capture the individual statements of an *expression*. The `GetFootprints()` function is used to recursively traverse this list while tracking its *metamodel footprints* (Algorithm 3).

Algorithm 3

```
1:  procedure GETFOOTPRINTS(currentStatement, expression, path, context)
2:      if currentStatement is a ModelOrNavigationCall then
3:          path → APPEND(currentStatement.name)
4:      else if currentStatement is a OperationCall then
5:          path → APPEND(#) → APPEND(currentStatement.name)
6:          GETCALLSTACK(currentStatement, expression)
7:      else if currentStatement is a VARIABLE then
8:          expression.root ← SOLVEVARREFERENCE(currentStatement.var)
9:
10:     nextStatement ← currentStatement → getChildNodes()
11:     if nextStatement.length == 0 then
12:         expression.footprints ← path // last statement
13:         if context ! = null then
14:             expression.root ← context
15:     else
16:         GETFOOTPRINTS(nextStatement, expression, path, context)
```

The `GetFootprints()` function recursively analyses a Source node depending of its subtype. If the Source node is a NavigationOrAttributeCall it will append its value into the `path`

parameter (Algorithm 3, line 2); this parameter will be recursively accumulated as the function iterates throughout the individual Source nodes in a Binding. The final value will then be assigned to the *footprints* of its corresponding *expression* during backtracking (Algorithm 3, line 12). If the Source under analysis is an OperationCall, the function will append its name to the `path` preceded by a pound sign (#) (Algorithm 3, line 5). This information will be used in the second step of our analysis to solve the metamodel-context switches caused by potential calls to helper rules within the expression. This is necessary to determine the metamodel elements to which subsequent navigated attributes belong. If the Source under analysis is an OperationCall (Algorithm 3, line 6), a recursive function, namely `AnalyzeCall()`, is used to complete its analysis (Algorithm 4).

As a concrete example, assume a binding expression such as `a <- b.x.y.foo().z.bar();` its resulting *footprints* are x.y.#foo.z.#bar. A call is instantiated for both of its helper calls i.e., `foo()` and `bar()`, thus capturing the *footprints* of their inner-binding expressions. Furthermore, if `foo()` or `bar()` use *helper rules*, their corresponding *stacks* are recursively initialized. Each helper *call* represents a potential metamodel-context switch in a metamodel navigation path. It is important to note that if the Source under analysis is a VariableCall (Algorithm 3, line 7) the algorithm will process the last statement of a binding expression, or an intermediate iterator-based statement, e.g., `collect()`[2].

Algorithm 4

Require: *contextIndex* - return types of analyzed helpers.

```
1:  procedure ANALYZECALL(statement, expression)
2:      call ← INIT(statement)
3:      call.name ← statement.name
4:      call.returnType ← statement.return
5:      call.root ← getHelperContext(statement)
6:      call.parameters ← getHelperParameters(statement)
7:      contextIndex → put(call.name, call.returnType)
8:
9:      firstStatement ← GETASTNODEBYTAG(source, "Body")
10:     GETFOOTPRINTS(firstStatement, call, append, call.context)
11:     expression.stack → add(call)
```

In order to analyze a Source node that represents the use of a *helper rule*, `AnalyzeCall()` creates a *call* with its corresponding *root context*, *returnType*, and list of *parameters* (Algorithm 4, line 2-6). It then extracts the Source nodes corresponding to its syntax sub-tree (Figure 5.2).

As mentioned before, binding expressions inside *helper rules* have the same syntax tree as those in conventional *matched rules*. Thus, `AnalyzeCall()` uses the `GetFootprints()` func-

[2]This case is further analyzed by the `SolveVarReference()` function which can be found at `https://github.com/guana/ct/tree/master/atl`.

tion to obtain the *footprints* of the helper's binding expressions (Algorithm 4, line 9). Both `GetFootprints()` and `AnalyzeCall()` implement a tail-recursion strategy. In effect, the `expression` parameter is used to add potential nested *calls* to the *stack* during backtracking (Algorithm 4 line 10-11). To complete the *binding analysis step*, `AnalyzeCall()` stores the return type of each *helper rule* using a context index (Algorithm 4, line 7). The context index is used in the third step of our analysis to solve the context switches of a binding expression due to *helper rule* calls.

5.2.3 Binding Tuple Individualization (M2M)

The main goal of this step is to use the *aggregated footprints* of a transformation in order to identify the binding tuples of its binding expressions (Chapter 4). We use the transformation's source metamodel to determine the elements that own every source attribute navigated by the expressions, and export their corresponding traceability links. In our analysis technique, we assume that the metamodels used by a transformation conform to the Ecore meta-metamodel (Figure 5.6). However, it can be generalized to other metamodeling formalisms such as KM3 [153].

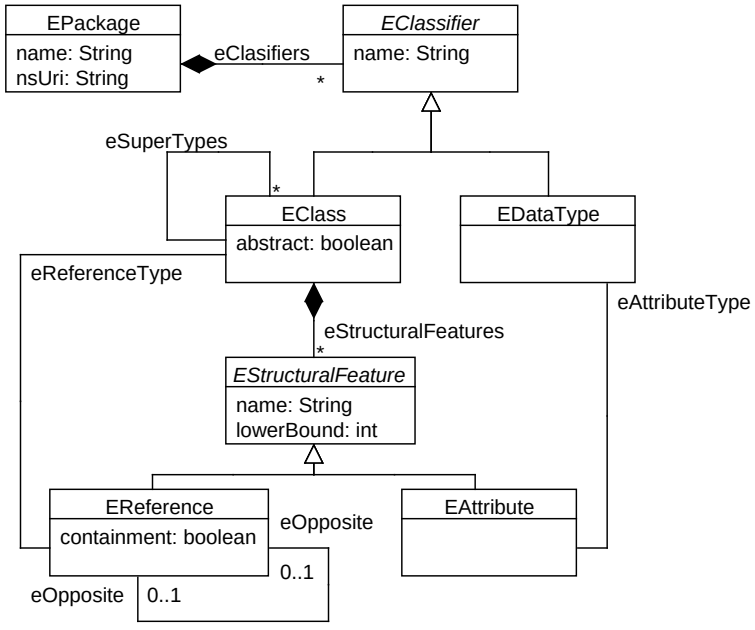


Figure 5.6: Simplified Ecore Metamodel [154]

The main function of our individualization step is `IndividualizeM2MBindingTuples()` (Algorithm 5). It receives as a parameter the *aggregated footprints* of a transformation. This function traverses the *rules* of a transformation, along with their corresponding *target elements* (te) and associated *expressions* (exs).

Algorithm 5

Require: *contextIndex* - return types of analyzed helpers.

```
1:  procedure INDIVIDUALIZEM2MBINDINGTUPLES(rules)
2:      for all rule ∈ rules do
3:          for all te ∈ rule.targets do
4:              for all ex ∈ te.exs do
5:                  SOLVEIMPLICITBINDINGS(rule, te, ex, ex.root)
6:
7:  procedure SOLVEIMPLICITBINDINGS(rule, te, expression, root)
8:      i ← 0
9:      step ← expression.footprints[i]
10:     contiguous ← step
11:     while step ≠ null do
12:         if step → CONTAINS(#) then
13:             root ← contextIndex → GET(step)
14:             contiguous ← empty
15:         else
16:             se ← GETOWNERELEMENT(contiguous, root, 0)
17:             INITM2MTRACEABILITYLINK(rule, se, step, te, expression.value)
18:         i ← i+1
19:         step ← expression.footprints[i]
20:         contiguous → APPEND(step)
21:     for all call ∈ expression.stack do
22:         SOLVEIMPLICITBINDINGS(rule, te, call, call.root)
```

The `SolveImplicitBindings()` function traverses the *footprints* of an *expression* and analyzes each one of its navigation *steps* (Algorithm 5, line 9). It determines whether a *step* is a *source attribute* (Algorithm 5, line 15) or a call to a *helper rule* (Algorithm 5, line 12). In the former case, the `GetOwnerElement()` function is used to obtain the *source element* (se) to which a navigated *source attribute* belongs (Algorithm 5, line 16). It uses the root context of the *expression* as the entry-point to prune the transformation's source metamodel, in order to find the element that contains the *source attribute* of interest (Figure 5.6 and Algorithm 6, lines 7-24). If `GetOwnerElement()` is unable to find the navigated *source attribute* in the list of EStructuralFeatures corresponding to its metamodel-root context, it calls itself recursively to continue the search in its super classes (Algorithm 6, lines 21-23).

As a concrete example, in the case of the *footprints* that describe the navigation path of the binding expression `b.chapters.first().digest.content` (Listing 4.1, line 21), i.e., *chapters.digest.content* (Figure 5.5), the `GetOwnerElement()` function is called once per navigation step with the following arguments.

Algorithm 6

Require: *metamodel* - root node of the source metamodel XMI.

```
1:  procedure GETOWNERELEMENT(path, root, stepIndex)
2:      classes ← GETASTNODEBYTYPE(metamodel, "EClass")
3:      for all class ∈ classes do
4:          if class == root then
5:              i ← stepIndex
6:              step ← path[i]
7:              while step ≠ null do
8:                  feats ← GETASTNODEBYTAG(class, "EStrcuturalFeature")
9:                  for all feature ∈ feats do
10:                     if feature is a "ERefference" then
11:                         if feature.type == step then
12:                             if i == path.length-1 then
13:                                 return root
14:                             else
15:                                 root ← INIT(reference.type)
16:                                 i ← i+1
17:                                 step ← path[i]
18:                     else if feature is a "EAttribute" then
19:                         if feature.name == step then
20:                             return root
21:                 super ← GETASTNODEBYTAG(class, "eSuperTypes")
22:                 for all parent ∈ super do
23:                     return GETOWNERELEMENT(path, parent, i)
24:                 return null
```

- Step 0: **contiguous**:= chapters, **root**:= Book (**return** → Book)
- Step 1: **contiguous**:= chapters.digest, **root**:= Book (**return** → Chapter)
- Step 2: **contiguous**: chapters.digest.content, **root**:= Book (**return** → Summary)

Please notice how the `contiguous` variable is used to store metamodel-navigation paths that share a common metamodel-root context (Algorithm 5, line 20). In the latter case, if a helper rule is found in the *footprints* of an *expression* (Algorithm 5, line 12) the `contextIndex` is used in order to update the metamodel-root context for the remaining steps under analysis (Algorithm 5, line 13). In this case, the `contiguous` variable is flushed out since the current metamodel-root context could have changed (Algorithm 5, line 14). In turn, the `SolveImplicitBindings()` function is recursively called to process the *calls* contained in the expression's *stack* (Figure 5.4 and Algorithm 5, line 21). Finally, the function `InitM2MTraceabilityLink()` (Algorithm 5, line 17) exports a traceability link in a well-formed JSON (Appendix A.3) following the formalism presented in Chapter 4.

5.3 A Traceability Analysis for M2T Transformations (Acceleo)

In this section we present our traceability-analysis technique for Acceleo M2T transformations (Figure 5.7). The technique involves five steps: (i) *binding expression analysis*, (ii) *binding tuple individualization*, (iii) *template traceability injection*, (iv) *transformation execution*, and (v) *generation link recovery*.

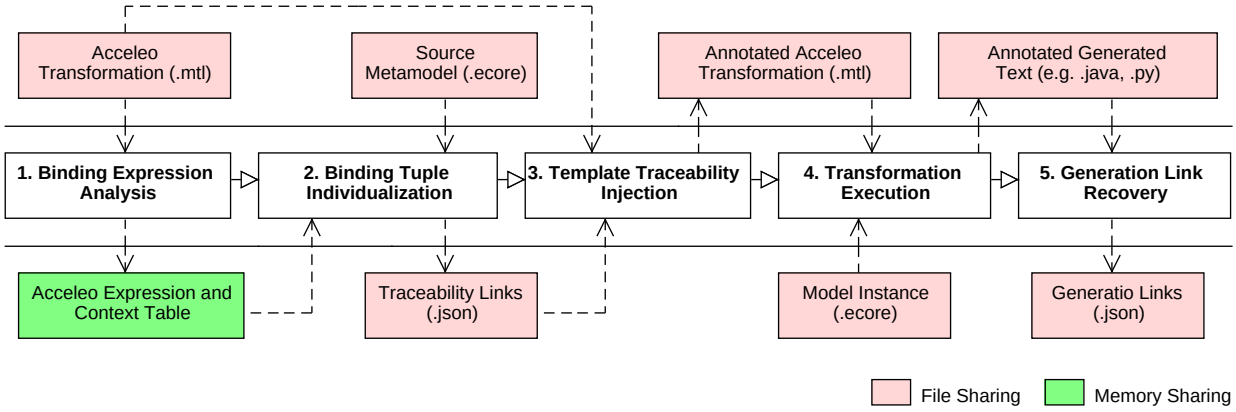


Figure 5.7: Acceleo Traceability Analysis Technique

The first step uses a custom parser in order to obtain a list of binding expressions from a template under analysis. In the second step, each expression is analyzed using a process similar to the one presented in Section 5.2. The last three steps of our technique identify *generation dependencies* between a template, and the lines of text generated after its execution. In order to do so, we follow a strategy similar to Olsen, et al. [103]; templates are annotated with traceability information and executed; the generated text is analyzed to identify its *generation dependencies*.

5.3.1 Binding Expression Analysis (M2T)

In Acceleo, there are three types of binding expressions, i.e., *loops, conditionals* and *simple expressions*. This step uses a programmatic tokenizing technique that classifies M2T binding expressions depending on their type. It considers spaces, line breaks, and other Acceleo non-terminal symbols in order to isolate their *metamodel footprints*[3].

Similarly to our *binding analysis* for M2M transformations, this step captures the *aggregated footprints* of a M2T transformation using a metamodel (Figure 5.8), and produces a context table with the metamodel-root contexts available to the expressions in the different segments of a template.

[3] A complete implementation of our tokenization algorithm can be found at `https://github.com/guana/ct/tree/master/acceleo`.

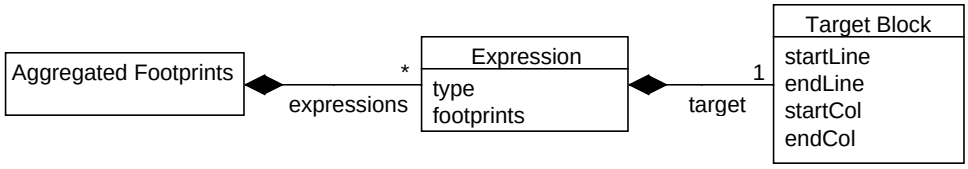


Figure 5.8: M2T Aggregated Footprints

```
1 [module generateIndex('http://ualberta.ssrg.publication')]
2 [template public generateIndex(aDatabase : Database)]
3 [comment @main/]
4 [file ('index.html', false, 'UTF-8')]
5 <h2> Publication Database </h2>
6   <table border="" style="with:50%">
7       [for (p : Publication | publications)]
8       <tr>
9           <td>
10            <p><h4>[p.title/]</h4></p>
11            <p>[p.prologue/]</p>
12            [if (p.prologue.size()> 1000)]
13              <p>Pages: [p.nbPages-p.prologue.size()/200/]</p>
14            [else]
15              <p>Pages: [p.nbPages/]</p>
16            [/if]
17          </td>
18      </tr>
19      [/for]
20  </table>
21 [/file]
22 [/template]
```

Listing 5.1: Publication2HTML (Conditional Version)

In Acceleo, the metamodel-root context of an expression can be defined by means of iterator variables found in *loop* expressions. The context of an expression can also be defined by means of global variables found in the header of a transformation template. In order to understand how metamodel contexts are defined in M2T transformations, let us briefly discuss a modified version of Publication2HTML (Listing 5.1). We will use this example to illustrate the process of collecting the traceability links and generation dependencies in M2T transformations with *loop* and *conditional* binding expressions. In this version of Publication2HTML, the reported page count of a Publication is recalculated depending on the length of its *prologue*. If the *prologue* of a Publication contains more than 1000 characters, its total page count will be reduced by a factor of one for every two hundred of those characters (Listing 5.1, lines 12-16)

Table 5.1 presents the metamodel-root contexts defined in Publication2HTML. The header of the transformation (Listing 5.1, line 2) defines the global-root context for all the expressions in the transformation, i.e., Database. Any binding expression might use the Database element as the entry-point for its metamodel-navigation statements. The expression `for (p : Publication`

`| publications)` (Listing 5.1, line 7), defines an additional metamodel-root context, i.e., Publication, which can be used by any expression declared inside of the *loop* (Listing 5.1, lines 7-19). Figure 5.9 presents the *footprints* of the binding expressions in Publication2HTML. For readability purposes, we have not included the start and end column of the expressions.

Start Line	End Line	Context	Variable
2	22	Database	aDatabase
7	19	Publication	p

Table 5.1: Publication2HTML - Metamodel Context Table

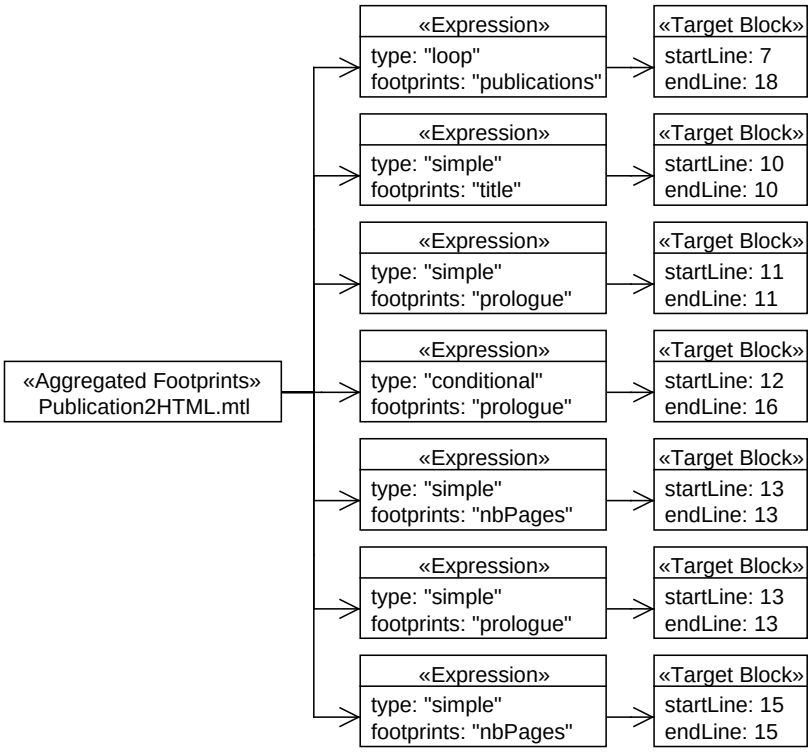


Figure 5.9: M2T Aggregated Footprints Instance - Publication2HTML

5.3.2 Binding Tuple Individualization (M2T)

Given that binding expressions in different template locations might have access to multiple metamodel-root contexts, it is necessary to precisely pinpoint the context element used by each binding expression. This is particularly important when analyzing transformations that rely on nested loops and conditionals.

Algorithm 7

```
1:  procedure INDIVIDUALIZEM2TBINDINGTUPLES(expressions, contexts)
2:      for all exp ∈ expressions do
3:          i ← 0
4:          step ← exp.footprints[i]
5:          contiguous ← step
6:          while step ≠ null do
7:              if ¬ ISCONTEXTVARIABLE(step) then
8:                  for all ctx ∈ contexts do
9:                      sourceElem ← GETOWNER(contiguous, ctx)
10:                     if se ! = null then
11:                         sourceAttr ← INIT(step)
12:                         INITM2TTRACEABILITYLINK(sourceElem, sourceAttr, exp)
13:                         break
14:                 i ← i+1
15:                 step ← exp.footprints[i]
16:                 contiguous → APPEND(step)
```

The function `IndividualizeM2TBindingTuples()` (Algorithm 7) initializes a template's traceability links considering the binding tuples existing in its *footprints*. It receives as a parameter the *aggregated footprints* of a template and its contexts table. The function navigates each *step* in the *footprints* of an *expression* in order to identify its context element (Algorithm 7, line 9). `IndividualizeM2TBindingTuples()` also uses the `GetOwnerElement()` function (Section 5.2) in order to identify the elements that own the source attributes in the *footprints* of a binding expression. In this case, if `GetOwnerElement()` returns a valid metamodel element with any of the elements in the context table, a M2T traceability link is initialized using the `InitM2TTraceabilityLink()` function (Algorithm 7, line 12). In the case of *loop* and *conditional* expressions, `InitM2TTraceabilityLink()` will export a traceability link for each of their contained lines. Our implementation exports traceability links in a JSON file (Appendix A.3).

5.3.3 Template Traceability Injection and Execution

The last three steps of our analysis technique are designed to obtain the dependency relationships between a M2T transformation, and the files generated after its execution. We call these dependencies *generation links*. To do so, a M2T transformation is annotated with its *aggregated footprints*. In the annotation process, each binding expression is surrounded by specially-formatted identifiers that will be transfered to the generated files during execution. We have developed a customized Acceleo Launcher[4] capable of (a) injecting *aggregated footprints* annotations, (b) compiling and executing transformations, and (c) removing annotations after they have been analyzed. This launcher allows developers to collect *generation links* in a seamless manner.

[4] `https://github.com/guana/ct/tree/master/accelo`

Figure 5.10-A presents the Publication2HTML transformation with its *footprints* annotations. Figure 5.10-B portrays the HTML report generated after its execution. In this example, a simple input model has been provided in order to execute the transformation. It contains two publications, namely, *Thinking Python* (Figure 5.10 (b), lines 4-18) and *Head First Design Patterns* (Figure 5.10b), lines 29-33). Figure 5.10-C depicts the generated report as seen in a web browser.

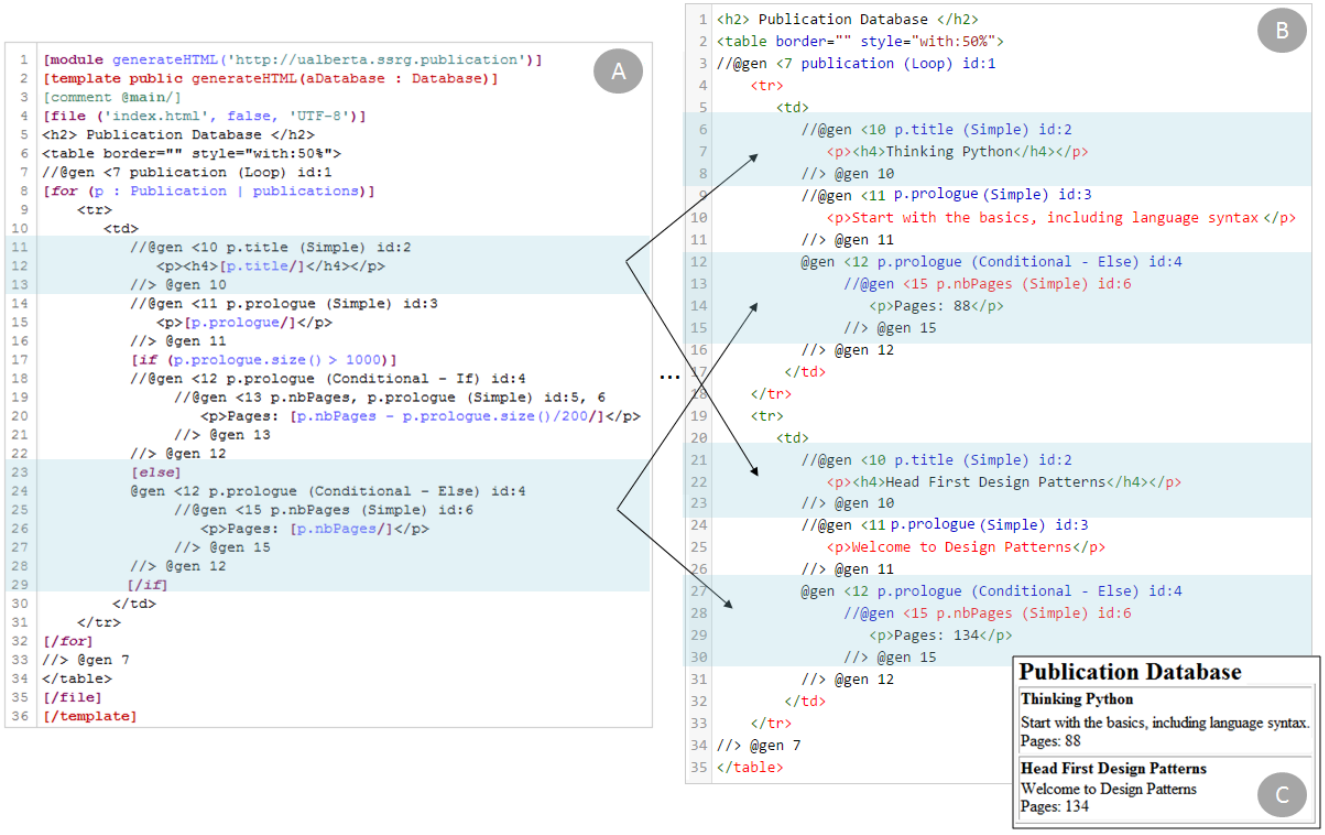


Figure 5.10: (A) Annotated Publication2HTML, (B) HTML Report Generation Instance, (C) HTML Report Web Browser View

An identifier is comprised of an opening statement, i.e., `//@gen <template-line, binding-footprints, binding-type, binding-id>` and a closing statement, i.e., `//> @gen template-line`. In the case of a *simple* binding expression, a traceability identifier is injected surrounding the line where it is defined (e.g., Figure 5.10 (a), lines 11-13). In the case of lines with multiple binding expressions, a single identifier is used to capture multiple *generation links* (e.g., Figure 5.10 (a), line 19). In the case of *loop* expressions, an identifier is injected before and after their opening and closing statements (e.g., Figure 5.10 (a), lines 7 and 33).

In Acceleo, there are three types of *conditional* expressions; namely, *if, if/else*, and *if/else-if*. In the case of an *if conditional*, identifiers are injected immediately after its opening line, and immediately before its closing line. Indeed, only expressions in conditional blocks whose invariants

are satisfied during execution are effectively binded. *if/else conditionals* are annotated following a similar strategy. In this case, the identifier used to annotate an *if* block is used to annotate its corresponding *else* block (e.g., Figure 5.10 (a), lines 18-22 and 24-28). Expressions in both conditional blocks are alternatively executed, thus creating a *generation link* with the same line in a transformation template. Finally, in the case of *if/else-if conditionals*, an identifier is included in the *else* block in order to capture the *generation links* that stem from the bindings in its alternative conditional invariant.

5.3.4 Generation Link Recovery

Algorithm 8 presents the function that drives our generation-link recovery process, i.e., `Collect GenerationLinks()`. It analyzes the lines of generated files looking for *footprints* annotations. When an opening statement is identified, an annotation is initialized and indexed in the `annotation` table (Algorithm 8, lines 5-8). When its corresponding closing statement is found, the annotation is removed (Algorithm 8, lines 9-12). If a generated line is found, a *generation link* is created for all of the open annotations in the `annotation` table (Algorithm 8, lines 13-15). It is important to mention that the number of generated lines is recalculated as annotations are found (Algorithm 8, lines 8 and 12). This is required since generated files will be striped out of annotations after being analyzed. The function `InitGenerationLink()` will then export a *generation link* for each of the binding tuples found in the annotation *footprints*. Similarly to M2M and M2T traceability links, our current implementation exports *generation links* using JSON.

Algorithm 8

```
1:  procedure COLLECTGENERATIONLINKS(generatedFiles)
2:      for all file ∈ generatedFiles do
3:          genLine ← 0
4:          for all line ∈ file do
5:              if line ← CONTAINS(//@gen <) then
6:                  annotation ← INITANNOTATION(line, file)
7:                  open ← PUT(annotation.line, annotation)
8:                  genLine ← genLine - 1
9:              else if line ← CONTAINS(// > @gen) then
10:                 annotation ← INITANNOTATION(line)
11:                 open ← REMOVE(annotation.line)
12:                 genLine ← genLine - 1
13:             else
14:                 for all note ∈ annotation do
15:                     INITGENERATIONLINK(genLine, note)
16:             genLine ← genLine + 1
```

In Chapter 6, we discuss ChainTracker. An integrated traceability-analysis environment that makes traceability information available to developers. ChainTracker identifies and visualizes the *dependent traceability links* of a model-transformation chain based on the traceability links identified in its individual M2M and M2T transformations.

5.4 Traceability-Analysis Evaluation

We evaluated our traceability-analysis technique using 25 *model-to-model* transformations, and 18 *model-to-text* transformations. The former correspond to the 14 Ecore-based transformation projects available in the *ATLZoo* (Table 5.2). The latter have been obtained from the *Acceleo Example Repository*[5] and correspond to 5 individual code-generation projects (Table 5.4).

All transformations were manually inspected to characterize their binding expressions and corresponding binding tuples. The inspection process was conducted by two software engineering researchers (the author and a colleague) with 6 and 2 years of experience in model-transformation technologies, respectively. Researchers examined the transformations separately. Next, they had a final inspection meeting in order to resolve characterization discrepancies.

5.4.1 Evaluation: Model-to-Model Traceability Analysis (ATL)

Table 5.3 presents the traceability coverage of our analysis technique for M2M transformations. It presents the number of *implicit* and *explicit* binding tuples in the 25 ATL transformations under analysis. The traceability coverage reflects the number of traceability links that were successfully collected out of all the links found in their comprising binding expressions, i.e., recall.

Our traceability-analysis technique for M2M transformations obtained an average coverage of 91%. We observed a traceability coverage of 100% in 13 of 25 transformations.

27.4% of the M2M binding tuples in the evaluation dataset are *implicit*. Transformations are predominantly comprised by simple binding expressions (43%), i.e., expressions that do not rely on iterative expressions, e.g., `select()` and `collect()`. Only 2% of binding tuples were found in iterative expressions. 6% of binding expressions use *helper rules* (Table 5.2).

[5] `https://github.com/eclipse/acceleo/tree/master/examples`

Table 5.2: ATLZoo (Ecore-based) Transformations - Expression Metrics

Project	Transformation	Simple Exp.	Iterator Exp.	Helper Exp.	Helper Att.	Lazy Rules	Target E. Cross-Ref.	Using Exp.	Resolve Temp	Constant Exp.	Action Blocks
ATL2BindingDebugger	ATL2BindingDebugger	9	0	0	0	0	2	0	0	1	0
ATL2Problem	ATL-WFR	33	0	0	0	0	0	0	0	18	0
CPL2SPL	CPL2SPL	12	1	0	0	3	32	0	0	15	0
DSLBridge	DSL2KM3	34	0	3	0	0	3	0	0	8	0
	KM32DSL	31	0	18	4	0	6	0	0	19	0
Grafcet2PetriNet	Grafcet2PetriNet	24	0	0	0	0	0	0	0	0	0
	PetriNet2PNML	16	0	0	0	0	11	0	2	0	0
	PNML2XML	6	12	0	0	0	28	0	3	23	0
OWL2XML	OWL2XML	2	1	28	0	2	23	0	0	60	0
SimpleClass2RDBMS	SimpleClass2RDBMS	1	0	0	0	0	3	8	0	0	0
SoftwareQuality2Bugzilla	Bugzilla2XML	30	0	0	0	0	30	13	0	42	0
	SoftwareQuality2Bugzilla	13	0	0	0	0	1	0	0	21	0
SoftwareQuality2Mantis	Mantis2XML	31	0	0	0	0	41	19	0	61	0
	SoftwareQuality2Mantis	7	0	1	0	0	9	1	0	31	0
Table2MicrosoftExcel	SpreadsheetML2XML	4	0	1	0	0	2	1	0	24	0
	Table2SpreadsheetML	0	0	0	0	0	6	2	0	3	0
TruthTable2DecisionTree	TT2BDD	8	0	1	0	0	0	2	0	0	0
VisualRepCodeClone	CloneDr2Codelone	8	0	1	0	0	0	0	0	1	0
	CodeClone2SVG	13	0	0	0	0	7	0	0	43	0
	Simian2CodeClone	8	0	1	0	0	0	0	0	1	0
	SVG2XML	8	0	0	0	0	7	0	0	11	8
WSDL2R2ML	R2ML2WSDL	19	0	0	0	0	14	0	0	5	0
	WSDL2R2ML	16	0	0	0	1	1	0	0	2	0
XSLT2XQuery	XSL2XML	31	0	0	0	0	0	0	0	2	0
	XSLT2XQuery	12	0	0	0	0	14	0	0	6	0

Table 5.3: ATLZoo (Ecore-based) Transformations - Traceability Analysis Metrics

Project	Transformation	Explicit Bindings	Implicit Bindings	Total Bindings	Bindings Succesfully Analyzed	Bindings Missed	Coverage %
ATL2BindingDebugger	ATL2BindingDebugger	11	4	15	15	0	100.0
ATL2Problem	ATL-WFR	38	9	47	42	5	89.4
CPL2SPL	CPL2SPL	29	9	38	30	8	78.9
DSLBridge	DSL2KM3	44	19	63	63	0	100.0
	KM32DSL	81	17	98	85	13	86.7
Grafcet2PetriNet	Grafcet2PetriNet	24	0	24	24	0	100.0
	PetriNet2PNML	17	0	17	15	2	88.2
	PNML2XML	18	30	48	48	0	100.0
OWL2XML	OWL2XML	76	29	105	105	0	100.0
SimpleClass2RDBMS	SimpleClass2RDBMS	1	0	1	1	0	100.0
SoftwareQuality2Bugzilla	Bugzilla2XML	36	0	36	36	0	100.0
	SoftwareQuality2Bugzilla	15	6	21	21	0	100.0
SoftwareQuality2Mantis	Mantis2XML	35	16	51	51	0	100.0
	SoftwareQuality2Mantis	10	4	14	13	1	92.9
Table2MicrosoftExcel	SpreadsheetML2XML	26	0	26	26	0	100.0
	Table2SpreadsheetML	2	0	2	2	0	100.0
TruthTable2DecisionTree	TT2BDD	8	18	26	9	17	34.6
VisualRepCodeClone	CloneDr2Codelone	9	1	10	6	4	60.0
	CodeClone2SVG	29	18	47	47	0	100.0
	Simian2CodeClone	9	1	10	6	4	60.0
	SVG2XML	17	0	17	17	0	100.0
WSDL2R2ML	R2ML2WSDL	47	10	57	42	1	73.7
	WSDL2R2ML	26	30	56	34	22	60.7
XSLT2XQuery	XSL2XML	29	16	45	45	0	100.0
	XSLT2XQuery	11	2	13	12	1	92.3

Less than 5% of binding tuples are defined in imperative expressions. Only 6 *lazy rules* and 8 action blocks were found in the 25 transformations that we studied. It is important to note that the use of *lazy rules* and action blocks is highly discouraged by the *ATL Developer Guide*[6]. Finally, 35% of binding expressions contain constant values that are directly assigned to target attributes. This is mainly observed in transformations that manipulate structured-textual files, e.g., Bugzilla2XML, Mantis2XML, and PNML2XML, and in transformations used to digest communication protocols, e.g., CPL2SPL (Table 5.2). This binding strategy is widely recognized as a transformation anti-pattern, i.e., "magic literals" [155].

Limitations

Let us now discuss the limitations of our analysis technique, and the (M2M) transformations where it obtained a subpar traceability coverage.

1. Downcasting binding expressions: In order to deal with metamodels with complex inheritance hierarchies, *helper rules* can be called using metamodel elements that extend their context type. In this pattern, binding expressions perform type-checks, i.e., `oclIsKindOf()` or `oclIsTypeOf()` to access attributes only available to the rule's context and parameter sub-types.

Our current implementation considers binding expressions that use attributes inherited from a rule's context element super-type (Algorithm 6), and it is unable to analyze the *footprints* of downcasting expressions, i.e., binding expressions that use attributes defined in its context or parameter subtypes [155]. This binding pattern was found 17 times in TruthTable2BinaryDecisionTree, a transformation used to simplify the design of digital logic circuits. In this transformation, our analysis technique obtained a traceability coverage of 34.6% (Table 5.3). Downcasting expressions are rare and highly discouraged [94, 155].

2. Cross-referenced target elements: In ATL, *matched rules* can produce multiple target elements from a source element. A target element te_a is cross-referenced with a target element te_b if they are both initialized by the same *matched rule*, and the binding expressions used to initialize te_a, use te_b as a metamodel-root context. Our analysis technique assumes that the *aggregated footprints* of a *matched rule* share a common metamodel-root context (Figure 5.4). This precondition is not guaranteed in binding expressions with cross-referenced target elements. In CPL2SPL, a transformation used to translate telephony control protocols, 8 out of 38 bindings contain cross-referenced target elements; the traceability coverage of our analysis in this case was 79%. This limitation was also observed in the analysis of PetriNet2PNML and R2ML2WSDL, where our analysis technique obtained a traceability coverage of 88% and 73.7%, respectively (Table 5.3).

[6] `https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`

3. Context-less Helper Attributes: In ATL, *helper attributes* can be seen as *helper rules* without parameters. They are commonly used to define context-dependent constants. The context of a *helper attribute* can be omitted in order to make it globally accessible to the expressions of a transformation, regardless of their scope. Our analysis technique does not guarantee the correct individualization of binding tuples in context-less *helper attributes*. In fact, 3 *context-less helper attributes* were found in KM32DSL (Table 5.2). This transformation also contains 6 cross-referenced target attributes. In this case, the presence of both types of binding expressions limited the traceability coverage of our technique to 86.7% (Table 5.3).

4. Local Variables in Matched Rules: In ATL, a *matched rule* can optionally include a *using* section in order to define local variables. In the 25 transformations under analysis, local variables were used to flatten source attributes with high multiplicity, into collections of primitive types, i.e., SimpleClass2SimpleRDBMS (8 local variables, Table 5.2), and to initialize default attributes in malformed source elements, i.e., Mantis2XML (19 local variables, Table 5.2). Even though our analysis technique considers local variables defined in *helper rules*, it does not consider those defined in *matched rules*. The use of local variables in *matched rules* is relatively rare in our evaluation dataset. However, we believe they can be highly useful to practitioners and researchers alike. The ability of collecting traceability links from local variables defined outside *helper rules* will be included in our future work.

5.4.2 Evaluation: Model-to-Text Traceability Analysis (Acceleo)

The 18 M2T transformations used in the evaluation of our traceability-analysis technique correspond to the 5 code-generation projects available in the *Acceleo Example Repository*:

1. *AcceleoAndroid - Obeo:* a code generator for the construction of multi-layered Android applications.
2. *Ecore2Python:* a code generator for the rapid prototyping of object-oriented architectures in Python.
3. *Ecore2UnitTests:* a code generator to support the the orchestration of test suits built-on JUnit.
4. *UML2JAVA,* and *UML2Java - Helios*: two code generators for the construction of Java applications based on high-level specifications.

Table 5.4 presents the characterization of the binding expressions in our M2T evaluation dataset. Table 5.5 presents the traceability coverage of our traceability-analysis technique for M2T transformations.

Our traceability-analysis technique for M2T transformations obtained an average coverage of 85.4%.

Table 5.4: Acceleo Transformations - Expression Metrics

Project	Transformation	Simple Exp.	Loop Exp.	Conditional Exp.	Template Queries
AcceleoAndroid - Obeo	AndroidManifestXML	2	0	0	0
	DBadapter	28	0	0	2
	Edit	34	0	0	1
	EditXML	12	0	0	0
	List	17	0	0	0
	ListXML	1	0	0	0
	ListRowXML	2	0	0	0
	StringsXML	3	0	0	0
Ecore2Python	Factory	13	4	0	2
	Init	16	3	1	7
Ecore2UnitTests	AbstractTestClass	1	0	0	1
	CommonFiles	3	0	0	0
	EnumerationTest	17	0	0	1
UML2Java	ClassJavaFile	25	6	6	13
	EnumJavaFile	4	1	0	4
	InterfaceJavaFile	9	4	2	5
UML2Java - Helios	ClassBody	33	4	6	12
	InterfaceBody	2	3	1	4

Table 5.5: Acceleo Traceability Analysis - Metrics

Project	Transformation	Explicit Bindings	Implicit Bindings	Total Bindings	Bindings Collected	Bindings Missed	% Collected
AcceleoAndroidExample	AndroidManifestXML	2	0	2	2	0	100.0
	DBAdapter	29	0	29	28	1	96.6
	Edit	35	0	35	34	1	97.1
	EditXML	12	0	12	12	0	100.0
	List	17	0	17	16	1	94.1
	ListXML	1	0	1	1	0	100.0
	ListRowXML	2	0	2	2	0	100.0
	StringsXML	3	0	3	3	0	100.0
Ecore2Python	Factory	17	4	21	17	4	81.0
	Init	22	5	27	26	1	96.3
Ecore2UnitTests	AbstractTestClass	2	0	2	1	1	50.0
	CommonFiles	3	0	3	3	0	100.0
	EnumerationTest	18	18	36	17	19	47.2
UML2Java	ClassJavaFile	42	6	48	30	18	62.5
	EnumJavaFile	5	0	5	5	0	100.0
	InterfaceJavaFile	17	2	19	16	3	84.2
UML2Java - Helios	ClassBody	48	9	57	30	27	52.6
	InterfaceBody	7	1	8	6	2	75.0

In our evaluation dataset, 13% of the M2T binding tuples are *implicit*. Similarly to M2M transformations, 66% of M2T binding expressions in our evaluation dataset are simple. 7% of all binding tuples were found in loop expressions, and 5% in conditionals. Finally, 18% of the binding expressions use *template queries*.

Limitations

Let us now discuss the limitations of our analysis technique for M2T transformations and the cases where it obtained a low traceability coverage.

1. Template Queries: Similarly to *helper rules* in ATL, Acceleo enables developers to define *template queries* in order to encapsulate factorized binding expressions. Our current implementation does not include a mechanism to gather traceability links from *template queries*. The main challenge behind the analysis of *template queries* is to determine their location within a transformation codebase. This requires a static analysis technique to interpret the import statements in the header of a transformation template, in order to in-line their corresponding binding expressions. Transformations such as *Factory* in Ecore2Python, *ClassJavaFile* in UML2Java, *InterfaceBody* in UML2Java, and *EnumerationTest* in Ecore2Python, heavily rely on utility templates that are exclusively comprised of *template queries*. This effectively limited the traceability coverage of our technique to 81%, 62%, 75%, and 42%, respectively (Table 5.5).

2. Template Cross-referencing: Cross-referenced templates import other templates, and use their generation capabilities from outlying sections of code. Our current analysis technique does not analyze *cross-referenced templates*, as they pose similar analysis challenges to *template queries*. As a concrete example, *Init* in Ecore2Python includes multiple cross-referenced templates, which reduced the traceability coverage of our analysis technique, i.e., 96%.

5.5 Threats to Validity

5.5.1 Internal Validity

Are there unknown factors which might affect the outcome of the evaluation? – In order to evaluate our traceability-analysis technique we measured its traceability coverage in the context of 25 ATL M2M transformations, and 18 M2T transformations. The transformations used in our evaluation were manually inspected by two researchers with 2 and 6 years of experience. The main goal of the inspection process was to characterize the binding expressions of the transformations and their corresponding traceability links.

In order to mitigate potential errors, the two researchers inspected the transformations individually. In turn, a conflict resolution meeting was conducted in order to examine categorization disagreements. Researchers compared the traceability links collected by our analysis tool, with those obtained in the manual inspection process. The traceability coverage of each transformation was evaluated by both researchers independently. Their results were consolidated in a final inspection meeting. We do not see any significant threats to the internal validity of our evaluation.

5.5.2 External Validity

To what extent is it possible to generalize the findings? – The limited number of model transformations used to measure the coverage of our analysis technique is a concern for the external validity of our evaluation. The 25 ATL M2M transformations were obtained from the *ATLZoo*, a research-oriented transformation repository. Furthermore, the 18 Acceleo M2T transformations were obtained from the *Acceleo Example Repository*. This repository is hosted by Obeo, the principal strategic partner of the Eclipse Foundation for model-transformation technologies. We can not claim that the results of our evaluation process can be generalized to other transformation languages. However, our conceptual framework is generalizable to any transformation language built-on OCL. Furthermore, Acceleo and ATL are aligned to the OMG's *Query/View/Transformation (QVT)* standard for M2M transformations [156], and the *Model to Text Transformation Language (MOF)* standard for M2T transformations [157], respectively. Therefore, the effectiveness of our analysis technique provides considerable insights regarding its extensibility to other relational and template-based transformation languages that comply with the same set of standards.

Given that the transformations used in our evaluation have only been used in a research environment, we can not claim that the results of our evaluation can be generalized to industrial transformation ecosystems. To the best of our knowledge, our traceability-analysis technique is the first of its kind to be evaluated using a non-trivial evaluation dataset. Considering that model-transformation languages are a fairly new technology, and are yet to be adopted by the software-engineering community at large, our work provides a first step towards increasing the evaluation rigor of metamodel-level traceability-analysis techniques.

6

ChainTracker

In this chapter, we present ChainTracker, a metamodel-level traceability-analysis environment for model transformations. ChainTracker implements the traceability-analysis technique presented in Chapter 5, and it adheres to the traceability conceptual framework presented in Chapter 4. The usability of ChainTracker is evaluated in Chapter 7.

6.1 The ChainTracker Analysis Environment

ChainTracker identifies and visualizes the *end-to-end traceability links* of a transformation chain based on the traceability links collected from individual M2M and M2T transformations. It takes as input one or multiple model transformations, along with their corresponding source and target metamodels, and optionally model instances that conform to these metamodels. The analysis environment considers traceability links as symbolic dependencies between three main artifacts, (a) a transformation's source metamodel, (b) a transformation's target metamodel, and (c) a transformation's binding expressions. Likewise, ChainTracker considers traceability in M2T transformations as symbolic dependencies between (a) a transformation's source metamodel, (b) a transformation's binding expressions, and (c) the lines of text generated after its execution.

A transformation ecosystem can be understood as implementation units with runtime behaviors, or as a collection of static elements that are bound to each other at development time. The ChainTracker analysis environment presents two views of a transformation ecosystem using an interactive multi-view approach. According to Clements et al. [158], a view can be understood as the *representation of a set of system elements and the relationships associated with them*. Each view defines the concrete and abstract syntaxes of the elements and relationships of the system. The goal of having multiple views for a software system is to enable developers to think about the

architecture of a software system in multiple ways simultaneously. In the context of model-driven engineering, the GEMOC Studio [159, 160] integrates different language workbenches by means of multiple views that represent their operational semantics at the model-level (Chapter 2). In [161], Soni et al. present the four types of views that allow developers to reflect on a system's architecture using complementary perspectives:

1. The *conceptual* view describes a system in terms of its design elements and relationships.
2. The *module* view presents a system as a set of implementation or functional units.
3. The *execution* view reflects on the the runtime behavior and interactions of a system.
4. The *code* view portrays how a system's implementation units relate to non-software elements in its environment.

ChainTracker proposes a visualization technique for the *module* and *execution* views of a transformation ecosystem. It enables developers to visualize the different artifacts that comprise an ecosystem and its execution mechanics (at the metamodel-level), as well as to use projectional code editors to simultaneously explore its underlying transformation codebases.

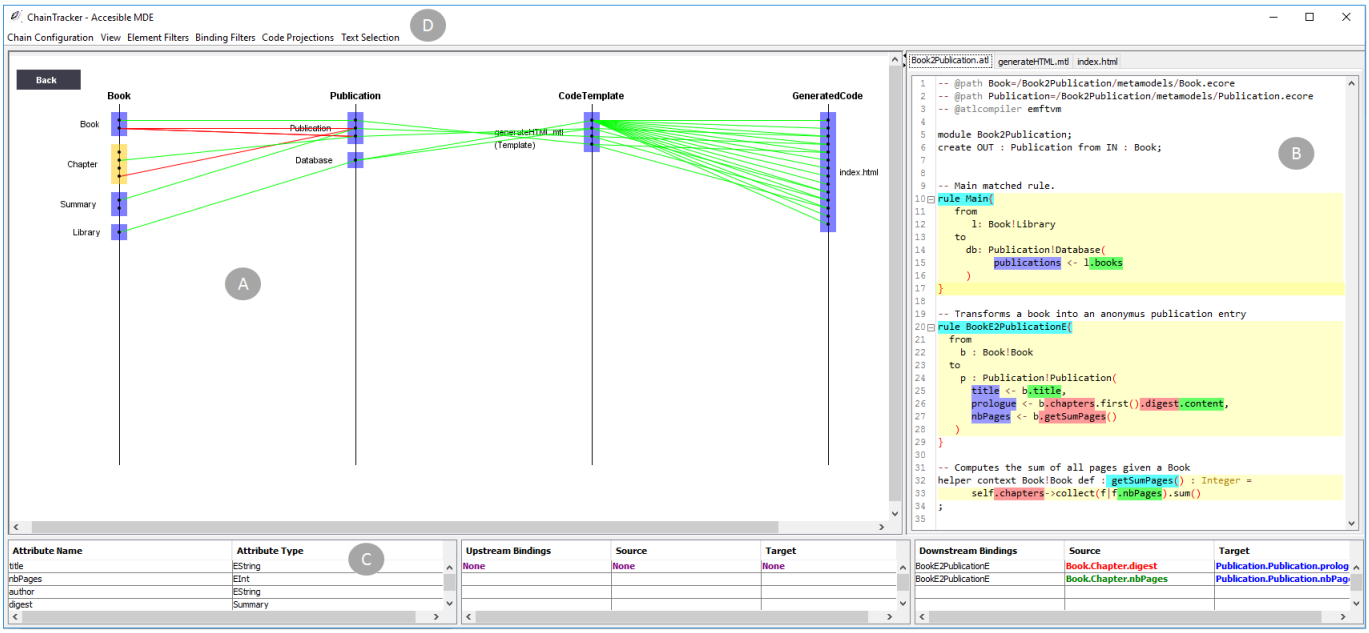


Figure 6.1: The ChainTracker Traceability Analysis Environment

The ChainTracker analysis environment is divided in three main areas: the *transformation visualizations*, the *projectional code editors*, and the *contextual tables* (Figure 6.1-A, B, and C, respectively). Each area of the analysis environment is synchronized with each other to provide

developers with an unified and context-aware experience. ChainTracker helps developers to investigate how source-metamodel elements and their attributes are transformed into different intermediate metamodels, and into final textual files. ChainTracker not only allows developers to explore the visualizations of M2M and M2T transformations, but also to project their information onto the analyzed codebases using text highlighters. Furthermore, if concrete model instances are provided to the analysis environment, ChainTracker is capable of executing transformation ecosystems, in order to include generated textual files in the traceability analysis and visualization process. Let us briefly discuss each one of the areas of the environment using the *Library to Anonymous Index* example presented in Chapter 4.

6.1.1 The Transformation Visualizations

ChainTracker provides two different types of transformation visualizations: the *overview visualization* (Figure 6.2) and the *branch visualization* (Figure 6.3). Developers can switch between visualization types using ChainTracker's command menu (Figure 6.1-D)

The ChainTracker Overview Visualization

The *overview visualization* presents a *module view* of the ecosystem under analysis. This view enables developers to abstract the complexity of individual and isolated transformation scripts, into a single picture that summarizes its compositional structure. It follows a graph-based approach in which blue nodes represent the source and target artifacts of a transformation step. In the case of M2M transformations, blue nodes portray source and target metamodels. In M2T transformations, blue nodes portray textual templates and generated textual artifacts such as code. Edges in this visualization diagrammatically depict dependencies between the steps of a transformation chain. As a concrete example, Figure 6.2 portrays the *overview visualization* of the *Library to Anonymous Index* example.

The *overview visualization* can be used to quickly obtain information about the order of precedence of the transformation in a complex transformation ecosystem. Developers can click on the edges of the visualization to obtain information boxes with details about the ecosystem's transformation scripts, code templates, and generated textual artifacts. For example, Figure 6.2-A presents information corresponding to the transformation rules contained in the Book2Publication transformation (Listing 4.1, line 5 and 15, respectively), Figure 6.2-B portrays the templates comprised of the Publicationt2HTML transformation (Listing 4.2, line 2), and Figure 6.2-C presents the list of the generated files derived after its execution, i.e., *index.html*.

ChainTracker automatically identifies the dependencies between transformation scripts and determines its order of precedence in the case of model-transformation chains. The *overview*

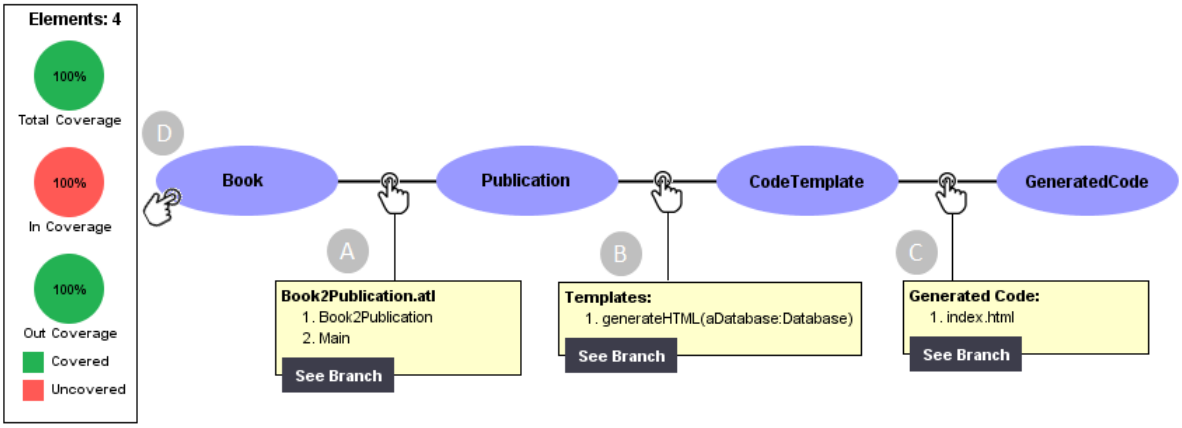


Figure 6.2: The ChainTracker Overview Visualization

visualization provides quick insights on the branching structure of a composition, by creating a graphical representation of its major implementation units. Summarizing information about the ecosystem's high-level structure enables developers to assess and, potentially, optimize its overall design and correctness [162].

In order to study how well the information captured by the metamodels of an ecosystem is used by its transformations, developers can click on the nodes of the overview visualization to determine their coverage throughout the different steps of a transformation chain (Figure 6.2-D). Coverage metrics provide insights about how well the information captured by a metamodel is used by the transformations of an ecosystem. In effect, coverage information is a vital component when reflecting about the quality of a transformation composition, or when assessing the impact of evolutionary changes. In ChainTracker, coverage metrics are summarized in contextual pie-charts that contain in- and out-coverage metrics. The in-coverage metric reports the percentage of elements in a metamodel that are effectively targeted by the bindings defined in the transformations of an ecosystem. The out-coverage metric represents the percentage of elements in a metamodel used by transformation bindings to either generate textual artifacts, or to derive intermediate models in a multi-step transformation chain. This information might lead developers to remove unused elements from a metamodel, or to take advantage of their semantic value and include them in the scope of a transformation. Maximizing metamodel coverage makes transformations less convoluted and less error prone while, at the same time, freeing metamodels from unused semantic constructs [22].

In the *Library to Anonymous Index* example, the out-coverage of the Publication metamodel is 100% (Figure 6.2-D). This means that all the elements of the Publication metamodel are used by the Bank2Report transformation. Conversely, the in-coverage metric for the same metamodel is 100%

uncovered since it is the root of the transformation chain, and no binding targets its elements.

The ChainTracker Branch Visualization

The *branch visualization* presents an *execution view* of the transformation ecosystem under analysis (Figure 6.3). The goal of the *branch visualization* is to present information about the fine-grained traceability links that exist throughout a transformation ecosystem. This visualization aims at portraying all the symbolic dependencies that individual binding expressions establish between metamodel elements and their properties, textual templates, and potential generated artifacts.

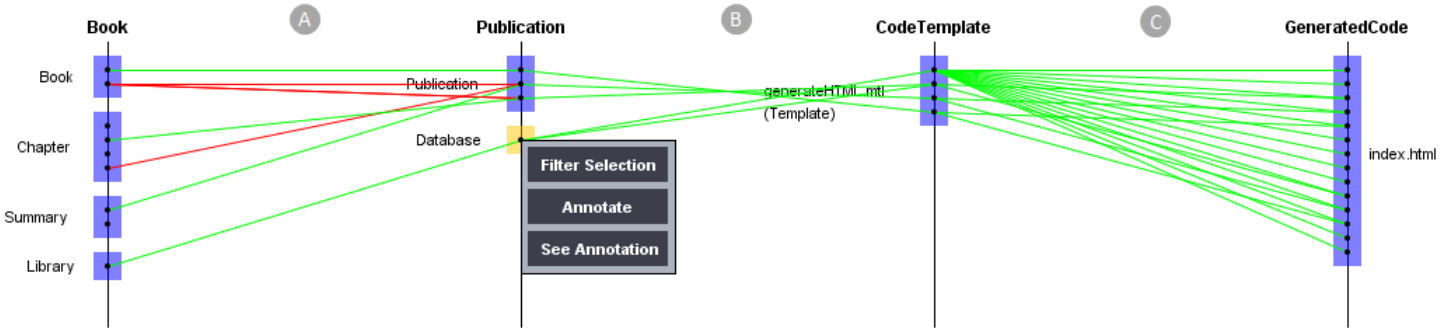


Figure 6.3: The ChainTracker Branch Visualization

The *branch visualization* follows a graphical notation inspired by parallel-coordinate visualizations for hyper-dimensional data [163]. Parallel coordinates have been widely used to represent the relationships between multidimensional datasets. They are commonly used to represent relationships between continuous numerical variables grouped in interdependent classes[1]. In the context of transformation ecosystems, the artifacts in each step of a transformation chain, including the resulting files obtained after its execution, can be considered as an interdependent set of categorical information. Each set is interconnected by means of transformation expressions that bind their comprising elements.

The ultimate purpose of a transformation chain is to reduce, split, merge, or augment the information provided as input to systematically transform it, into one or multiple output representations that conform to a textual or metamodel syntax. In transformation ecosystems that involve M2M and M2T transformations we can find at least three semantic dimensions in the transformation process: metamodels, textual templates, and generated textual artifacts. The *branch visualization* captures each step of a transformation composition and portrays its corresponding artifacts in individual

[1]Multiple examples of parallel coordinate visualizations are available at https://syntagmatic.github.io/parallel-coordinates/

coordinate dimensions.

Figure 6.3 portrays the *branch visualization* of the *Library to Anonymous Index* case study. Each semantic dimension of its underlying transformation chain is represented using vertical lines. Depending on the nature of each transformation step, its corresponding graphical notation contains a different set of artifacts. In the case of M2M transformations, vertical lines contain blue boxes that portray the elements of its source and target metamodels; black dots inside these boxes represent their corresponding properties (Figure 6.3, Book and Publication metamodels). For M2T transformations, vertical lines contain blue boxes that represent individual template modules, and black dots portray binding statements embedded in templates that access the properties of a metamodel element (Figure 6.3: *generateHTML.mtl*). Moreover, in the context of generated textual artifacts, each blue box represents a generated text file, and black dots individual lines of text generated inside the file (Figure 6.3: *index.html*). Blue boxes are multipurpose visual elements for artifacts that have hierarchical structure, such as metamodels (that contain elements and properties), templates (that contain template modules and individual binding statements), and generated textual files (that include non-variable text snippets, and variable generated lines of text). The *branch visualization* combines the multidimensional properties of parallel-coordinate visualizations, and the scalability and filtering power of hierarchical edges [131] to visualize adjacency relations in hierarchical data.

Figure 6.3 also presents the different types of edges used to represent traceability links in a transformation ecosystem. These include *M2M transformation bindings* (Figure 6.3 - A), *M2T transformation bindings* (Figure 6.3 - B) and *generation links* (Figure 6.3 - C). As mentioned before, bindings represent all the potential fine-grained traceability links between artifacts of the transformation ecosystem. They dictate how the properties of a metamodel element are used in order to derive a target metamodel property (in the case of M2M transformations) or a line of text (in the case of a M2T transformation). Furthermore, *generation links* are understood as the runtime dependencies between a M2T transformation and a generated textual file. They represent the dependency relationships between a textual template, and the lines of text generated after its execution (Chapter 5).

The *branch visualization* can be used to grain access to end-to-end traceability information of a transformation chain. It can be used to identify the metamodel and binding expressions that intervene in the generation of a line of code. Furthermore, this visualization enables developers to identify upstream and downstream metamodel dependencies in isolated or composed M2M transformations. In order to help developers to more accurately identify such dependencies, ChainTracker visually distinguishes between *explicit* and *implicit* bindings (Chapter 4) (Figure 6.3, *green* and *red* edges, respectively). As mentioned before, explicit bindings reflect the dependency relationships caused by

assignment expressions in a transformation, while implicit bindings portray dependency relationships given by intermediate statements that manipulate, constrain, or navigate the structure of a metamodel.

Distinguishing between *explicit* and *implicit* bindings enables developers to study the coarse-grained dependencies when assessing the impact of changes in a transformation ecosystem. Furthermore, it also helps them to consider individual statements in complex expressions, and potentially discover fine-grained artifacts that are indirectly bound by navigation or constraint statements. In the *Library to Anonymous Index* example, there are multiple *implicit* bindings between the properties of the Book and Chapter source elements, and the properties of the Publication target element (Figure 6.3 and Table 4.1).

The ChainTracker Binding Filters

ChainTracker includes two main filtering mechanisms to isolate elements and bindings of interest, namely *element filters* and *binding filters*. Using *element filters* (Figure 6.4 - A) developers can select multiple metamodel elements to study all of its dependencies. Furthermore, using *binding filters* (Figure 6.4 - B) developers can select one or multiple elements in the visualization, and isolate all of its related upstream or downstream binding expressions.

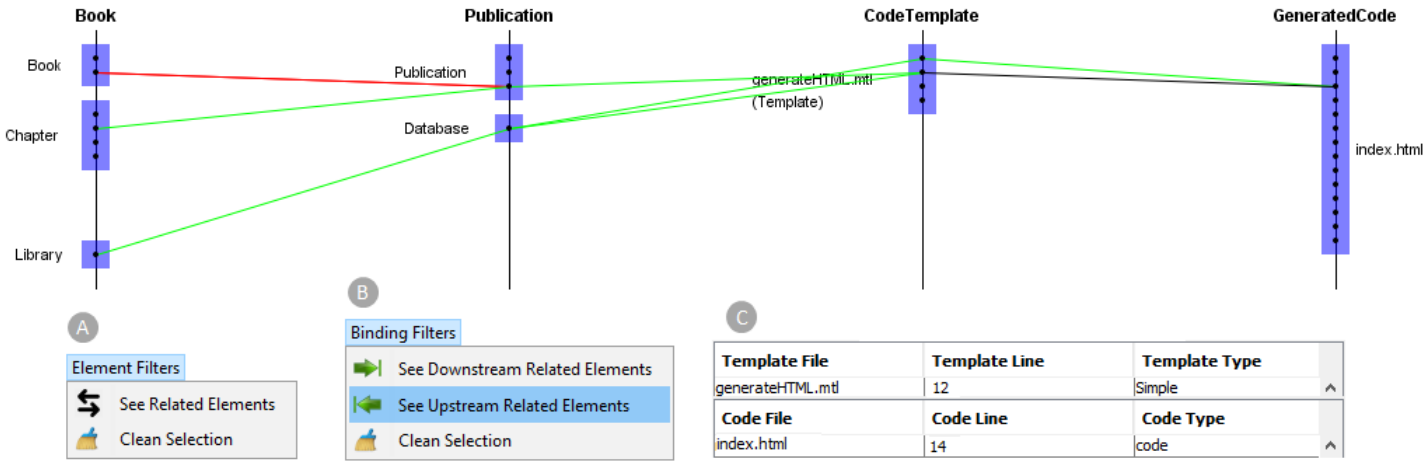


Figure 6.4: ChainTracker - Branch Visualization Filtered

Figure 6.4 presents the *branch visualization* of the *Library to Anonymous Index* transformation chain (Chapter 4). It portrays the result of applying a binding filter to investigate the origins of the *generated line 14* in *index.html*. This line is generated by the *template line 12* in *Publication2HTML.mtl*, by querying the property *nbPages* of the Publication element in the Publication metamodel. Moreover, it depends on the *template line 7* which iterates over the *publications* property of the same element. Furthermore, the property *nbPages* is in turn derived by *Book2Publication* using the properties *nbPages*, *chapters*, and *books* of the Book, Chapter, and Library elements,

respectively (Listing 4.1, lines 20 and 31).

As metamodels and transformations grow in size and complexity, so does the number of artifacts that need be represented in the ecosystem's views. As a result, our parallel-coordinate implementations can potentially become cluttered when dealing with extremely large ecosystems. We have adopted a parallel coordinate arrangement similar to the one presented in [164]. Each categorical dimension is placed equidistant to each other and perpendicular to the x-axis. Furthermore, each category is allocated space proportional to the number of its comprising elements. We have found that for most of the ecosystems available in the literature, our current rendering parameters make visualizations scalable and easy to understand. Nevertheless, ChainTracker allows developers to manually modify the distance between dimensions in order to increase the usability of its visualizations with larger ecosystems.

6.1.2 The ChainTracker Projectional Code Editors

It is important to mention that none of the existing approaches allow developers to project information obtained from their interaction with the transformation visualizations onto transformation editors. This fundamentally limits their usability in the context of supporting developers building and maintaining transformation ecosystems [165].

In [166], Myers et al. explored how conventional human-computer interaction (HCI) methods can help researchers to better understand the needs of developers when dealing with complex software artifacts. Particularly, Myers et al. have found that a key use for code visualizations is to guide developers to the right code to look at, instead of being an aid to understanding on their own. Following this principle, ChainTracker enables developers to project information they have discovered in the visualizations onto textual editors, and vice versa.

ChainTracker offers a code projection menu where developers can find three types of projections, namely *downstream projections*, *upstream projections*, and *single binding projections* (Figure 6.5). In all cases, developers can select artifacts in the visualizations, in order to find their original textual representation. To understand their usage, let us briefly examine four examples.

1. Figure 6.5-A showcases the use of *downstream projections* in order to isolate the textual representation of the bindings associated to the Book element of the Book metamodel. In this case all the related bindings are located in the Book2Publication (M2M) transformation (Listing 4.1). However, in Chapter 3 we discuss how downstream dependencies in multi-

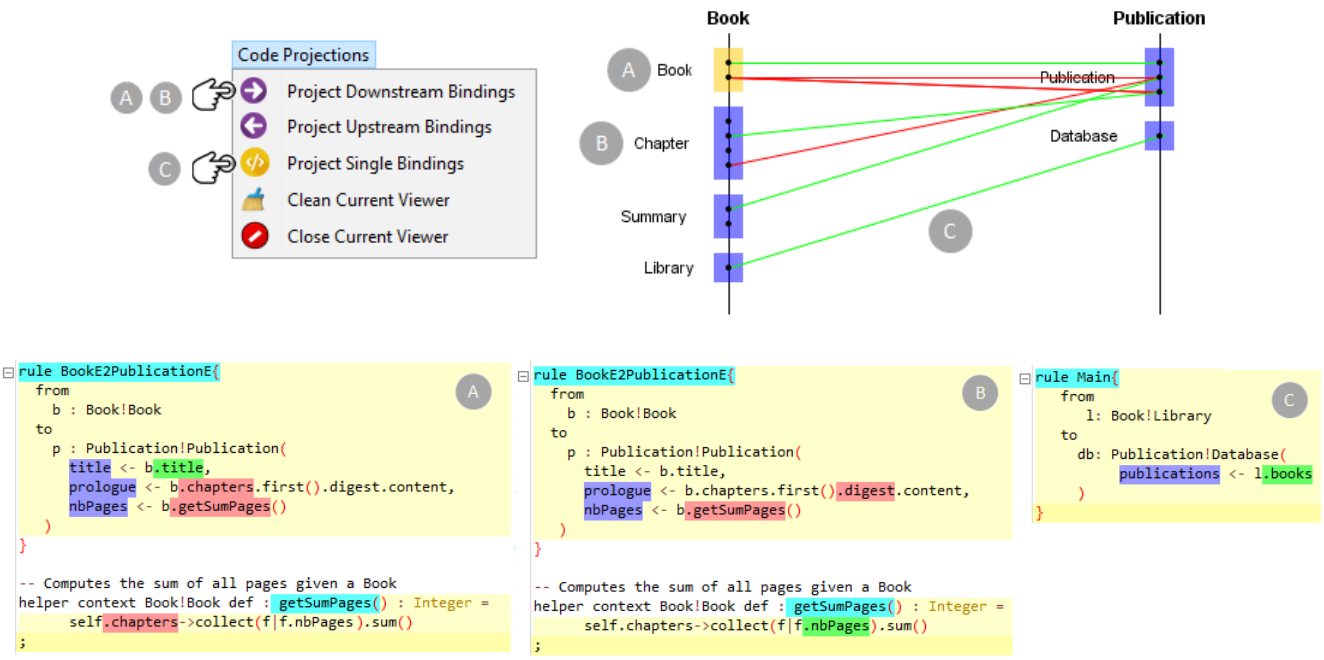


Figure 6.5: ChainTracker - M2M Binding Projections

branched transformation ecosystems can be found in multiple transformations. In effect, M2M binding expressions are highlighted by first locating the rule where each expression is located (*yellow* shadow), the rule's name (*cyan*), the property targeted by the binding (*blue*), and its comprising binding tuples (*green* and *red* for explicit and implicit bindings, respectively).

2. Figure 6.5-B presents the result of applying *downstream projections* to obtain the textual representation of bindings associated to the Chapter element of the Book metamodel. A similar result will be obtained when applying *upstream projections* to elements of the Publication metamodel. ChainTracker identifies helper calls as implicit bindings, and helps developers to follow their execution using code projections.

3. Figure 6.5-C presents the result of applying a *single binding projection* in order to isolate the expressions that realize the binding between the *"books"* property of the Library element, and the *"publications"* property of the Database element.

4. Figure 6.6-A depicts the result of selecting a M2T binding, and projecting it onto the textual editor (B). In this particular case the editor showcases the contents of the Publication2HTML (M2T) transformation (Listing 4.2). It highlights the binding expression that uses the property *"prologue"* of the Publication element in the Publication metamodel. Information relevant to the current selection is conveyed in the contextual tables of the environment (C). Moreover, generation links can also be subject of the projections onto generated textual files.

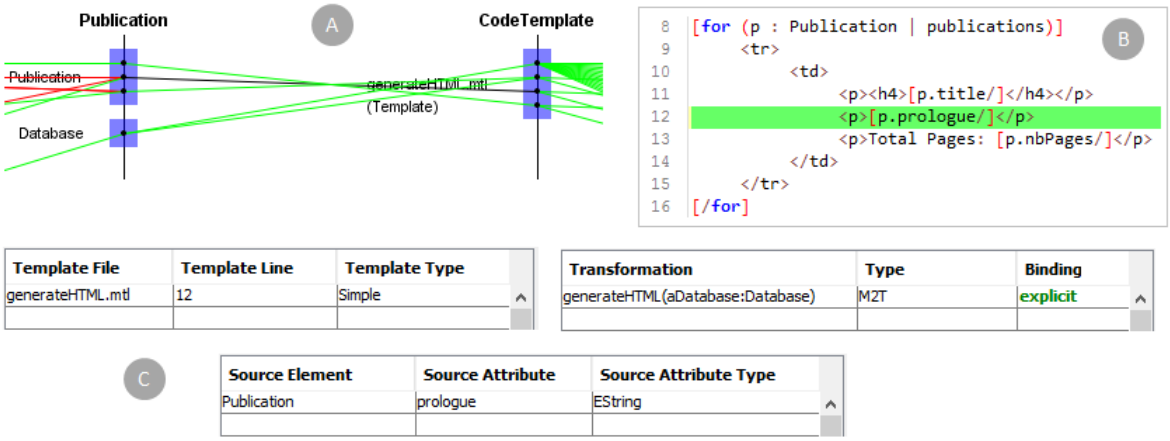


Figure 6.6: ChainTracker - M2T Binding Projections

Code projections help developers to navigate the visualizations and map their semantics onto the expressions that they represent. They also provide a straight forward strategy to find unused code in transformation scripts, so they can be refactored or deleted. In this context, a portion of a transformation script can be considered orphan if it remains clear after projecting all of its associated bindings from the branch visualization. ChainTracker also includes a reverse code projection feature. It enables developers to select portions of text in templates and generated files, in order to investigate its precise graphical representation (Figure 6.1-D)[2].

6.1.3 The ChainTracker Contextual Tables

One of the strategies that ChainTracker uses to minimize the amount of information displayed in its visualizations is to omit details about the names and properties of the metamodel elements and binding expressions. However, this information might be of high value to developers debugging complex transformation expressions. The contextual tables present information that reveal details not available in the visualizations of the environment (Figure 6.1-C).

Developers can select a metamodel element in the visualizations, and the three contextual tables will display information about the name and type of its properties, related upstream bindings (bindings that have as target the selected element), and related downstream bindings (bindings that use the selected element as the source for the creation of another element). Developers can also select individual bindings to obtain information about the transformation where the binding is located. When a binding is selected, the three contextual tables will portray information regarding the name of its corresponding transformation, and binded elements and properties. If the selected binding is a M2T binding, its template location will be presented along with its expression type, i.e., binding due to a simple expression, conditional expression, or loop expression (Chapter 5).

[2]A more elaborated example of this feature can be found in `https://guana.github.io/chaintracker/tutorial.html`.

7

ChainTracker Usability Evaluation

In this chapter, we present an empirical study that evaluates the usability of ChainTracker using Eclipse Modeling (with ATL and Acceleo plugins) as the industry baseline. This chapter is structured as follows: Section 7.1 presents the design of our study, including our hypotheses, dependent and independent variables, participants, and the traceability-driven tasks proposed in the study. Section 7.2 discusses the protocol of the study. Section 7.3 presents our results. Section 7.4 presents our analysis. Finally, Section 7.5 presents the threats to validity and our mitigation strategies.

7.1 Empirical Study Design

We investigate the performance of developers when reflecting on the execution semantics of M2M and M2T transformations. We measure the accuracy and efficiency of developers when asked to identify dependency relationships between transformation artifacts using ChainTracker, and compare their performance with that of developers using Eclipse Modeling. We use PhyDSL and ScreenFlow (Chapter 3) as the evaluation object systems. Concretely, we investigate two research questions:

- **RQ1**: Do developers using ChainTracker identify metamodel and generation dependencies in transformation ecosystems more accurately and efficiently than those using Eclipse Modeling?
- **RQ2**: Do the size and complexity of transformation ecosystems affect the effectiveness of ChainTracker in helping developers identify their metamodel and generation dependencies?

7.1.1 Hypotheses

In this study, we hypothesize that enabling developers to interactively explore the execution semantics of a transformation ecosystem can significantly improve their performance when reflecting on an ecosystem's metamodel and generation dependencies. In order to investigate our research questions, we outlined two individual null hypotheses:

- H_{01}: Developers spend an equal amount of time identifying metamodel and generation dependencies in model transformations using ChainTracker and Eclipse Modeling editors.
- H_{02}: Developers provide equally correct answers identifying metamodel and generation dependencies in model transformations using ChainTracker and Eclipse Modeling editors.

7.1.2 Dependent Variables

Considering the hypotheses H_{01} and H_{02}, our experiment has two dependent variables on which our treatments are compared:

- Var_A: Time developers spend solving each task.
- Var_B: Developers' accuracy solving each task.

7.1.3 Independent Variables

The four independent variables of this study are summarized in Table 7.1. Variables $VarCT_1$ and $VarEM_1$ represent the tasks designed to evaluate the performance of developers working in the context of ScreenFlow, and $VarCT_2$ and $VarEM_2$ in the context of PhyDSL, using ChainTracker (CT) and Eclipse Modeling (EM), respectively.

Object System	**Tasks CT**	**Tasks EM**
Object 1: ScreenFlow	$VarCT_1$	$VarEM_1$
Object 2: PhyDSL	$VarCT_2$	$VarEM_2$

Table 7.1: ChainTracker Evaluation - Independent Variables.

7.1.4 Tasks

In this study, we measured the performance of developers when asked to identify dependency relationships between artifacts in two transformation ecosystems of different complexity. In order to do so, we created a set of task templates that aim at understanding how developers identify dependency relationships at different levels of granularity, using different tool treatments. These tasks are grouped in five main families, namely, (a) *determining metamodel coverage and expression location*,

(b) *identifying metamodel dependencies in M2M transformations*, (c) *identifying metamodel dependencies in M2T transformations*, (d) *identifying generation dependencies in M2T transformations*, and e) i*dentifying generation dependencies in model transformation chains* (Section 7.1.4 - 7.1.4). The proposed question templates can be used to replicate this study with different object systems. Appendix A.1 presents the questionnaires that instantiate the proposed templates in the context of PhyDSL and ScreenFlow[1]. Let us briefly present each family of tasks.

Determining Metamodel Coverage and Expression Location

The goal of this family of tasks is to investigate how developers identify the major components of a transformation ecosystem, and measure their performance when inferring its high-level compositional structure. Tasks in this family use the following templates:

- Are there any unused elements in the *[metamodel-name]* metamodel? if so which ones?
- What transformation rule contains the binding expression *[expression]*?
- What transformation script contains the *[rule-name]* rule?
- What files does the *[template-name]* template generate?

Identifying Metamodel Dependencies in M2M Transformations

This set of tasks requires developers to identify the dependencies that exist between the source and target metamodels of a single M2M transformation. This family of tasks is divided in two categories, namely *element-* and *property-level* dependency tasks.

More often than not, *property-level* dependencies are localized in non-trivial binding expressions that realize the intent of a transformation script; these include metamodel navigation statements or procedural calls to helpers. Tasks in this family are also distinguished by the direction of the dependencies that need to be identified. While some tasks require developers to identify upstream dependencies, other tasks investigate their performance identifying downstream dependencies. Tasks in this family are specified using the following templates:

- What metamodel elements are used in the creation of the *[metamodel-name ! element-name]* element? (i.e., element upstream dependencies)
- What metamodel elements are created using the *[metamodel-name ! element-name]* element? (i.e., element downstream dependencies)

[1]The PhyDSL and ScreenFlow source code, and corresponding ChainTracker visualizations can be found at: `https://github.com/guana/chaintracker-eval`.

- What metamodel elements are created using the property *[property-name]* of the *[metamodel-name ! element-name]* element? (i.e., property downstream dependencies)

Identifying Metamodel Dependencies in M2T Transformations

This family of tasks investigates how developers identify upstream dependencies in M2T transformations. Concretely, they ask developers to determine the metamodel elements required for the execution of one or multiple bindings expressions in a M2T transformation. Furthermore, they ask developers to evaluate whether such elements have upstream dependencies in M2M transformations that are potentially linked in previous steps of a transformation chain. Tasks in this family follow the template:

- Considering the entire transformation chain, what metamodel elements does the template line *[line-number]* in *[template-name]* depend on?

Identifying Generation Dependencies in M2T Transformations

This set of questions requires developers to identify dependencies between generated textual artifacts, e.g., code, and their originating M2T transformations. They are presented to the participants using the following template:

- What template lines in *[template-name]* are used in the generation of line *[line-number]* in *[generated-file-name]*?

Identifying Generation Dependencies in MTCs

This collection of tasks investigate how developers identify the dependencies of a generated textual artifact in a holistic fashion. They ask developers to determine all the metamodel elements and properties that intervene in the generation of one or multiple lines in a generated textual artifact. They are also divided in two categories; namely, *element-* and *property-level* dependency tasks. The tasks in this family use the following templates:

- Considering the entire transformation chain, what metamodel elements does the generation of line *[line-number]* in *[generated-file-name]* depend on?
- Considering the entire transformation chain, what metamodel properties does the generation of line *[line-number]* in *[generated-file-name]* depend on?

7.1.5 Detailed Hypotheses

Taking into account our two high-level null hypotheses and our two object systems, this study tries to reject four detailed hypothesis:

$$H_0Var_{A1} : \tilde{V}ar_ACT_1 = \tilde{V}ar_AEE_1$$
$$H_0Var_{A2} : \tilde{V}ar_ACT_2 = \tilde{V}ar_AEE_2$$
$$H_0Var_{B1} : \tilde{V}ar_BCT_1 = \tilde{V}ar_BEE_1$$
$$H_0Var_{B2} : \tilde{V}ar_BCT_2 = \tilde{V}ar_BEE_2$$

Hypotheses H_0Var_{A1} and H_0Var_{A2} compare the median time spent by developers solving the proposed tasks in single and multi-branched transformation chains, respectively *(i.e., developers spend an equal amount of time solving questions using ChainTracker and Eclipse editors for single and multi-branched transformation chains)*. Moreover, hypotheses H_0Var_{B1} and H_0Var_{B2} compare the median accuracy (in terms of task solution correctness) of developers solving the proposed tasks on single and multi-branched transformation chains, respectively *(i.e., developers provide equally correct answers using ChainTracker than they do using Eclipse editors for single and multi-branched transformation chains)*.

7.1.6 Participants

This study involved 25 software engineers with an average of 7.5 years of software development experience in industry. All participants were enrolled in a professional masters program which includes an intensive course on software-engineering automation using model transformation languages. The participants had an average of 6 months of training in rule- and template-based model-transformation languages. Their training also included Eclipse Modeling as their main development environment for model transformations. All of the participants reported having used Eclipse in their professional development practice, and Eclipse Modeling in the context of their graduate course in model transformation technologies. None of the participants had experience with model transformation technologies in an industrial setting. Furthermore, none of the participants had previous knowledge of the case studies used in this study. Our pool of participants is representative of a community in which developers have only introductory training on model-transformation technologies, yet considerable experience in the general software engineering field.

7.1.7 Data Analysis

Given the sample size and the non-normal distribution of the data collected in this study, we adopted a Mann-Whitney "U" non-parametric test to investigate our hypothesis propositions. The Mann-Whitney compares the median of the observations for datasets with pronounced outliers. This makes the test a suitable analysis tool for small unpaired datasets with skewed distributions. All of our hypotheses were evaluated using a two-tailed version of test. In this study, we consider an alpha level with a p-value lower than 5%, thus we consider an acceptable probability of 0.05 for Type-I error, i.e., rejecting the null hypothesis when it is true.

7.2 Empirical Study Protocol

The protocol of the study was divided in three sessions conducted in the course of one week (Figure 7.1). Session 1 (training session) involved an introductory tutorial on the features of ChainTracker. Sessions 2 and 3 involved two independent working sessions in which 25 participants solved the tasks assigned for the object systems of the study.

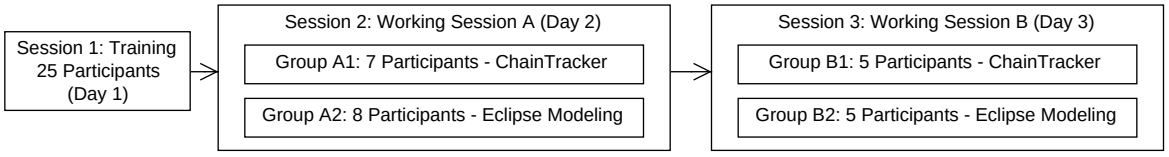


Figure 7.1: The Empirical Study Protocol

7.2.1 Training

The training session was structured in two 60 minute parts divided by a 15 minute break. The first half of the session involved a tutorial on the use of ChainTracker. The second half consisted of a laboratory workshop that provided hands on experience with the analysis environment.

The 25 participants received a presentation that introduced a case study of similar complexity to *Library to Anonymous Index* in Chapter 4. The case study was used to explore the features of ChainTracker and the graphical notation of its visualizations. Additionally, participants completed a worksheet with questions conforming to the templates described in Section 7.1.4. Two research assistants helped participants to solve questions as they completed the guide[2]. At the end of the first session, the 25 participants were randomly divided in two groups of 15 and 10 participants, namely, Group A and B, respectively. Group A was assigned to the working session A, and Group B to the working session B.

7.2.2 Working Sessions

Participants assigned to Group A were randomly divided in two sub-groups of 7 and 8 participants, namely Group A1 and A2, respectively. Similarly, participants in Group B were randomly divided in two sub-groups of 5 participants, namely Group B1 and B2. Each sub-group was assigned to individual computer laboratories with virtual machines containing ChainTracker and Eclipse Modeling with ATL and Acceleo plug-ins. Participants in Working Sessions A and B were assigned ScreenFlow and PhyDSL as their case studies, respectively.

[2]The training material and case study can be found at `https://guana.github.io/chaintracker/tutorial.html`.

At the beginning of each session, participants received a 15 minute presentation on the major components of their corresponding case studies. This presentation included 10 minutes of questions and final setup. All participants received a printed copy of the metamodels that comprise their assigned case study. Finally, the participants in Groups A1 and B1 were instructed to solve 25 tasks using ChainTracker, and participants in Groups A2 and B2 using Eclipse Modeling (Appendix A.1). Both working sessions had a maximum time restriction of 2.5 hours. Each computer laboratory was supervised by a research assistant who was available to answer high-level questions about both Eclipse and ChainTracker, as well as the setup of the virtual machines. At the end of both working sessions participants were rewarded with a gift card equivalent to $25CAD for their time.

7.2.3 Data Collection

The working sessions were instrumented with a survey application developed by our research team[3]. Each participant logged into the survey application using a unique ID assigned before the beginning of the session. The application presents one task at the time until all tasks in the questionnaire are answered. The application does not allow participants to skip tasks. Furthermore, it does not allow participants to return to a task once it has been completed. Our survey application measures the total time spent by the participant in each task. The total time is calculated as the time between a task is presented to the participant, and the time a valid answer is submitted.

The survey application is capable of presenting three types of questions, namely, multiple-choice questions, list-based questions, and multiple-selection questions. Each type captures the participant's answers using different mechanisms. Multiple choice questions receive one answer from a predefined set of options. List-based questions receive one or multiple open-ended answers, and multiple selection questions receive one or multiple answers from a predefined set of options. Our survey application stores the results of a session in a remote server using a REST API.

It is important to mention that tasks in each questionnaire were organized to make the experience of the participants engaging and balanced throughout the entire session. Similar tasks (i.e., tasks belonging in the same family) were distributed throughout the questionnaires, in almost regular intervals, and their level of difficulty was considered to avoid having collections of consecutive task with disproportionate complexity level. We categorized the difficulty of the each task in three increasing levels –easy, medium, and hard– based on our observations during the preliminary pilot studies. The questionnaires took this classification into account and they do not present more than two hard questions or more than three medium questions, one after the other. Since participants only use one tool treatment throughout the course of the study, counterbalancing was not sought.

[3] `https://www.github.com/guana/surveygen`

In order to quantify the developers' accuracy for each task, we designed a point-based scoring mechanism similar to the one used in [167] and [168]. Each task in the questionnaire is scored individually. For multiple choice questions each participant is given a single point if the selected answer is correct. In the case of multiple selection questions, one point is given for each correct option selected. Furthermore, half a point is taken for every incorrect option selected, as well as for every correct option missed. Similarly, for list-based questions one point is given for correct answers, and half a point is taken for incorrect or missed options.

7.3 Results

In this section, we present the results of our study. We use summary tables to present the performance of developers in each family of tasks (Appendix A.2). Result summary tables are divided in two main areas. The first area presents the p-values corresponding to the statistical evaluation of our hypotheses. We present individual p-values for each of the dependent variables under consideration, namely the time and accuracy of developers completing a task. The second area compares the mean, standard deviation (SD), and median of the dependent variables across treatments. Furthermore, summary tables include the maximum attainable score for each task, and two box and whisker diagrams that portray the distribution of the recorded measurements for both variables under consideration. In all diagrams, green boxes portray the distribution of measurements for participants using ChainTracker (CT), and purple boxes for participants using Eclipse (EC).

7.3.1 Determining Metamodel Coverage and Expression Location

The goal of this family of tasks is to investigate the performance of developers identifying the major components of a transformation ecosystem, and the high-level compositional structure of its underlying transformations. Concretely, these tasks require developers (a) to identify the files generated by a given M2T transformation, (b) to locate individual binding expressions in the transformations of an ecosystem, and (c) to evaluate the coverage of an ecosystem's metamodels. Tables A.3 and A.4 (Appendix A.1) summarize the results for this family of tasks.

1. When identifying the files generated by a given M2T transformation in ScreenFlow (Q9 and Q10, Table A.3) developers were on average 63% more efficient with Eclipse than with ChainTracker. This effect is less pronounced, i.e., 22%, in the case of developers working on PhyDSL (Q9 and Q10, Table A.4). However, there is no substantial difference regarding the accuracy of both groups of developers.

2. When asked to isolate individual binding expressions in ScreenFlow (Q1 and Q2, Table A.3), developers supported by Eclipse were on average 29% more efficient than those using Chain-

Tracker. In the context of PhyDSL, no performance difference was observed. Furthermore, there is no difference regarding the accuracy of developers in the ecosystems under study.

3. Developers assessing the coverage of metamodels using ChainTracker were on average 46% more efficient and 72% more accurate that those using Eclipse in ScreenFlow (Q17 and Q18, Table A.3). On cursory examination, developers using Eclipse seem to be more efficient assessing metamodel coverage in PhyDSL (Q17 and Q18, Table A.4). However, considering their lower accuracy, we believe this group of developers only submitted partial answers. Participants using ChainTracker were on average 200% more accurate assessing the coverage of metamodels in PhyDSL.

There is no statistically significant evidence to reject any of our hypothesis propositions for developers completing this family of tasks.

7.3.2 Identifying Metamodel Dependencies in M2M Transformations

This set of tasks investigates the performance of developers identifying the upstream and downstream dependencies between a collection of metamodels, in the context of individual M2M transformations. The results for this family of tasks are divided in two levels of granularity, namely identifying *element-level* dependencies (Tables A.5 and A.6), and identifying *property-level* dependencies (Tables A.7 and A.8, Appendix A.1).

1. ChainTracker improved the accuracy of developers assessing *element-level dependencies* in the context of individual M2M transformations (Q3, Q4, Q11, and Q12, Tables A.5 and A.8). Developers using ChainTracker were on average 83% more accurate, and 48% more efficient than those using Eclipse, in both ecosystems under analysis.

2. Participants using ChainTracker were significantly more accurate than those using Eclipse when determining *property-level* dependencies (Q23 and Q13, Tables A.7 and A.8), almost 900% more accurate, in fact. In terms of efficiency, participants using ChainTracker were on average 36% faster than those supported by Eclipse in PhyDSL.

Considering our accuracy-related hypotheses, we can reject H_0Var_{B1} (Q11: p=0.0310, and Q12: p=0.0006), and H_0Var_{B2} (Q11: p=0.0310, Q4: 0.0072, Q3: p=0.0065, and Q12: p=0.0065) for developers identifying *element-level dependencies*. We can also reject our efficiency-related hypothesis H_0Var_{A1} (Q11: p=0.0021, Q4: p= 0.0139, and Q12: p=0.0012) in the case of developers working on ScreenFlow. However, there is not statistically significant evidence to reject H_0Var_{A2}. Finally, we have statistically significant evidence to reject our accuracy-related hypotheses, namely

H_0Var_{B1} (Q23: p=0.0046 and Q13: p=0.0491) and H_0Var_{B2} (Q23 p=0.0412 and Q13 p=0.0393) for developers identifying *property-level dependencies*.

7.3.3 Identifying Metamodel Dependencies in M2T Transformations

This family of tasks requires developers to identify the metamodel elements consumed by one, or multiple, binding expressions in a M2T transformation, and to evaluate whether these elements have upstream dependencies in a model-transformation chain. Tables A.9 and A.10 (Appendix A.1) summarize the relevant results.

1. Developers using ChainTracker were on average 55% more accurate than those using Eclipse (Q5, Q15, and Q22, Table A.9) in ScreenFlow. In the case of PhyDSL, the ChainTracker advantage is even more pronounced. Developers using ChainTracker were 90% more accurate than those using Eclipse (Table A.10).

We found statistically significant evidence to reject our accuracy-related hypotheses H_0Var_{B1} (Q5: p= 0.0119 and Q22: p=0.0025), and H_0Var_{B2} (Q5: p=0.0072, Q15: p=0.0097 and Q22: p=0.0117) for developers identifying metamodel dependencies in M2T transformations. There is no significant difference in the efficiency of developers completing this family of tasks.

7.3.4 Identifying Generation Dependencies in M2T Transformations

This set of tasks requires developers to identify dependencies between individual lines of text, e.g., code, and their originating M2T transformations. Tables A.11 and A.12 (Appendix A.1) summarize the relevant results.

1. Developers using ChainTracker were on average 19% more efficient and 94% more accurate than those using Eclipse (Q6, Q16, and Q25).

Although this difference is statistically significant for Q25 in ScreenFlow, and Q6 in PhyDSL, there is no significant evidence to reject any of our hypothesis propositions for developers completing this family of tasks.

7.3.5 Identifying Generation Dependencies in MTCs

This collection of tasks investigates how developers identify the upstream dependencies of a generated textual artifact, i.e., the metamodel elements needed for the generation of one or multiple lines of code. It is important to mention that this family of tasks inquires about the metamodel dependencies throughout an entire transformation chain. The performance measurements for this family of tasks are also divided in two levels of granularity, namely identifying *element-level*

dependencies (Tables A.13 and A.14), and identifying *property-level* dependencies (Tables A.15 and A.16, Appendix A.1).

1. When identifying *element-level* dependencies in ScreenFlow, developers using ChainTracker were on average 62% more accurate than developers using Eclipse. This effect is significantly more pronounced in PhyDSL where developers were 100% more accurate (Q7, Q8, Q19, Q20, and Q21, Tables A.13 and A.14). This observation is statistically significant for all the tasks under consideration.

2. Developers using ChainTracker were 66% more efficient than those using Eclipse in ScreenFlow. This finding is statistically significant for all tasks. In PhyDSL, we observed that even though the median time spent by developers in these tasks is lower for developers using ChainTracker, the difference is significant only in Q8.

3. When tracing *property-level* dependencies, ChainTracker developers were on average 68% more accurate than those using Eclipse in both ecosystems under study (Q14, and Q24, Tables A.15 and A.16). Similar to *element-level* dependencies, we observed that identifying end-to-end *property-level* dependencies, in both linear and multi-branched transformation chains is a challenging task. In this context, participants using Eclipse obtained a median accuracy score of 0.5 and 1.0 for tasks with maximum attainable scores of 8.0 (Q14 in ScreenFlow) and 4.0 (Q14 in PhyDSL), respectively.

We found statistically significant evidence to reject our accuracy-related hypotheses, namely H_0Var_{B1} (Q7: p=0.0032, Q8: p=0.0073, Q19: P=0.0443, Q20: p=0.0007, and Q21: p=0.0007), and H_0Var_{B2} (Q7: p=0.0094, Q8: p=0.0072, Q19: p=0.0066, Q20: p=0.0055, and Q21: p=0.0055) for developers identifying end-to-end *element-level dependencies*. Furthermore, we can reject our efficiency-related hypothesis H_0Var_{A1} (Q7: p=0.0093, Q8 p= 0.0012, Q19: p=0.0037, Q20: p=0.0003, and Q21: p= 0.0003).

We can also reject our accuracy-related hypotheses, namely H_0Var_{B1} (Q14: p= 0.0090, and Q24: p= 0.04871), and H_0Var_{B1} (Q14: p= 0.0066 and Q24: p=0.0248) for developers identifying end-to-end *property-level* dependencies. With respect to our efficiency-related hypothesis, we found statistically significant evidence that ChainTracker outperforms Eclipse Modeling in helping developers analyzing end-to-end *property-level* dependencies. However, we believe that most of the time, observations can not be fairly analyzed considering that most developers using Eclipse were highly inaccurate. We require further analysis and additional empirical information to validate our hypotheses for developers completing this type of task.

7.4 Discussion

7.4.1 Determining Metamodel Coverage and Expression Location

Developers using ChainTracker performed less efficiently than developers using Eclipse in locating individual binding expressions across the ecosystems under study. At the same time, ChainTracker developers were considerably more accurate and more efficient when determining the coverage of their metamodels. None of these results, however, are statistically significant.

Tasks Q1 and Q2 require developers to isolate individual binding expressions in the the ecosystems under study. Furthermore, Tasks Q9 and Q10 require developers to identify the files generated by a given M2T transformation (Tables A.1 and A.2). We noticed that developers using Eclipse relied on the pattern-matching features of the environment to complete these tasks. As shown in Tables A.3 and A.4, this strategy proved very effective. Eclipse developers were between 24% and 51% more efficient than those supported by ChainTracker in Q1 and Q2, and between 22% and 63% more efficient in Q9 and Q10. As ChainTracker does not offer pattern-matching capabilities, developers had to manually explore the transformations of the ecosystems to complete the aforementioned tasks.

Tasks Q17 and Q18 require developers to determine the coverage of two metamodels in their respective ecosystems, namely Mockup and GUI for ScreenFlow, and PhyDSL and Dynamics for PhyDSL (Tables A.1 and A.2). Eclipse developers had to manually explore all the upstream and downstream bindings that operate on each metamodel to evaluate its coverage. In ScreenFlow, they had to examine 56 lines of code, corresponding to its single M2M transformation (Figure 3.14). In PhyDSL, they had to study a total of 243 lines of code, corresponding to its four M2M transformations (Figure 3.1). Moreover, the Mockup and GUI metamodels of ScreenFlow have a total of 14 metamodel elements, and the PhyDSL and Dynamics metamodels of PhyDSL have 47 elements. Indeed, manually examining large transformation codebases, looking for the usage of dozens of metamodel elements, is a daunting, if not impossible, task.

Participants using ChainTracker were not required to manually study transformation codebases to identify uncovered metamodel elements. Coverage information is presented in the *overview visualization* using pie-charts that summarize the in- and out-coverage metrics for all of the metamodels of an ecosystem. Furthermore, using ChainTracker's *branch visualization*, developers were able to identify metamodel elements that have no bindings attached to them, which makes uncovered elements easy to identify (Figure 6.3). This information can be effectively used to precisely remove unused metamodel elements and properties, and to identify portions of code that are never executed.

7.4.2 Identifying Metamodel Dependencies in M2M Transformations

Developers using ChainTracker are significantly more accurate and more efficient than those using Eclipse at identifying *element-level* dependencies in M2M transformations. In effect, identifying downstream dependencies is significantly more difficult to developers, than pinpointing their upstream counterparts.

We believe that most of the positive impact that ChainTracker had on the accuracy of developers stems from the usability of its *branch visualization*, which presents a unified view of the traceability links contained in the transformations of an ecosystem.

Identifying downstream dependencies (Q4 and Q12) in rule-based transformations is fundamentally more difficult than identifying upstream dependencies (Q2 and Q11, Tables A.1 and A.2). To complete both sets of tasks, developers need to examine the transformations and interpret their execution semantics. In order to identify upstream dependencies, developers only need to examine the binding expressions located in the matched rule corresponding to the element of interest. These expressions determine the metamodel elements that are used for its creation. In contrast, developers looking for an element's downstream dependencies, require to pinpoint all the binding expressions located in potentially multiple transformation rules, used to create an undetermined number of target elements. In both cases, developers need to consider all the implicit and explicit binding expressions that use the element of interest to either constrain their execution, or gain access to other metamodel elements.

ChainTracker interactive-filtering capabilities proved effective in isolating information corresponding to individual metamodel elements, and their related binding expressions. This enabled developers to analyze the downstream and upstream dependencies despite of the size of the visualizations. Furthermore, ChainTracker enables developers to quickly identify implicit and explicit bindings in complex OCL expressions. Our empirical observations revealed that the automatic reasoning of model transformations, and the quick access to fine-grained traceability links, enabled developers to identify dependency relationships in a more efficient and accurate manner. As shown in Tables A.5 and A.6, developers using Eclipse had a very low accuracy in tasks that included the analysis of expressions with multiple implicit bindings (Q4, and Q12).

Identifying *property-level* dependencies in M2M transformation is particularly difficult due to the large search spaces that need to be explored to complete this task. ChainTracker developers were significantly more accurate in identifying *property-level* dependencies than participants using Eclipse. This phenomenon is more pronounced as the complexity of the ecosystem increases.

Similarly to tasks that require developers to identify downstream element dependencies, identifying *property-level* dependencies requires developers to manually interpret the binding expressions of all the transformations in an ecosystem. We observed that Eclipse users used its pattern-matching capabilities to find all the expressions that used a property of interest. This strategy is somehow effective in the case of properties with very distinctive names, such as in the case of *"fromScreen"* in Q23 (Table A.1). However, for properties with common names, such as *"value"* or *"name"* (Q23 and Q13 in Table A.2) the text-based search approach proved ineffective.

7.4.3 Identifying Metamodel Dependencies in M2T Transformations

Developers using ChainTracker were significantly more accurate than those using Eclipse in identifying upstream metamodel dependencies in M2T transformations. This effect is more pronounced in PhyDSL, which suggests that the complexity of these tasks, and the usefulness of ChainTracker, increases in multi-branched transformation chains.

Developers using Eclipse appeared to be more efficient than those using ChainTracker when addressing Q5, Q15, and Q22. However, they were much less accurate. Due to the small set of participants in each working session, we are unable to isolate developers that use Eclipse and that obtained high accuracy scores, in order to make a fair statistical analysis of our time-dependent hypotheses. It is worth noticing that developers using ChainTracker obtained a perfect accuracy score for all task in this family. Conversely, Eclipse developers had a sparse accuracy distribution (Table A.9). This suggests that Eclipse developers struggle to identify upstream dependencies in multi-step transformation chains, even when these are due to simple binding expressions, e.g., Q15 in ScreenFlow.

As a concrete example of the challenges that developers face completing this family of tasks, let us briefly explore Q15 in the context of PhyDSL (Table A.2). This task requires developers to find the element dependencies of the expressions located in *line 110* of the *generateScoring.mtl*

transformation (Figure 7.2 - A). Furthermore, developers need to determine whether there are additional metamodel dependencies due to potential upstream M2M transformations in PhyDSL (Figure 7.3 - B).

A

```
109 scoreTotal += [collisionRule.collisionAction.points/];
110 playerWins = ![collisionRule.collisionAction.userLoses/];
111 gameEnds = [collisionRule.collisionAction.gameEnds/];
112 [if (collisionRule.collisionAction.removeActorId.equalsIgnor
113 if(a.m_userData.equals(PhysicsView.[collisionRule.actorAId/]
114     mainActivity.removebody(a);
115 } else if (b.m_userData.equals(PhysicsView.[collisionRule.ac
116     mainActivity.removebody(b);
```

Figure 7.2: PhyDSL - generateScoring.mtl (M2T)

B

```
65 rule Effect2Action {
66  from
67       effect : Phydsl!ScoreEffect
68  to
69     action : Scoring!Action (
70
71       gameEnds <- effect.end.boolean.solveBool(),
72       points <- effect.points.value.toString(),
73       userLoses <- if not effect.loses.oclIsUndefined()
74                    then effect.loses.boolean.solveBool()
75                    else false
76                    endif,
77       removeActorId <- if not effect.dissapears.oclIsUndefined()
78                    then effect.dissapears.name
79                    else ''
80                    endif
81     )
82 }
...
92 helper context Phydsl!BooleanType def: solveBool():Boolean =
93      if self.value = 1
94      then true
95      else false
96      endif;
```

Figure 7.3: PhyDSL - `Effect2Action` and `solveBool()` (M2M)

Figure 7.4 presents the ChainTracker visualization, relevant to task Q15 for PhyDSL. Line 110 in *generateScoring.mtl* uses two attributes corresponding to two elements in the Scoring metamodel, namely Action and CollisionRule (Figure 7.4 - A). This line generates a portion of code that implements the scoring mechanisms of the video games generated by PhyDSL. The Action element is created by the `Effect2Action` rule (Figure 7.3). This rule uses the `solveBool()` helper to complete its transformation intent. As shown in Figure 7.4 - B, the Action element has multiple implicit dependencies given by binding expressions that include helper calls, and metamodel navigation statements. Due to the complexity of the binding expressions, manually locating the usage of the metamodel of interest, and interpreting their corresponding dependency relationships is a challenging task.

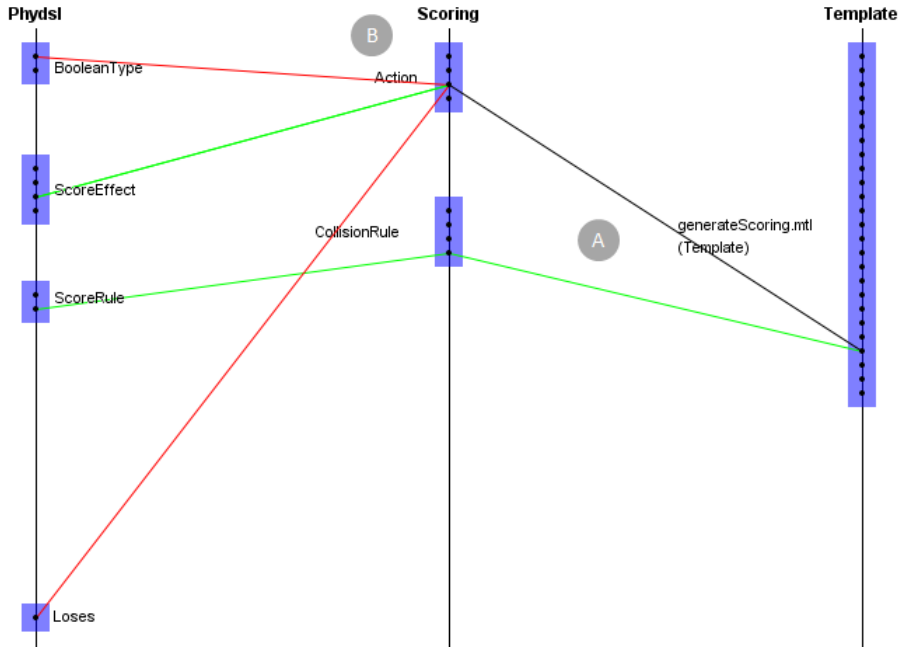


Figure 7.4: PhyDSL - Task Q15 ChainTracker Visualization

Our observations during the study suggest that ChainTracker's *branch visualization*, and its code-projection capabilities, enabled developers to gain end-to-end traceability information, and to quickly filter upstream metamodel dependencies in both ecosystems under analysis.

7.4.4 Identifying Generation Dependencies in M2T Transformations

We observed that participants using Eclipse use the Acceleo Profiler [112] in order to find generation dependencies in the M2T transformations under study. The Acceleo Profiler enables developers to debug M2T transformations, and to identify the binding expressions responsible for the generation of a particular line of code. On the other hand, ChainTracker developers were able to select portions of generated code, and obtain their corresponding generation dependencies using its reverse code projection capabilities. The projections not only highlighted the M2T bindings that originated the selected portions of code, but provided quick access to the metamodel elements and properties, used for their generation.

ChainTracker developers were on average 19% more efficient in identifying generation dependencies in M2T transformations. We believe this is because reverse code projections do not require to interactively execute transformations to study their bindings. Instead, they offer a self-contained view that can be studied as a whole, regardless of the complexity of the transformations under analysis, or the number of steps required for their execution. Our observations during the study suggest that interactive traceability visualizations are as useful as off-the-shelf transformation debuggers to identify generation dependencies in M2T transformations.

7.4.5 Identifying Generation Dependencies in MTCs

ChainTracker developers were significantly more accurate and more efficient at identifying end-to-end generation dependencies. Fifty percent of Eclipse developers obtained accuracy scores below their observed medians, which in most cases was 50% less of the maximum attainable score.

ChainTracker provides a significant advantage in the accuracy of developers interpreting binding expressions in both M2M and M2T transformations. The average median difference with respect to the accuracy of developers is more pronounced for those working on PhyDSL. Our results suggest that the benefits of using traceability visualization techniques are magnified in scenarios where the ecosystems are composed in non-trivial transformation chains. It is important to mention that all developers using ChainTracker had a perfect accuracy score when identifying *element-level generation dependencies* (Tables A.13 and A.14), and *property-level generation dependencies* (Tables A.15 and A.16).

We believe that the effectiveness of developers using ChainTracker to identify multi-step generation dependencies is due to the quick access that they have to implicit and explicit binding information. We found that not only the *branch visualization* is highly useful to developers, but also the active use of editors that allow the projective interactions between graphical representations and transformation codebases.

7.5 Threats to Validity

7.5.1 Construct Validity

Do we measure what is intended? – In this study we measured the performance of developers identifying dependency relationships in M2M and M2T transformations. We compared two tools namely, ChainTracker and Eclipse Modeling (with ATL and Acceleo plugins). We used two model-driven code generators of different complexity to investigate the effect of their size on the usability of ChainTracker. Moreover, we understand developers' performance in terms of the time they take completing each task and their solution correctness.

We have developed an in-house survey application that presents participants with tasks that reflect on their ability to identify metamodel and generation dependencies at different levels of granularity, and between different artifacts of a transformation ecosystem. Our survey application has been extensively tested and did not present any failures during the execution of the study. Furthermore, we have carefully instantiated our question templates in the context of the two case

studies under consideration. The correctness of each expected solution has been validated by three model-transformation experts with 6, 1, and 2 years of experience in model-transformation technologies, respectively. The questions were designed in an iterative fashion, looking for representative tasks of different complexity inside each of the case studies.

An early version of the protocol used for this empirical study was presented in the *1st International Workshop on Human Factors in Modeling* collocated at the MODELS conference in 2015 [169]. The protocol of the study and the proposed task templates were discussed and reformulated based on feedback gained in informal meetings with industry and academic practitioners. We chose Eclipse Modeling as an industry baseline given that it is the official, and most popular, development environment for both ATL and Acceleo technologies. None of the alternative traceability visualization and analysis frameworks reviewed in Chapter 2 are available to the public. We do not see any significant threats to the construct validity of this study.

7.5.2 Internal Validity

Are there unknown factors which might affect the outcome of the experiments? – The limited number of participants and their heterogeneous expertise on model-driven development technologies is a concern for the internal validity of the study. However, this study was conducted with a pool of participants with a broad industrial development experience, and an intensive 6-month course in model transformation technologies. Considering that model-transformation languages are a fairly new technology, and are yet to be adopted by the software-engineering community at large, our pool of participants is representative of most model-driven engineering practitioners.

We are aware that the learning curve of ChainTracker and Eclipse Modeling may impact the developers performance. In order to minimize the impact of this threat to validity, we included an introductory tutorial in the training session of our protocol (Section 7.2.1). The first half of the training session involved a tutorial on the use of ChainTracker. The second half consisted of a laboratory workshop that provided hands on experience with the analysis environment. The training session was structured in two 60 minute parts divided by a 15 minute break. Our protocol did not include an introductory tutorial on Eclipse Modeling. In effect, all participants received 6 months of training in Eclipse Modeling as a part of their graduate course, which included by-weekly hands-on tutorials. Furthermore, all participants reported having used Eclipse as a development environment in professional and academic settings.

Participants were not allowed to return to a task once it was completed. This strategy might fail to account for the exploratory nature of model-transformation comprehension. Developers might want to review previously answered questions based on understanding gained throughout the course of the working sessions. Limiting our survey application to a strictly linear answering

mechanism was motivated by our desire to precisely measure the efficiency of developers solving individual tasks. We minimize this threat to validity in two ways. First, each of our working sessions includes a 15 minute presentation that explores the metamodels and transformations of their corresponding case studies. Additionally, a 10 minute window was allocated to allow further discussion on the implementation details of each ecosystem. Second, our questionnaires minimize the overlapping between segments of code that need to be analyzed throughout each session. More sophisticated mechanisms are needed to allow a more flexible answering strategy, thus increasing the generality of our results. However, the strategy used in this paper is realistic in the context of state-of-the art program comprehension studies, that measure the efficiency of developers using linear questionnaires, such as in [170, 171] and [172].

Our survey application presents three types of questions, namely, multiple-choice questions, list-based questions, and multiple-selection questions. We are aware that multiple-choice and multiple-selection questions may provide hints to participants, guiding them to investigate specific artifacts, thus reducing the precision of our performance measurements. In order to minimize this threat to validity, multiple-choice and multiple-selection questions presented a comprehensive list of artifacts needed to be considered in each task at hand. As a concrete example, tasks with the general form *"what metamodel elements are used in the creation of the [metamodel-name ! element-name] element"*, included all of the metamodel elements found in the corresponding ecosystem under analysis. This effectively avoids drawing the attention of participants to specific artifacts, as well as narrowing their search to specific segments of code[4].

Concretely, 22 out of 25 questions in both questionnaires are multiple-choice or multiple-selection questions. The remaining 3 questions are list-based, which receive one or multiple open-ended answers. List-based questions were used in tasks that require developers identifying bindings in M2T transformations. All multiple-choice and multiple-selection questions in the ScreenFlow questionnaire included a comprehensive list of their potential answers, i.e., metamodel elements and properties, transformation rules, and generated files. In the case of the PhyDSL, 20 questions provide all possible answers. The remaining 2 questions (which require the selection of metamodel element properties) provide a reduced, yet large number of answer possibilities. We do not believe that the nature of our type questions provided significant hints to developers during the completion of our study.

The study was divided in three sessions that took place over the span of a week. Our protocol was designed to minimize the fatigue of developers, and allowed them to review the training material in between sessions. We believe this can potentially increase the developers' familiarity with the proposed families of tasks. Finally, during the last two years, we have iterated over ChainTracker's

[4]An example of how this is presented to developers can be found here: `https://github.com/guana/chaintracker-eval`.

graphic user interface. We have conducted informal focus groups in order to make its features accessible and intuitive for developers. We have integrated the lessons learned in the tool demo sessions where ChainTracker has been showcased, i.e., the *International Conference on Software Maintenance and Evolution (ICSME)* in 2014 [36], the *International Conference on Model Driven Engineering Languages and Systems* in 2015 [37], and the *IBM Technology Showcase* in 2015.

7.5.3 External Validity

To what extent is it possible to generalize the findings? – The case studies of our study are two model-driven code generators implemented using ATL, a rule-based *M2M* transformation language, and Acceleo, a template-based *M2T* transformation technology. Therefore, any conclusions drawn from this study cannot be fully generalized to the performance of developers solving software-engineering tasks on other model-transformation technologies. However, both Acceleo and ATL are widely adopted by academic and industry practitioners. More importantly, both languages are aligned to the OMG's *Query/View/Transformation (QVT)* standard for M2M transformations [156], and the *Model to Text Transformation Language (MOF)* standard for M2T transformations [157], respectively. The observations of this study can potentially be generalized to developers completing the same set of tasks, in ecosystems of similar size and complexity, and built using relational and template-based transformation languages that comply with the same set of standards.

The case studies used in this study were developed in a research environment. We can not claim that the results in this study can be generalized to industrial ecosystems. However, both case studies have been used in real software-construction scenarios. Furthermore, they both have been through development cycles that included platform and metamodel evolution scenarios, in order to meet the requirements of different clients. Considering the scope of our research, and the limited availability of industrial transformation ecosystems, we believe that the results of this study provide substantial insights on how developers trace and pinpoint metamodel and generation dependencies between the artifacts of a transformation ecosystem.

Even though the case studies considered in this study are model-transformation chains, the structure of the questionnaires included tasks that investigated the performance of developers dealing with individual transformation steps, both M2M and M2T. The results of this study can be generalized to the performance of developers dealing with transformations used in isolation, as well as in non-trivial transformation chains.

8

Conclusions and Future Work

Even though MDE can be used to solve complex software-engineering tasks, its adoption among the general software-engineering community faces multiple economical, cultural, and technical challenges [2]. Development environments, specifically tailored to support the construction and maintenance of model transformations, are a fundamental requirement to increase the adoption of model-driven engineering practices [5, 11, 12]. The construction of these environments involves the analysis and visualization of traceability information [11]. Access to metamodel-level fine-grained traceability links enables developers to assess evolutionary scenarios in transformation ecosystems, to effectively debug complex binding expressions, and to accurately determine the metamodel coverage in transformation chains.

Unfortunately, current traceability-analysis techniques do not consider *implicit bindings* when collecting traceability information from complex transformation expressions, and do not conceive M2M and M2T transformations as equal constituent elements of a unified MDE toolbox. This effectively limits their usability in the construction and maintenance of non-trivial *model-driven code generators*. Furthermore, to the best of our knowledge, the effectiveness of current traceability analyses, and the development environments built on top of them, has yet to be validated in empirical studies with real developers. We believe this contributes to the skepticism of the software-engineering community regarding the evaluation rigorosity of model-driven engineering practices, as well as the pragmatism of their development tools. Let us briefly summarize the contributions of this thesis to tackle these shortcomings.

8.1 Contributions

C1: A Traceability Framework and Analysis Technique for End-to-end Traceability

In this thesis, we present a formal conceptual framework for end-to-end traceability at the metamodel level. Our conceptual formalization presents a technology-agnostic foundation towards conceiving a unified traceability vision accessible to the community at large (Chapter 4). Furthermore, we contribute a traceability-analysis technique to gather metamodel-level fine-grained traceability links from individual M2M and M2T transformations, and transformation chains combining the two (Chapter 5). Our conceptual framework and analysis techniques are generalizable to transformation languages built-on OCL.

We validated the traceability coverage of our analysis technique with 25 ATL M2M transformations and 18 Acceleo M2T transformations. The traceability coverage of our analysis was 91% and 85.4% for M2M and M2T transformations, respectively. It is important to mention that, to the best of our knowledge, current traceability-analysis techniques do not consider *implicit bindings* when collecting traceability information. Thus, our traceability coverage can not be compared effectively to that of other analysis techniques at the metamodel-level. Indeed, our analysis technique is the first of its kind to be evaluated using a non-trivial evaluation dataset.

C2: A Traceability Analysis and Usability Evaluation Dataset

We present a curated traceability evaluation dataset with 14 individual transformation projects from the *ATLZoo*, and 5 individual code generators from the *Acceleo Example Repository*. These transformations were manually inspected to characterize their binding expressions, and corresponding traceability links (Chapter 5). Furthermore, we introduced two fully featured *model-driven code generators*, namely PhyDSL and ScreenFlow. PhyDSL has been developed in the context of physics-based video games, while ScreenFlow is focused on user-interface prototyping (Chapter 3). We used PhyDSL and ScreenFlow to evaluate the usability of ChainTracker, an integrated traceability-analysis environment built on our analysis technique.

C3: The ChainTracker Integrated Traceability Analysis Environment

We introduced ChainTracker, a developer-oriented traceability analysis environment for M2M and M2T transformations, as well as for heterogeneous model transformation chains (Chapter 6). It provides transformation visualizations, projectional code editors, and contextual tables. Each area of the analysis environment is synchronized with each other and provides interactive features that in conjunction help developers to reflect about the execution of a transformation ecosystem. Our work has been driven by the belief that enabling developers to interactively explore the execution semantics of a transformation ecosystem can significantly improve their performance when reflecting on an ecosystem's design and evolution.

In order to evaluate ChainTracker's usability, we conducted an empirical study in the context of two fully-featured *model-driven code generators*, i.e. PhyDSL and ScreenFlow, and proposed a collection of traceability-driven tasks in order to understand how developers reflect on the execution semantics of a transformation ecosystem (Chapter 7). These tasks were grouped in five main families: (a) determining metamodel coverage and expression location, (b) identifying metamodel dependencies in M2M transformations, (c) identifying metamodel dependencies in M2T transformations, (d) identifying generation dependencies in M2T transformations, and (e) identifying generation dependencies in model-transformation chains. These tasks are the practical backbone of model-transformation construction and maintenance processes. To recapitulate, the two main research questions driving our empirical study are discussed below.

RQ1*: Do developers using ChainTracker identify metamodel and generation dependencies in transformation ecosystems more accurately and efficiently than those using Eclipse Modeling?*

- Developers using ChainTracker had statistically significantly higher performance identifying metamodel dependencies in M2M transformations than those supported by Eclipse Modeling, i.e., 83% in *element-level dependencies*, and between 36% and 900% in *property-level dependencies*.

- Developers assessing the coverage of metamodels using ChainTracker were between 72% and 200% more accurate, and 46% more efficient than those using Eclipse Modeling.

- Due to the lack of pattern-matching capabilities in ChainTracker, its developers performed worse in finding individual transformation expressions, i.e., 63% less accurate. We believe supporting this type of task is extremely useful to detect duplicate binding expressions, i.e., code clones, as well as identifying transformation refactoring opportunities. Our future work includes extending ChainTracker with pattern-matching capabilities to efficiently support developers locating bindings of interest in large transformation codebases.

- The performance of developers using Eclipse Modeling was much worse identifying metamodel downstream-element dependencies than metamodel upstream-element dependencies, in M2M transformations. This is mostly due to the large code spaces that need to be manually analyzed if not supported by automatic reasoning tools. The impact of ChainTracker on the performance of developers was significantly higher when they were required to analyze model transformation chains, and to determine end-to-end metamodel dependencies caused by the interdependent execution of M2M and M2T transformations.

- Manually examining large transformation codebases, looking for the usage of dozens of metamodel elements is a daunting, if not impossible, task. Developers using Eclipse performed very poorly identifying generation dependencies in model transformation chains. Most of

them obtained accuracy scores below the observed medians for this type of task, which overall was less than 50% of the maximum attainable score. Conversely, developers supported by ChainTracker were 100% accurate, and were on average 60% more efficient.

- Our empirical observations suggest that interactive traceability visualizations are as useful as runtime transformation debuggers, to identify generation dependencies in M2T transformations. In the case of model-transformation chains, developers using ChainTracker were able to identify *element-* and *property-level* generation dependencies in transformation chains 66% more efficiently and 68% more accurately, than developers using Eclipse Modeling.

RQ2*: Do the size and complexity of transformation ecosystems affect the effectiveness of ChainTracker in helping developers identify their metamodel and generation dependencies?*

- For the five families of traceability-driven tasks in this study, their complexity is considerably exacerbated by the size of the transformations, and the number of metamodel elements, in the ecosystem under analysis. The impact of ChainTracker on the performance of developers was more pronounced in the context of a multi-branched transformation chain, i.e., PhyDSL, than in a linear-transformation ecosystem, i.e., ScreenFlow.

- We believe that the overall higher performance of developers using ChainTracker is due to the support that it provides identifying implicit and explicit bindings in complex OCL expressions. Considering that most developers are used to the execution semantics of imperative programing languages, ChainTracker considerably lowers the cognitive challenges that they face when getting used to declarative programing semantics. In effect, the use of editors that allow two-way interactions between visualizations and transformation codebases was proved as an effective mechanism to investigate metamodel and generation dependencies.

8.2 Future Work

Our future avenues of investigation include the development of additional static-analysis features to collect traceability links from *cross-referenced target elements, helper attributes,* and *local variables* defined in *matched rules*, in the context of M2M transformations. We recognize these expressions can be potentially more predominant in industrial-size transformation ecosystems. Furthermore, we are working on analysis capabilities to collect traceability links from *template queries* and *cross-referenced templates* in the context of M2T transformations. Considering the state-of-the-art of metamodel-level traceability-analysis approaches, we believe our work is an important contribution towards increasing their technical robustness, and their evaluation rigorosity.

Our traceability-analysis technique focuses on completely declarative model transformations [93]. However, transformation languages such as ATL and ETL implement a hybrid approach with both *declarative* and *imperative* semantics. In ATL, for example, *action blocks* enable developers to derive model elements using imperative instructions. Our future work includes extending our analysis technique to collect traceability information from imperative binding expressions in hybrid transformation languages, and other *direct manipulation approaches* (Chapter 2.3).

In this thesis, we gained important insights about the performance of developers using ChainTracker and Eclipse Modeling. Furthermore, we increased the understanding of the field on the challenges developers face while interpreting the execution semantics of non-trivial transformation ecosystems. However, further research is needed to identify alternative developer-oriented features to be implemented in ChainTracker.

The ChainTracker *branch visualization* portrays metamodel elements without explicitly presenting their *inheritance* and *containment* relationships. We are currently developing synchronized visualizations that enable developers use the *branch visualization* to reflect on an element's transformation dependencies, while at the same time, obtaining filtered and contextual information using their classic (UML-like) syntax. We believe this is particularly useful to developers assessing the impact of changes in transformation ecosystems with highly hierarchical metamodels. Future extensions to the ChainTracker *branch visualization* also include the development of context-aware tooltips, as a complementary feature to contextual tables, in order to make attribute names more easily accessible to developers.

A natural extension of our work includes the integration of ChainTracker with metamodel and transformation version control systems (VCS) [173, 174, 175]. We believe this will further enable developers to explore the impact of changes in different scenarios of evolution, and at different points during the life-cycle of a transformation ecosystem. Furthermore, research work in the field of *model management and analysis tools as service (MaaS)* [176] motivates the integration of our traceability-analysis engine into web-based development tools, e.g., MDEForge [115], by means of Web APIs that expose the ChainTracker analysis and visualization services. Moreover, with the advance of scalable model-driven engineering [177], and the emerging fields of parallel model transformation execution [178], we want to extend ChainTracker with visualizations that enable developers monitoring the execution of transformation chains in distributed computational platforms.

Finally, the software-engineering community requires access to empirical evidence regarding the advantages of model-driven engineering techniques, and truthful insights about its limitations, in order to invest in significant model-driven development efforts. Access to industrial-size model-driven development infrastructures with realistic life-cycles, is a challenging endeavor. Our future

work includes investigating the characteristics of metamodel and platform evolution scenarios in non-academic *model-driven code generators*. We would like to evaluate the performance of developers completing traceability-driven tasks to fix, synchronize, and optimize the design of industrial-size transformation ecosystems using ChainTracker.

Bibliography

[1] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?" in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 1–17.

[2] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 471–480.

[3] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[4] Z. Hemel, L. Kats, D. Groenewegen, and E. Visser, "Code generation by model transformation: a case study in transformation modularity," *International Journal on Software and Systems Modeling*, vol. 9, no. 3, pp. 375–402.

[5] F. Tomassetti, M. Torchiano, A. Tiso, F. Ricca, and G. Reggio, "Maturity of software modelling and model driven engineering: A survey in the italian industry," in *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*. IET, 2012, pp. 91–100.

[6] J. Sprinkle and G. Karsai, "A domain-specific visual language for domain model evolution," *Journal of Visual Languages & Computing*, vol. 15, no. 3, pp. 291–307, 2004.

[7] A. Van Deursen, E. Visser, and J. Warmer, "Model-driven software evolution: A research agenda," in *Proceedings 1st International Workshop on Model-Driven Software Evolution*, 2007, pp. 41–49.

[8] K. Bennett and V. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 73–87.

[9] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems," in *International Conference on Graph Transformation*. Springer, 2012, pp. 20–37.

[10] V. Guana and E. Stroulia, "End-to-end model-transformation comprehension through fine-grained traceability information," in *International Journal on Software and Systems Modeling*, 2017.

[11] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[12] M. F. Van Amstel and M. G. Van Den Brand, "Model transformation analysis: Staying ahead of the maintenance nightmare," in *Theory and Practice of Model Transformations*. Springer, 2011, pp. 108–122.

[13] C. F. Kemerer, "Now the learning curve affects case tool adoption," *IEEE Software*, vol. 9, no. 3, pp. 23–28, 1992.

[14] B. C. Hardgrave, F. D. Davis, and C. K. Riemenschneider, "Investigating determinants of software developers' intentions to follow methodologies," *Journal of Management Information Systems*, vol. 20, no. 1, pp. 123–151, 2003.

[15] S. Walderhaug, E. Stav, U. Johansen, and G. K. Olsen, "Traceability in model-driven software development," *Designing Software-Intensive Systems: Methods and Principle*, pp. 133–159, 2008.

[16] A. Von Knethen and M. Grund, "Quatrace: a tool environment for (semi-) automatic impact analysis based on traces," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2003, pp. 246–255.

[17] S. Winkler and J. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *International Journal on Software and Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

[18] B. Amar, H. Leblanc, and B. Coulette, "A traceability engine dedicated to model transformation for software engineering," in *ECMDA Traceability Workshop (ECMDA-TW)*, 2008, pp. 7–16.

[19] R. Brcina and M. Riebisch, "Defining a traceability link semantics for design decision support," in *ECMDA Traceability Workshop (ECMDA-TW)*, 2008, pp. 39–48.

[20] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Traceability visualization in metamodel change impact detection," in *Proceedings of the Second Workshop on Graphical Modeling Language Development*. ACM, 2013, pp. 51–62.

[21] M. van Amstel, A. Serebrenik, and M. van den Brand, "Visualizing traceability in model transformation compositions," in *Pre-proceedings of the First Workshop on Composition and Evolution of Model Transformations*, 2011.

[22] J. Wang, S.-K. Kim, and D. Carrington, "Verifying metamodel coverage of model transformations," in *Australian Software Engineering Conference*. IEEE, 2006, pp. 10–pp.

[23] J. Wang, S. Kim, and D. Carrington, "Automatic generation of test models for model transformations," in *Australian Conference on Software Engineering*. IEEE, 2008, pp. 432–440.

[24] O. Finot, J.-M. Mottu, G. Sunyé, and T. Degueule, "Using meta-model coverage to qualify test oracles," in *Analysis of Model Transformations*, 2013, pp. 1613–0073.

[25] I. Galvao and A. Goknil, "Survey of traceability approaches in model-driven engineering," in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007, pp. 313–313.

[26] I. Santiago, A. Jiménez, J. M. Vara, V. De Castro, V. A. Bollati, and E. Marcos, "Model-driven engineering as a new landscape for traceability management: A systematic literature review," *Information and Software Technology*, vol. 54, no. 12, pp. 1340–1356, 2012.

[27] J.-R. Falleri, M. Huchard, and C. Nebut, "Towards a traceability framework for model transformations in kermeta," in *ECMDA-TW'06: ECMDA Traceability Workshop*. Sintef ICT, Norway, 2006, pp. 31–40.

[28] F. Jouault, "Loosely coupled traceability for ATL," in *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*, vol. 91. Citeseer, 2005.

[29] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers, "Constructing and visualizing transformation chains," in *Model Driven Architecture–Foundations and Applications*. Springer, 2008.

[30] N. D. Matragkas, D. S. Kolovos, R. F. Paige, and A. Zolotas, "A traceability-driven approach to model transformation testing." in *AMT@MoDELS*, 2013.

[31] I. Santiago, J. M. Vara, M. V. de Castro, and E. Marcos, "Towards the effective use of traceability in model-driven engineering projects," in *Conceptual Modeling*. Springer, 2013, pp. 429–437.

[32] B. Grammel, S. Kastenholz, and K. Voigt, "Model matching for trace link generation in model-driven software development," in *Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 609–625.

[33] D. Di Ruscio, L. Iovino, and A. Pierantonio, "Managing the coupled evolution of metamodels and textual concrete syntax specifications," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*. IEEE, 2013, pp. 114–121.

[34] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," in *International Conference on Software Language Engineering*. Springer, 2012, pp. 144–163.

[35] V. Guana and E. Stroulia, "Chaintracker, a model-transformation trace analysis tool for code-generation environments," in *Theory and Practice of Model Transformations*. Springer, 2014, pp. 146–153.

[36] V. Guana, K. Gaboriau, and E. Stroulia, "Chaintracker: Towards a comprehensive tool for building code-generation environments," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 613–616.

[37] V. Guana and E. Stroulia, "Reflecting on model-based code generators using traceability information," in *Model Driven Engineering Languages and Systems*, 2015.

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

[39] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, vol. 2, 2006.

[40] J. Warmer and A. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[41] V. Guana and E. Stroulia, "Phydsl: A code-generation environment for 2d physics-based games," in *2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM)*, 2014.

[42] V. Guana, E. Stroulia, and V. Nguyen, "Building a game engine: A tale of modern model-driven engineering," in *Fourth International Workshop on Games and Software Engineering (GAS 2015)*, 2015.

[43] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, "Declarative versus imperative process modeling languages: The issue of understandability," in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2009, pp. 353–366.

[44] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[45] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[46] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering-Designing, Implementing and Using Domain-Specific Languages*. dslbook. org, 2013.

[47] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[48] S. Beydeda, M. Book, V. Gruhn *et al.*, *Model-driven software development*. Springer, 2005, vol. 15.

[49] R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, p. 308, 2000.

[50] J. Bézivin, "In search of a basic principle for model driven engineering," *Novatica Journal, Special Issue*, vol. 5, no. 2, pp. 21–24, 2004.

[51] J.-M. Favre, "Towards a basic theory to model model driven engineering," in *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[52] K. Czarnecki, M. Antkiewicz, C. Kim, S. Lau, and K. Pietroszek, "Model-driven software product lines," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 126–127.

[53] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3.

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

[55] H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, *Graph Transformations*. Springer, 2008.

[56] D. Varró and A. Balogh, "The model transformation language of the viatra2 framework," *Science of Computer Programming*, vol. 68, no. 3, pp. 214–234, 2007.

[57] A. Schürr, A. J. Winter, and A. Zündorf, "The progres approach: Language and environment," in *Handbook of graph grammars and computing by graph transformation*. World Scientific Publishing Co., Inc., 1999, pp. 487–550.

[58] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai, "The graph rewriting and transformation language: GReAT," *Electronic Communications of the EASST*, vol. 1, 2007.

[59] J. De Lara and H. Vangheluwe, "Atom3: A tool for multi-formalism and meta-modelling," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2002, pp. 174–188.

[60] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place emf model transformations," *Model Driven Engineering Languages and Systems*, pp. 121–135, 2010.

[61] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[62] J. Cuadrado, J. Molina, and M. Tortosa, "Rubytl: A practical, extensible transformation language," in *Model Driven Architecture–Foundations and Applications*. Springer, 2006, pp. 158–172.

[63] L. Rose, R. Paige, D. Kolovos, and F. Polack, "The epsilon generation language," in *Model Driven Architecture–Foundations and Applications*. Springer, 2008, pp. 1–16.

[64] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon object language (EOL)," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2006, pp. 128–142.

[65] P. Stevens, "A landscape of bidirectional model transformations," in *Generative and transformational techniques in software engineering II*. Springer, 2008, pp. 408–424.

[66] N. Kahani and J. R. Cordy, "Comparison and evaluation of model transformation tools," Technical Report 2015-627, December, Tech. Rep., 2015.

[67] P. Boocock, "Jamda: The Java model driven architecture." *URL:http://jamda.sourceforge.net/*, 2003.

[68] T. Degueule, B. Combemale, A. Blouin, and O. Barais, "Reusing legacy DSLs with Melange," in *Proceedings of the Workshop on Domain-Specific Modeling*. ACM, 2015, pp. 45–46.

[69] A. Kleppe, "First european workshop on composition of model transformations - cmt 2006," *Technical Report TR-CTIT-06-34*, 2006.

[70] K. Czarnecki, "Generative programming: Methods, techniques, and applications tutorial abstract," *Software Reuse: Methods, Techniques, and Tools*, pp. 477–503, 2002.

[71] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," *Model Driven Engineering Languages and Systems*, pp. 623–637, 2011.

[72] D. N. Card, G. T. Page, and F. E. McGarry, "Criteria for software modularization," in *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 1985, pp. 372–377.

[73] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[74] I. Kurtev, K. Van den Berg, and F. Jouault, "Evaluation of rule-based modularization in model transformation languages illustrated with atl," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 1202–1209.

[75] L. Geiger, C. Schneider, and C. Reckord, "Template- and model-based code generation for MDA-Tools," in *Proceedings of the 3rd International Fujaba Days*, 2005, pp. 57–62.

[76] G. Spanoudakis and A. Zisman, "Software traceability: a roadmap," *Handbook of Software Engineering and Knowledge Engineering*, vol. 3, pp. 395–428, 2005.

[77] K. Pohl, *Process-centered requirements engineering*. John Wiley & Sons, Inc., 1996.

[78] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE transactions on software engineering*, vol. 27, no. 1, pp. 58–93, 2001.

[79] R. Oliveto, G. Antoniol, A. Marcus, and J. Hayes, "Software artefact traceability: the never-ending challenge," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 485–488.

[80] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE transactions on software engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[81] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 125–135.

[82] A. Zisman, G. Spanoudakis, E. Pérez-Miñana, and P. Krause, "Tracing software requirements artifacts." in *Software Engineering Research and Practice*, 2003, pp. 448–455.

[83] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 4, p. 13, 2007.

[84] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[85] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 190–198.

[86] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[87] H. Kagdi and J. Maletic, "Software repositories: A source for traceability links," in *International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSEâ07)*, 2007, pp. 32–39.

[88] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 68–71.

[89] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Can information retrieval techniques effectively support traceability link recovery?" in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 307–316.

[90] G. Karsai and A. Narayanan, "On the correctness of model transformations in the development of embedded systems," in *Composition of Embedded Systems. Scientific and Industrial Issues*. Springer, 2007, pp. 1–18.

[91] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, "Using traceability to enhance mutation analysis dedicated to model transformation," in *Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa)*. IEEE, 2010, pp. 1–6.

[92] E. Bauer, J. M. Küster, and G. Engels, "Test suite quality for model transformation chains," in *Objects, Models, Components, Patterns*. Springer, 2011, pp. 3–19.

[93] M. van Amstel and M. van den Brand, "Quality assessment of atl model transformations using metrics," in *Proceedings of the 2nd International Workshop on Model Transformation with ATL, Malaga, Spain*, 2010.

[94] A. Correa and C. Werner, "Applying refactoring techniques to uml/ocl models," in *« UML» 2004-The Unified Modeling Language. Modelling Languages and Applications*. Springer, 2004, pp. 173–187.

[95] C. Ermel, H. Ehrig, and K. Ehrig, "Refactoring of model transformations," *Electronic Communications of the EASST*, vol. 18, 2009.

[96] J. Almeida, P. Van Eck, and M. Iacob, "Requirements traceability and transformation conformance in model-driven development," in *International Enterprise Distributed Object Computing Conference*. IEEE, 2006, pp. 355–366.

[97] T. project (IRISA), *The metamodeling language kermeta*. http://www.kermeta.org, 2006.

[98] M. F. van Amstel, M. G. van den Brand, and A. Serebrenik, "Traceability visualization in model transformations with tracevis," in *International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, 2012, pp. 152–159.

[99] O. M. Group, "A proposal for an mda foundation model," *Needham ormsc/05-04-01 ed*, 2005.

[100] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model traceability," *IBM Systems Journal*, vol. 45, no. 3, pp. 515–526, 2006.

[101] J. Oldevik and T. Neple, "Traceability in model to text transformations," in *2nd ECMDA Traceability Workshop (ECMDA-TW)*. Citeseer, 2006, pp. 17–26.

[102] J. García, M. Azanza, A. Irastorza, and O. Díaz, "Testing mofscript transformations with handymof," in *Theory and Practice of Model Transformations*. Springer, 2014, pp. 42–56.

[103] G. K. Olsen and J. Oldevik, "Scenarios of traceability in model to text transformations," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2007, pp. 144–156.

[104] I. Santiago, J. M. Vara, V. de Castro, and E. Marcos, "Reducing the level of complexity of working with model transformations," in *International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer, 2014, pp. 1–17.

[105] F. Glitia, A. Etien, and C. Dumoulin, "Fine grained traceability for an mde approach of embedded system conception," in *ECMDA Traceability Workshop*. Citeseer, 2008, pp. 27–38.

[106] V. Guana, "Supporting maintenance tasks on transformational code generation environments," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1369–1372.

[107] "Atl transformation example: Uml to java," https://www.eclipse.org/atl/atlTransformations/UML2Java/ExampleUML2Java[v00.01].pdf, accessed: 2017-06-07.

[108] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes, "Engineering a dsl for software traceability," in *International Conference on Software Language Engineering*. Springer, 2008, pp. 151–167.

[109] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Merging models with the epsilon merging language (eml)," in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 215–229.

[110] B. Grammel and S. Kastenholz, "A generic traceability framework for facet-based traceability data extraction in model-driven software development," in *Proceedings of the 6th ECMFA Traceability Workshop*. ACM, 2010, pp. 7–14.

[111] "Atlas transformation language (ATL) user guide," http://goo.gl/KzPaze, accessed: 2016-09-14.

[112] "Acceleo traceability: Eclipse plug-in," http://goo.gl/eenOE3, accessed: 2016-09-14.

[113] A. Marcus, X. Xie, and D. Poshyvanyk, "When and how to visualize traceability links?" in *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. ACM, 2005, pp. 56–61.

[114] L. Iovino, A. Pierantonio, and I. Malavolta, "On the impact significance of metamodel evolution in mde." *Journal of Object Technology*, vol. 11, no. 3, pp. 3–1, 2012.

[115] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "Mdeforge: an extensible web-based modeling platform." in *CloudMDE@ MoDELS*, 2014, pp. 66–75.

[116] F. Jouault, J. Bézivin, and I. Kurtev, "TCS:: a DSL for the specification of textual concrete syntaxes in model engineering," in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 249–254.

[117] A. Gamatié, S. Le Beux, É. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 10, no. 4, p. 39, 2011.

[118] A. van Lamsweerde, "Goal-oriented requirements engineering: From system objectives to uml models to precise software specifications," in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 744–745.

[119] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—open source graph drawing tools," in *Graph Drawing*. Springer, 2002, pp. 483–484.

[120] J. Von Pilgrim, K. Duske, and P. McIntosh, "Eclipse gef3d: Bringing 3d to existing 2d editors," *Information Visualization*, vol. 8, no. 2, pp. 107–119, 2009.

[121] R. Wieringa, "An introduction to requirements traceability," *Free University, Faculty of Mathematics and Computer Science*, 1995.

[122] F. A. Pinheiro, "Requirements traceability," in *Perspectives on software requirements*. Springer, 2004, pp. 91–113.

[123] C. Duan and J. Cleland-Huang, "Visualization and analysis in automated trace retrieval," in *2006 First International Workshop on Requirements Engineering Visualization (REV'06-RE'06 Workshop)*. IEEE, 2006, pp. 5–5.

[124] D. N. Card, "Designing software for producibility," *Journal of Systems and Software*, vol. 17, no. 3, pp. 219–225, 1992.

[125] J. Cleland-Huang, "Toward improved traceability of non-functional requirements," in *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2005, pp. 14–19.

[126] J. Cleland-Huang, O. C. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 55–69.

[127] 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 *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 2013, p. 2.

[128] R. Wettel and M. Lanza, "Program comprehension through software habitability," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 2007, pp. 231–240.

[129] H. Störrle, "On the impact of size to the understanding of uml diagrams," *international Journal on Software and Systems Modeling*, pp. 1–20, 2016.

[130] W. van Ravensteijn, "Visual traceability across dynamic ordered hierarchies," 2011.

[131] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[132] T. Tong, V. Guana, A. Jovanovic, F. Tran, G. Mozafari, M. Chignell, and E. Stroulia, "Rapid deployment and evaluation of mobile serious games: A cognitive assessment case study," *Procedia Computer Science*, vol. 69, pp. 96–103, 2015.

[133] B. Wallace, F. Knoefel, R. Goubran, P. Masson, A. Baker, B. Allard, E. Stroulia, and V. Guana, "Monitoring cognitive ability in patients with moderate dementia using a modified "whack-a-mole," *12th IEEE International Symposium on Medical Measurements and Applications*, 2017.

[134] J. Kuittinen, A. Kultima, J. Niemelä, and J. Paavilainen, "Casual games discussion," in *Proceedings of the 2007 conference on Future Play*. ACM, 2007, pp. 105–112.

[135] E. McDonald. (2017) The global games market will reach $108.9 billion in 2017 with mobile taking 42%. New Zoo. [Online]. Available: https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42

[136] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, p. 28, 2004.

[137] D. Callele, E. Neufeld, and K. Schneider, "Requirements engineering and the creative process in the video game industry," in *International Conference on Requirements Engineering*. IEEE, 2005, pp. 240–250.

[138] S. Tang and M. Hanneghan, "State-of-the-art model driven game development: A survey of technological solutions for game-based learning," *Journal of Interactive Learning Research*, vol. 22, no. 4, pp. 551–605, December 2011.

[139] H. Desurvire, M. Caplan, and J. A. Toth, "Using heuristics to evaluate the playability of games," in *CHI'04 Human Factors in Computing Systems*. ACM, 2004, pp. 1509–1512.

[140] D. Pinelle, N. Wong, and T. Stach, "Heuristic evaluation for games: Usability principles for video game design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1453–1462.

[141] P. N. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983, no. 6.

[142] S. Tang and M. Hanneghan, "State-of-the-art model driven game development: a survey of technological solutions for game-based learning," *Journal of Interactive Learning Research*, vol. 22, no. 4, pp. 551–605, 2011.

[143] A. W. Furtado and A. L. Santos, "Using domain-specific modeling towards computer games development industrialization," in *The 6th OOPSLA Workshop on Domain-Specific Modeling (DSM06)*. Citeseer, 2006.

[144] E. M. Reyno and J. Á. C. Cubel, "Model driven game development: 2d platform game prototyping." in *GAMEON*, 2008, pp. 5–7.

[145] J. D. Palmer, "Ficticious: Microlanguages for interactive fiction," in *international Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 2010, pp. 61–68.

[146] S. A. Robenalt, "Mdsd for games with eclipse modeling technologies," in *Entertainment Computing-ICEC 2012*. Springer, 2012, pp. 511–517.

[147] C. Karamanos and N. M. Sgouros, "Automating the implementation of games based on model-driven authoring environments," in *Entertainment Computing-ICEC 2012*. Springer, 2012, pp. 524–529.

[148] R. Hunicke, M. LeBlanc, and R. Zubek, "Mda: A formal approach to game design and game research," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004, pp. 04–04.

[149] S. Rogers, *Level Up!: The Guide to Great Video Game Design*. John Wiley & Sons, 2010.

[150] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. IEEE, 2013, pp. 323–328.

[151] "ATL transformation example: Book to publication," https://www.eclipse.org/atl/atlTransformations/Book2Publication/ExampleBook2Publication[v00.02].pdf, accessed: 2017-06-07.

[152] L. Burgueno, "Testing M2M/M2T/T2M transformations," *International Conference on Model Driven Engineering Languages and Systems (Student Competition)*, 2015.

[153] F. Jouault and J. Bézivin, "KM3: a DSL for metamodel specification," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2006, pp. 171–185.

[154] Eclipse.org, "Eclipse Modeling Framework (EMF) Integration," https://eclipse.org/Xtext/documentation/308_emf_integration.html, 2017, [Online; accessed 27-March-2017].

[155] A. Correa and C. Werner, "Refactoring object constraint language specifications," *International Journal on Software and Systems Modeling*, vol. 6, no. 2, pp. 113–138, 2007.

[156] OMG, "Mof model to text transformation language (mofm2t), 1.0," 2008.

[157] OMG., "Meta object facility (mof) 2.0 query/view/transformation (qvt)," 2015.

[158] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[159] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray, "Globalizing modeling languages," *IEE Computer*, pp. 10–13, 2014.

[160] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution framework of the gemoc studio (tool demo)," in *International Conference on Software Language Engineering*. ACM, 2016, pp. 84–89.

[161] D. Soni, R. L. Nord, and C. Hofmeister, "Software architecture in industrial applications," in *International Conference on Software Engineering (ICSE)*. IEEE, 1995, pp. 196–196.

[162] A. Kleppe, "First european workshop on composition of model transformations-cmt 2006," 2006.

[163] E. J. Wegman, "Hyperdimensional data analysis using parallel coordinates," *Journal of the American Statistical Association*, vol. 85, no. 411, pp. 664–675, 1990.

[164] A. Inselberg and B. Dimsdale, "Parallel coordinates: a tool for visualizing multi-dimensional geometry," in *Proceedings of the 1st conference on Visualization'90*. IEEE Computer Society Press, 1990, pp. 361–378.

[165] J. Juhár and L. Vokorokos, "Understanding source code through projectional editor," in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*. IEEE, 2015, pp. 1–4.

[166] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *IEEE Computer*, vol. 49, no. 7, pp. 44–52, 2016.

[167] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio, "Are fit tables really talking?" in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 361–370.

[168] F. Ricca, M. Leotta, G. Reggio, A. Tiso, G. Guerrini, and M. Torchiano, "Using unimod for maintenance tasks: An experimental assessment in the context of model driven development," in *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE Press, 2012, pp. 77–83.

[169] V. Guana and E. Stroulia, "How do developers solve software-engineering tasks on model-based code generators? An empirical study design," in *First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS*, 2015, pp. 33–38.

[170] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, no. 2, pp. 115–156, 2002.

[171] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? a controlled experiment on scratch programs," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[172] C. Gravino, M. Risi, G. Scanniello, and G. Tortora, "Do professional developers benefit from design pattern documentation? a replication in the context of source code comprehension," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 185–201.

[173] A. Cicchetti, F. Ciccozzi, T. Leveque, and A. Pierantonio, "On the concurrent versioning of metamodels and models: Challenges and possible solutions," in *Proceedings of the 2nd International Workshop on Model Comparison in Practice*. ACM, 2011, pp. 16–25.

[174] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "A metamodel independent approach to difference representation." *Journal of Object Technology*, vol. 6, no. 9, pp. 165–185, 2007.

[175] Y. Lin, J. Gray, and F. Jouault, "Dsmdiff: a differentiation tool for domain-specific models," *European Journal of Information Systems*, vol. 16, no. 4, pp. 349–361, 2007.

[176] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Collaborative repositories in model-driven engineering [software technology]," *IEEE Software*, vol. 32, no. 3, pp. 28–34, 2015.

[177] D. Di Ruscio, D. Kolovos, and N. Matragkas, "Scalability in model driven engineering: Bigmde'13 workshop summary," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 2013, p. 1.

[178] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, "On the concurrent execution of model transformations with linda," in *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 2013, p. 3.

A

Appendix

A.1 Evaluation Questionnaires

Table A.1: Session A: ScreenFlow Questionnaire

N.	Question
Q1	What ATL script contains the 'Trigger2Button' rule?
Q2	In Mockup2GUI.atl, what transformation rule contains the binding expression `isMain<-mockscreen.main.toString().endsWith('true')`?
Q3	What metamodel elements are used in the creation of the GUI!Application element?
Q4	What metamodel elements are created using the Mockup!TriggerSection element?
Q5	Considering the entire transformation chain, what metamodel elements does the template line 24 in generateControlles.mtl depend on?
Q6	What template lines in generateControlles.mtl are used in the generation of line 21 in PlayerActivity.java?
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 4 in login.xml depend on?
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 34 in AndroidManifest.xml depend on?
Q9	What files does the generateControllers.mtl template generate?
Q10	What files does the generateViews.mtl template generate?
Q11	What metamodel elements are used in the creation of the GUI!Screen element?
Q12	What metamodel elements are created using the Mockup!ScreenSection element?
Q13	What metamodel elements are created using the property 'name' of the Mockup!Screen element?
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 17 in login_activity.xml depend on?
Q15	Considering the entire transformation chain, what metamodel elements does the template line 19 in generateViews.mtl depend on?
Q16	What template lines in generateControllers.mtl are used in the generation of line 37 in LoginActivity.java?
Q17	Are there any unused elements in the Mockup metamodel? If so, which ones?
Q18	Are there any unused elements in the GUI metamodel?
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in AndroidManifest.xml depend on?
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 38 in LoginActivity.java depend on?
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in login_activity.xml depend on?
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 41-44 in generateControllers.mtl depend on?
Q23	What metamodel elements are created using the property 'fromScreen' of the Mockup!Transition element?
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 8 in login_activity.xml depend on?
Q25	What template lines in generateViews.mtl are used in the generation of line 27 in AndroidManifest.java?

Table A.2: Session B: PhyDSL Questionnaire

N.	Question
Q1	What ATL script contains the 'Effect2Action' rule?
Q2	In Game2Layout.atl, what transformation rule contains the binding expression `isBall<-r.actor Definition.first().isBall.boolean.solveBool()`?
Q3	What metamodel elements are used in the creation of the Scoring!TouchRule element?
Q4	What metamodel elements are created using the PhyDSL!Coordinate element?
Q5	Considering the entire transformation chain, what metamodel elements does the template line 148 in generateDynamics.mtl depend on?
Q6	What template lines in generateScoring.mtl are used in the generation of line 102 in ScoringManager.java?
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 100 in ScoringManager.java depend on?
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 220 in DrawingHelper.java depend on?
Q9	What files does the generateLayout.mtl template generate?
Q10	What files does the generateGraphics.mtl template generate?
Q11	What metamodel elements are used in the creation of the Scoring!CollisionRule element?
Q12	What metamodel elements are created using the PhyDSL!Ends element?
Q13	What metamodel elements are created using the property 'name' of the PhyDSL!Actor element?
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 76 in ScoringManager.java depend on?
Q15	Considering the entire transformation chain, what metamodel elements does the template line 110 in generateScoring.mtl depend on?
Q16	What template lines in generateLayout.mtl are used in the generation of line 264 in PhysicsView.java?
Q17	Are there any unused elements in the PhyDSL metamodel? If so, which ones?
Q18	Are there any unused elements in the Dynamics metamodel? If so, which ones?
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 141 in MainActivity.java depend on?
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 29 in ControlManager.java depend on?
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 281 in PhysicsView.java depend on?
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 104-108 in generateControls.mtl depend on?
Q23	What metamodel elements are created using the property 'value' of the PhyDSL!BooleanType element?
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 18 in ControlManager.java depend on?
Q25	What template lines in generateControls.mtl are used in the generation of line 99 in ControlManager.java?

A.2 ChainTracker Evaluation Result Tables

Table A.3: Session A (ScreenFlow) - Results: Metamodel Coverage and Expression Location

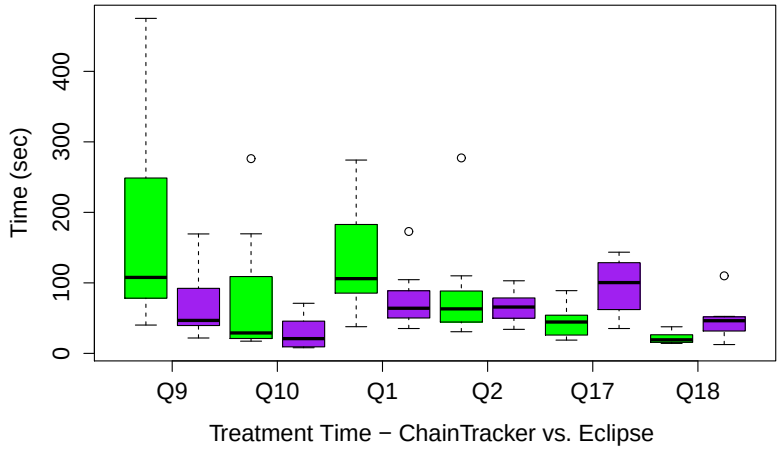
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q9	What files does the generateControllers.mtl template generate?	0.3496	0.0721
Q10	What files does the generateViews.mtl template generate?	0.1423	0.2319
Q1	What ATL script contains the 'Trigger2Button' rule?	0.4227	0.1206
Q2	In Mockup2GUI.atl, what transformation rule contains the binding expression `isMain`...?	1.0000	0.9551
Q17	Are there any unused elements in the Mockup metamodel?	**0.0066***	**0.0093***
Q18	Are there any unused elements in the GUI metamodel?	1.0000	**0.0400***

Treatments Time Mean and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q9	182.14	157.90	68.00	48.21
Q10	82.71	100.89	28.62	25.45
Q1	135.71	81.56	76.25	43.78
Q2	90.42	86.21	65.25	21.94
Q17	44.00	24.07	94.62	38.87
Q18	21.85	8.53	47.25	28.96

Treatments Score Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.78	0.56	2.00	0.00
5	3.92	2.24	5.00	0.00
1	1.00	0.00	0.75	0.70
1	1.00	0.00	1.00	0.00
2	1.28	1.34	-0.37	1.62
1	1.00	0.00	1.00	0.00

Median Time Comparison	
CT	**EC**
107.00	**46.00**
28.00	**20.50**
105.00	**63.50**
90.42	**65.00**
44.00	100.00
19.00	46.00

Median Score Comparison	
CT	**EC**
2.00	2.00
5.00	5.00
1.00	1.00
1.00	1.00
2.00	-1.50
1.00	1.00



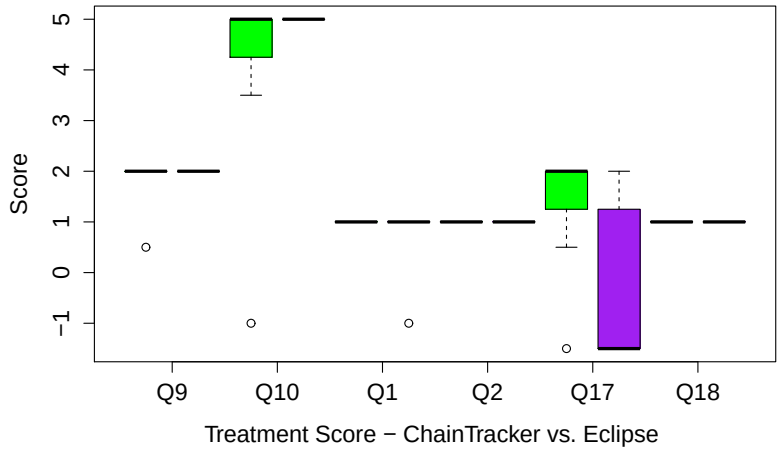


Table A.4: Session B (PhyDSL) - Results: Metamodel Coverage and Expression Location

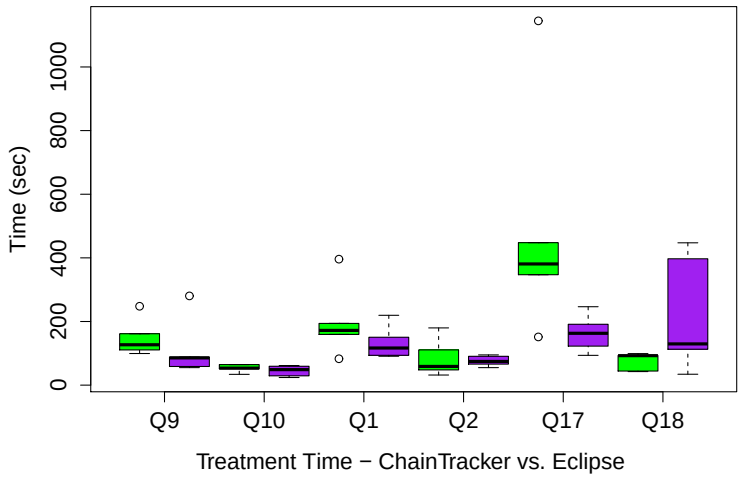
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q9	What files does the generateLayout.mtl template generate?	1.0000	0.1508
Q10	What files does the generateGraphics.mtl template generate?	1.0000	0.3095
Q1	What ATL script contains the 'Effect2Action' rule?	1.0000	0.4206
Q2	In Game2Layout.atl, what transformation rule contains the binding expression `isBall`...	1.0000	0.8413
Q17	Are there any unused elements in the PhyDSL metamodel?	0.1563	0.0555
Q18	Are there any unused elements in the Dynamics metamodel?	0.09296	0.1508

Treatments Time Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q9	148.60	59.78	113.20	0.00
Q10	53.00	12.70	44.00	17.29
Q1	200.00	116.74	133.60	53.43
Q2	85.00	60.31	75.60	17.06
Q17	493.80	379.79	162.80	59.69
Q18	74.00	28.43	223.80	185.28

Treatments Score Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	1.00	0.00	1.00	0.00
1	0.09	1.02	-0.80	0.44
1	0.60	0.89	-0.60	0.89

Median Time Comparison	
CT	**EC**
126.00	**85.00**
53.00	**48.00**
171.00	**116.00**
58.00	74.00
380.00	**162.00**
92.00	129.00

Median Score Comparison	
CT	**EC**
1.00	1.00
1.00	1.00
1.00	1.00
1.00	1.00
0.50	-1.00
1.00	-1.00



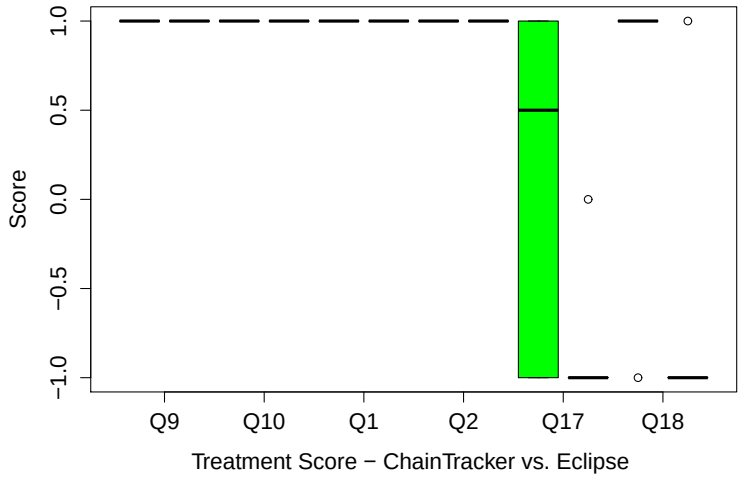


Table A.5: Session A (ScreenFlow) - Results: Metamodel Dependencies in M2M (Element Level)

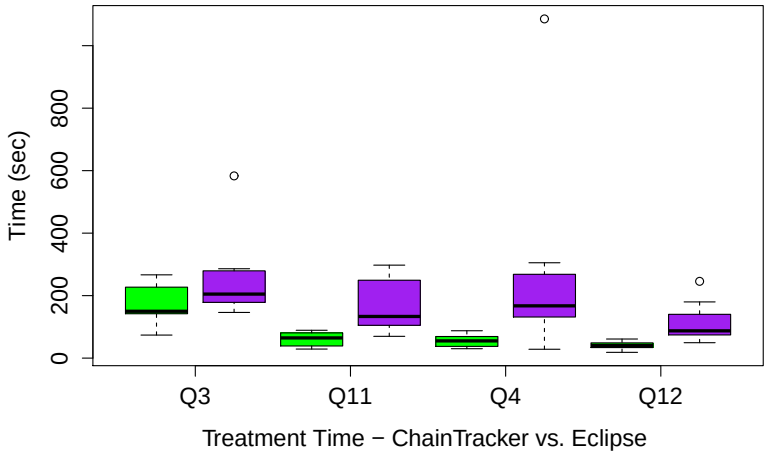
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	Score (p-value)	Time (p-value)
Q3	What metamodel elements are used in the creation of the GUI!Application element?	0.0675	0.1893
Q11	What metamodel elements are used in the creation of the GUI!Screen element?	**0.0310***	**0.0021***
Q4	What metamodel elements are created using the Mockup!TriggerSection element?	1.0000	**0.0139***
Q12	What metamodel elements are created using the Mockup!ScreenSection element?	**0.0006***	**0.0012***

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q3	174.85	66.98	256.37	140.37
Q11	59.71	25.02	167.12	84.90
Q4	54.85	22.01	280.75	334.90
Q12	40.28	14.52	111.62	66.18

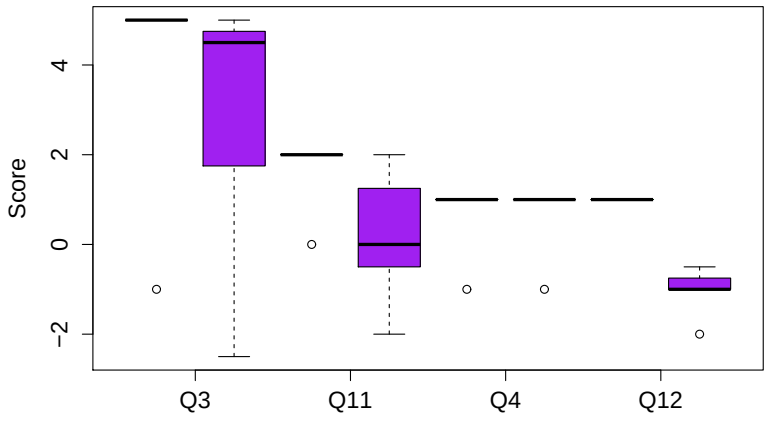
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
5	4.14	2.26	3.06	2.70
2	1.71	0.75	0.18	1.36
1	0.71	0.75	0.75	0.70
1	1.00	0.00	-1.00	0.46

Median Time Comparison	
CT	**EC**
149.00	204.00
64.00	132.50
55.00	167.00
40.00	86.50

Median Score Comparison	
CT	**EC**
5.00	4.50
2.00	0.00
1.00	1.00
1.00	-1.00



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

Table A.6: Session B (PhyDSL) - Results: Metamodel Dependencies in M2M (Element Level)

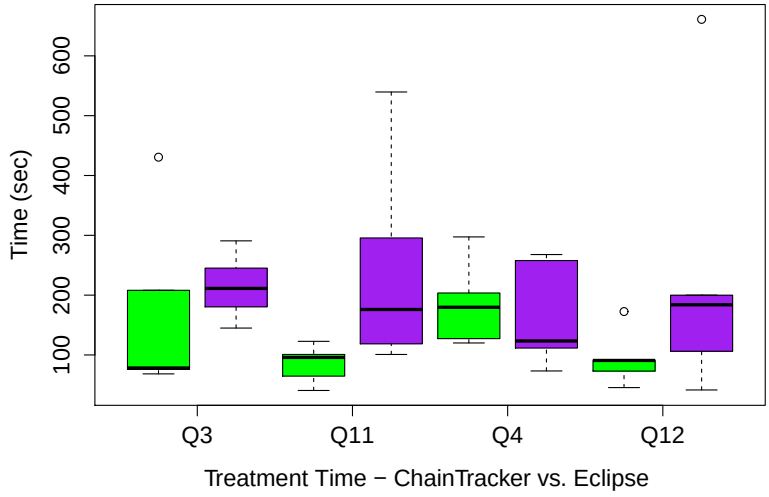
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	Score (p-value)	Time (p-value)
Q3	What metamodel elements are used in the creation of the Scoring!TouchRule element?	**0.0065***	0.3095
Q11	What metamodel elements are used in the creation of the Scoring!CollisionRule element?	**0.0310***	0.0555
Q4	What metamodel elements are created using the PhyDSL!Coordinate element?	**0.0072***	0.5476
Q12	What metamodel elements are created using the PhyDSL!Ends element?	**0.0065***	0.2222

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q3	171.80	155.66	214.00	56.57
Q11	84.20	32.23	245.60	180.86
Q4	185.20	71.57	166.20	89.44
Q12	94.20	47.30	237.80	244.36

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
3	3.00	0.00	0.90	0.82
3	3.00	0.00	0.40	0.89
3	1.80	1.64	-1.20	1.09
1	1.00	0.00	-0.40	0.82

Median Time Comparison	
CT	**EC**
78.00	211.00
95.00	176.00
179.00	123.00
90.00	183.00

Median Score Comparison	
CT	**EC**
3.00	1.50
3.00	0.50
3.00	-2.00
1.00	-1.00



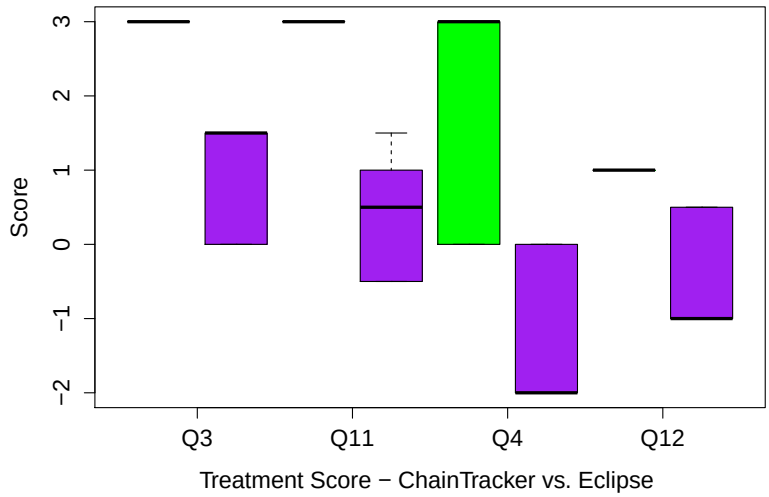


Table A.7: Session A (ScreenFlow) - Results: Metamodel Dependencies in M2M (Property Level)

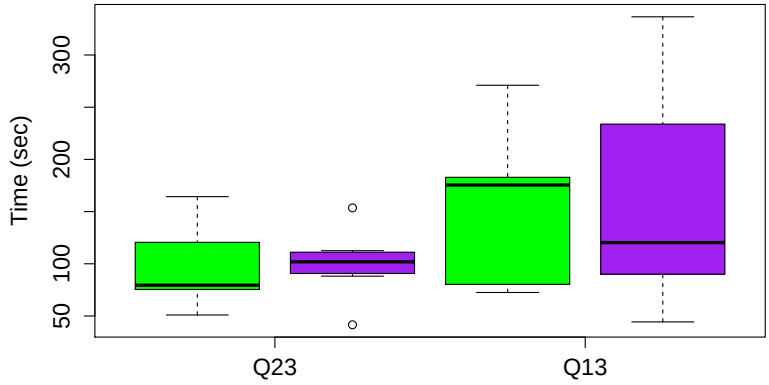
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q23	What metamodel elements are created using the property 'fromScreen' of the Mockup!Transition element?	**0.0046***	0.6943
Q13	What metamodel elements are created using the property 'name' of the Mockup!Screen element?	**0.0491***	0.9551

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q23	97.57	39.55	99.87	31.05
Q13	148.8	74.76	158.00	105.53

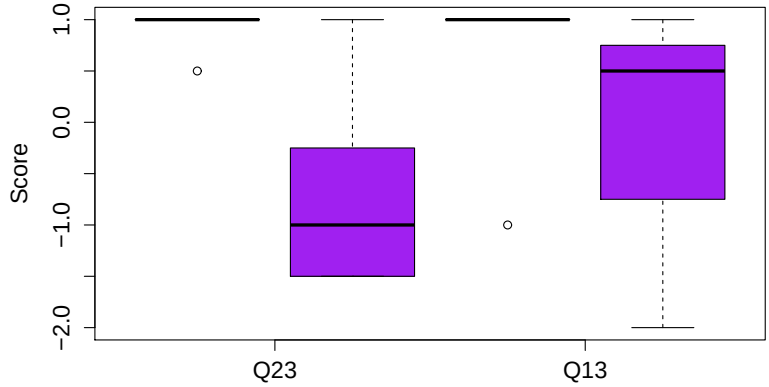
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	0.92	0.18	-0.75	0.96
1	0.71	0.75	0.00	1.13

Median Time Comparison	
CT	**EC**
79.00	101.5
175.0	119.5

Median Score Comparison	
CT	**EC**
1.00	-1.00
1.00	0.50



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

Table A.8: Session B (PhyDSL) - Results: Metamodel Dependencies in M2M (Property Level)

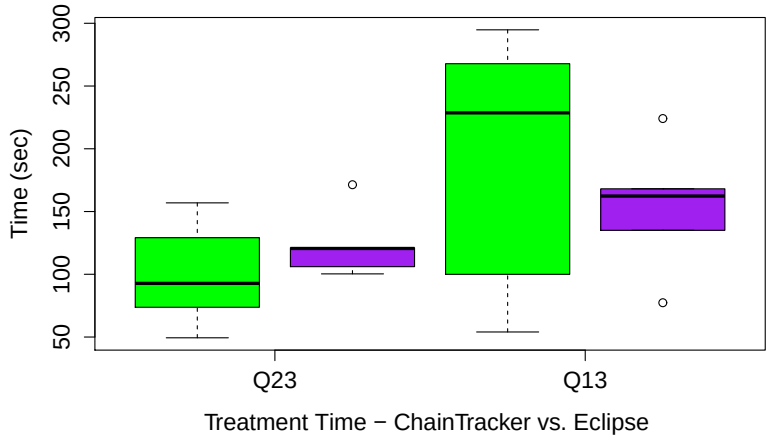
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q23	What metamodel elements are created using the property 'value' of the PhyDSL!BooleanType element?	**0.0412***	0.4206
Q13	What metamodel elements are created using the property 'name' of the PhyDSL!Actor element?	**0.0393***	0.5476

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q23	99.80	42.92	123.60	27.98
Q13	188.40	106.01	153.20	53.49

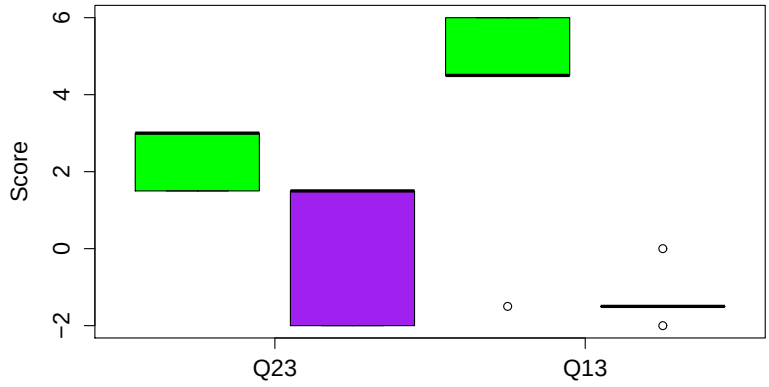
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
3	2.40	0.82	0.09	1.91
6	3.90	3.11	-1.30	0.75

Median Time Comparison	
CT	**EC**
92.00	120.00
228.00	**162.00**

Median Score Comparison	
CT	**EC**
3.00	1.50
4.50	-1.50



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

Table A.9: Session A (ScreenFlow) - Results: Metamodel Dependencies in M2T Transformations

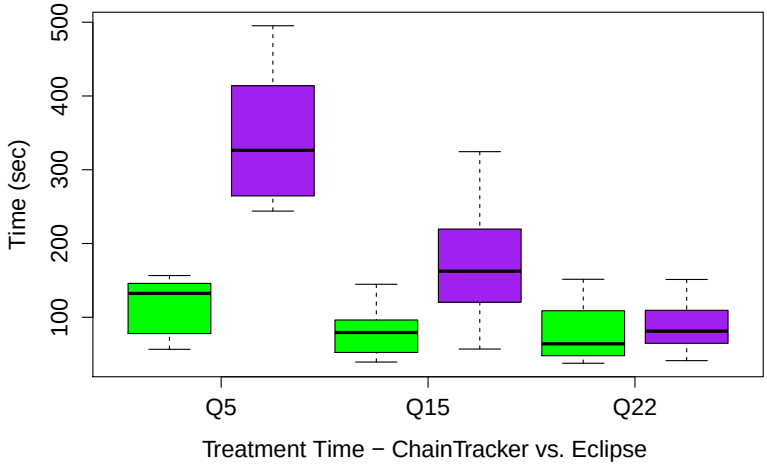
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q5	Considering the entire transformation chain, what metamodel elements does the template line 24 in generateControllers.mtl depend on?	**0.0119***	**0.0003***
Q15	Considering the entire transformation chain, what metamodel elements does the template line 19 in generateViews.mtl depend on?	0.2143	**0.0205***
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 41-44 in generateControllers.mtl depend on?	**0.0025***	0.6126

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q5	112.85	41.42	343.12	91.70
Q15	79.57	37.23	172.75	82.97
Q22	80.28	45.18	87.625	35.12

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.71	0.75	0.12	1.27
5	5.00	0.00	2.62	3.05
6	6.00	0.00	2.68	1.88

Median Time Comparison	
CT	**EC**
132.00	326.00
79.00	162.00
63.00	81.00

Median Score Comparison	
CT	**EC**
2.00	0.00
5.00	3.50
6.00	2.75



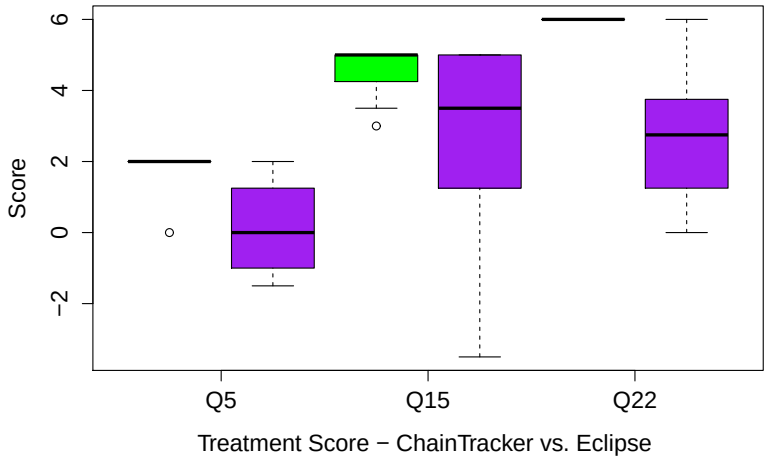


Table A.10: Session B (PhyDSL) - Results: Metamodel Dependencies in M2T Transformations

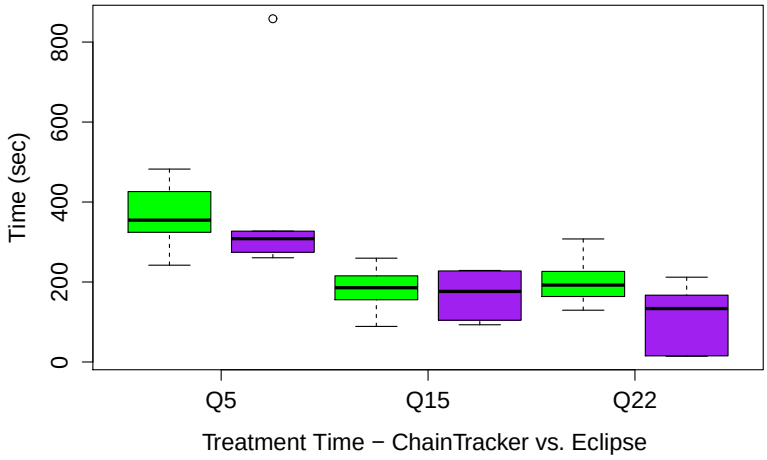
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q5	Considering the entire transformation chain, what metamodel elements does the template line 148 in generateDynamics.mtl depend on?	**0.0072***	0.6905
Q15	Considering the entire transformation chain, what metamodel elements does the template line 110 in generateScoring.mtl depend on?	**0.0097***	1.0000
Q22	Considering the entire transformation chain, what metamodel elements does the template lines 104-108 in generateControls.mtl depend on?	**0.0117***	0.2222

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q5	365.60	92.65	405.40	254.40
Q15	180.40	64.38	165.60	64.87
Q22	203.40	68.08	108.20	90.00

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
6	5.50	0.00	-1.00	1.96
5	4.90	0.22	-0.09	1.19
6	5.40	1.34	0.19	1.78

Median Time Comparison	
CT	**EC**
354.00	**308.00**
185.00	**176.00**
192.00	**133.00**

Median Score Comparison	
CT	**EC**
5.50	-1.50
5.00	0.00
6.00	0.00



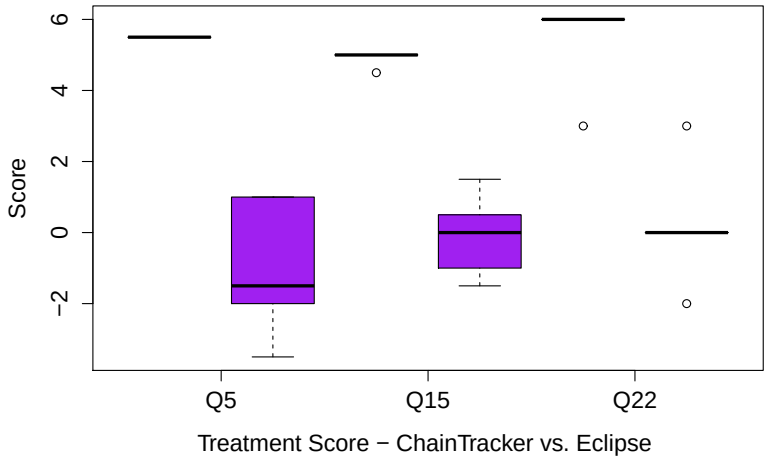


Table A.11: Session A (ScreenFlow) - Results: Generation Dependencies in M2T Transformations

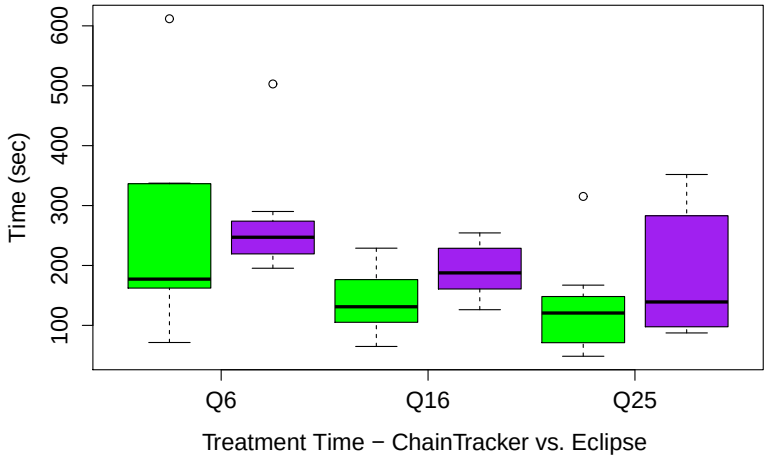
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q6	What template lines in generateControllers.mtl are used in the generation of line 21 in PlayerActivity.java?	0.5014	0.5358
Q16	What template lines in generateControllers.mtl are used in the generation of line 37 in LoginActivity.java?	0.0874	0.0938
Q25	What template lines in generateViews.mtl are used in the generation of line 27 in AndroidManifest.xml?	**0.0377***	0.3969

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q6	264.85	181.08	271.87	98.24
Q16	140.42	60.10	191.37	43.42
Q25	131.42	91.66	184.25	104.5

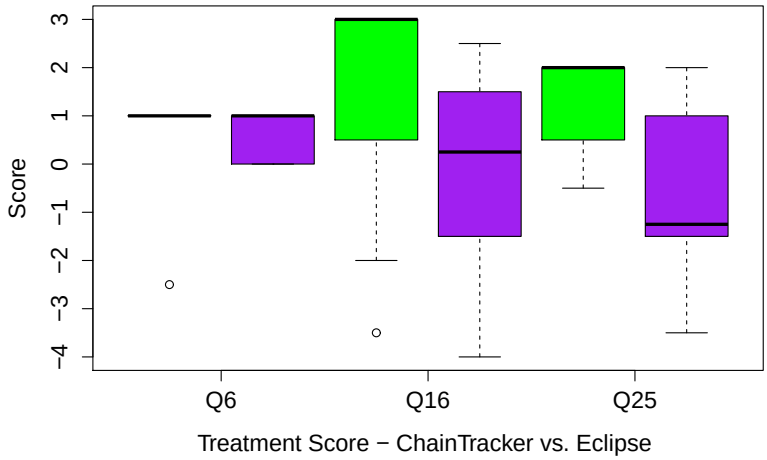
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
1	0.50	1.32	0.62	0.51
3	1.35	2.83	-0.12	2.19
2	1.21	1.03	-0.62	1.82

Median Time Comparison	
CT	**EC**
177.00	246.50
131.00	187.00
120.00	138.50

Median Score Comparison	
CT	**EC**
1.00	1.00
3.00	0.25
2.00	-1.25



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

Table A.12: Session B (PhyDSL) - Results: Generation Dependencies in M2T Transformations

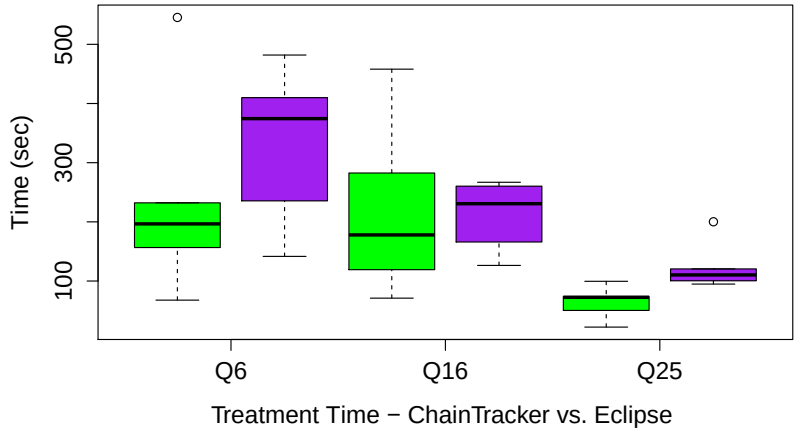
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q6	What template lines in generateScoring.mtl are used in the generation of line 102 in ScoringManager.java?	**0.0482***	0.4206
Q16	What template lines in generateLayout.mtl are used in the generation of line 264 in PhysicsView.java?	0.8288	1.0000
Q25	What template lines in generateControls.mtl are used in the generation of line 99 in ControlManager.java?	1.0000	**0.0158***

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q6	239.20	181.66	328.00	137.57
Q16	221.40	153.89	209.40	61.47
Q25	63.20	28.83	124.80	43.18

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	0.50	1.83	-2.60	1.47
4	2.20	1.95	1.59	2.85
2	1.60	0.65	1.50	0.70

Median Time Comparison	
CT	**EC**
196.00	374.00
177.00	230.00
72.00	110.00

Median Score Comparison	
CT	**EC**
0.50	-2.00
2.50	**3.00**
2.00	2.00



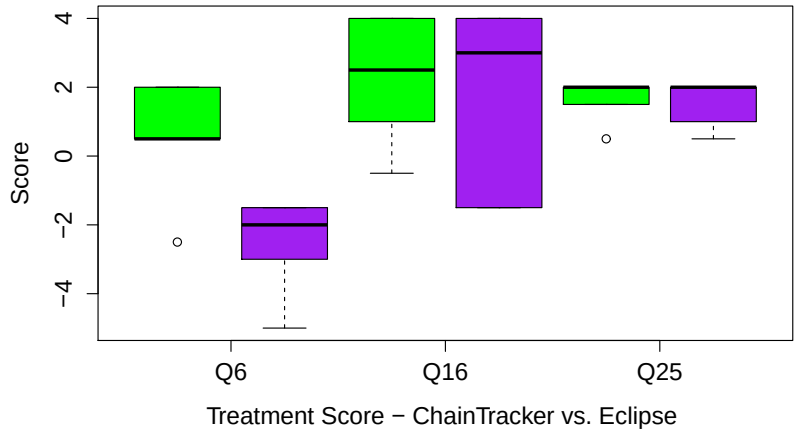


Table A.13: Session A (ScreenFlow) - Results: Generation Dependencies in MTCs (Element Level)

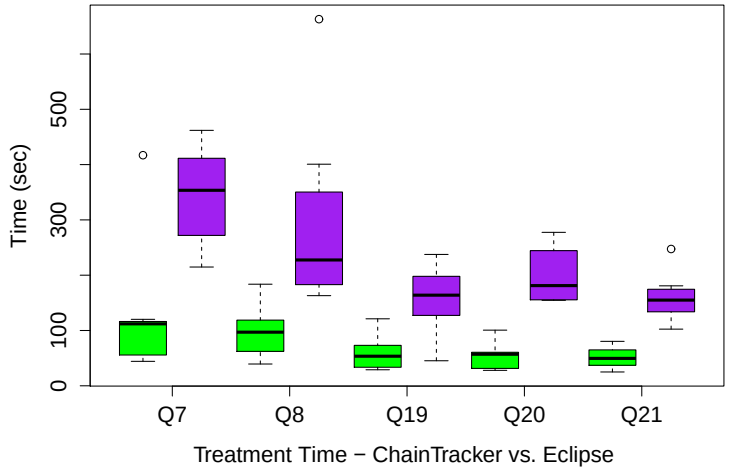
Results of Mann-Whitney Test (Two-Tailed) Time and Accuracy Score			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 4 in login.xml depend on?	**0.0032***	**0.0093***
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 34 in AndroidManifest.xml depend on?	**0.0073***	**0.0012***
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in AndroidManifest.xml depend on?	**0.0443***	**0.0037***
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 38 in LoginActivity.java depend on?	**0.0007***	**0.0003***
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 14 in login_activity.xml depend on?	**0.0007***	**0.0003***

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q7	130.57	129.99	343.25	85.39
Q8	96.85	50.61	293.12	168.04
Q19	59.00	3.23	157.12	60.58
Q20	52.28	25.75	198.75	51.61
Q21	50.85	22.11	159.12	43.22

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
2	1.71	0.75	-0.81	1.71
6	6.00	0.00	2.81	2.84
5	5.00	0.00	3.87	1.32
4	4.00	0.00	0.62	2.24
6	6.00	0.00	2.12	2.19

Median Time Comparison	
CT	**EC**
112.00	353.00
96.00	227.50
53.00	163.50
57.00	180.50
49.00	154.50

Median Score Comparison	
CT	**EC**
2.00	-0.50
6.00	3.00
5.00	4.25
4.00	1.00
6.00	2.75



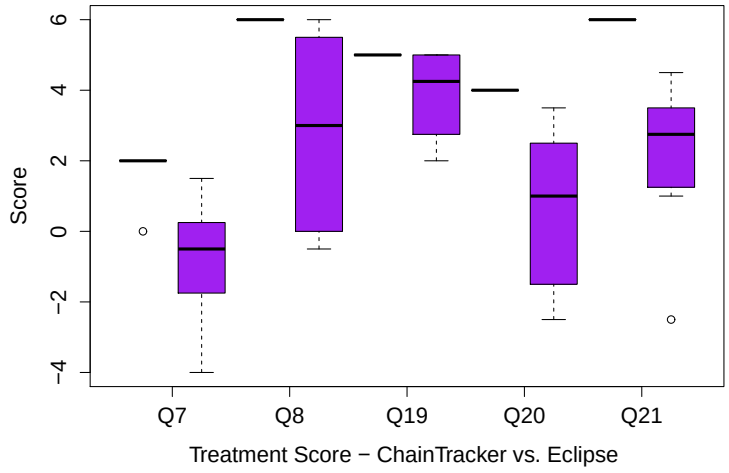


Table A.14: Session B (PhyDSL) - Results: Generation Dependencies in MTCs (Element Level)

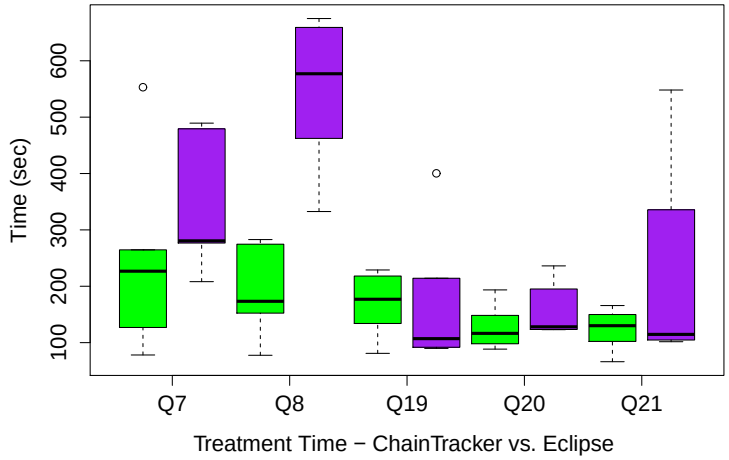
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q7	Considering the entire transformation chain, what metamodel elements does the generation of line 100 in ScoringManager.java depend on?	**0.0094***	0.3095
Q8	Considering the entire transformation chain, what metamodel elements does the generation of line 220 in DrawingHelper.java depend on?	**0.0072***	**0.0079***
Q19	Considering the entire transformation chain, what metamodel elements does the generation of line 141 in MainActivity.java depend on?	**0.0066***	0.8413
Q20	Considering the entire transformation chain, what metamodel elements does the generation of line 29 in ControlManager.java depend on?	**0.0055***	0.2222
Q21	Considering the entire transformation chain, what metamodel elements does the generation of line 281 in PhysicsView.java depend on?	**0.0055***	0.6905

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q7	249.4	185.43	346.40	128.87
Q8	191.6	86.61	540.80	143.90
Q19	167.0	61.49	180.20	133.31
Q20	128.4	42.80	161.00	51.86
Q21	122.4	39.29	240.40	198.48

Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
7	6.70	0.67	-1.00	2.00
5	5.00	0.00	1.00	1.27
8	8.00	0.00	0.20	1.95
7	7.00	0.00	0.10	1.34
5	5.00	0.00	-0.40	2.01

Median Time Comparison	
CT	**EC**
226.00	280.00
173.00	577.00
176.00	107.00
116.00	128.00
130.00	114.00

Median Score Comparison	
CT	**EC**
7.00	-1.50
5.00	1.50
8.00	-1.00
7.00	-0.50
5.00	0.50



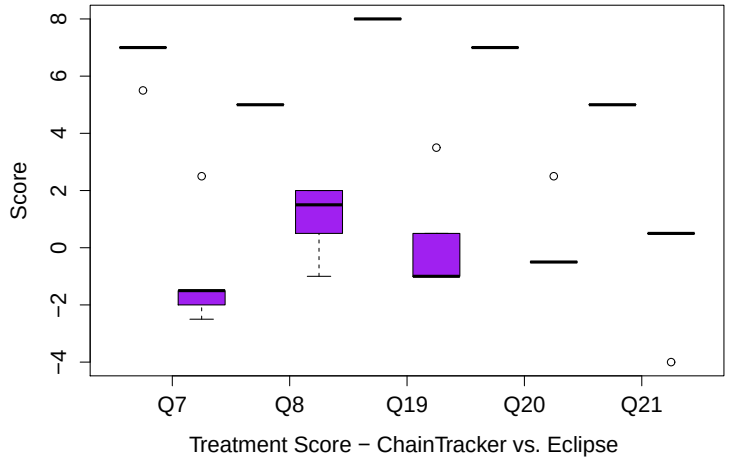


Table A.15: Session A (ScreenFlow) - Results: Generation Dependencies in MTCs (Property Level)

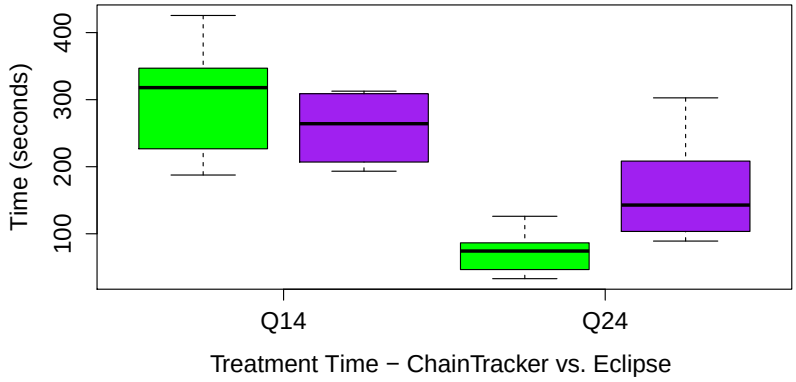
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 17 in login_activity.xml depend on?	**0.0090***	0.3969
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 8 in login_activity.xml depend on?	**0.04871***	**0.0059***

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q14	296.42	88.17	257.75	50.20
Q24	71.14	32.69	162.25	74.83

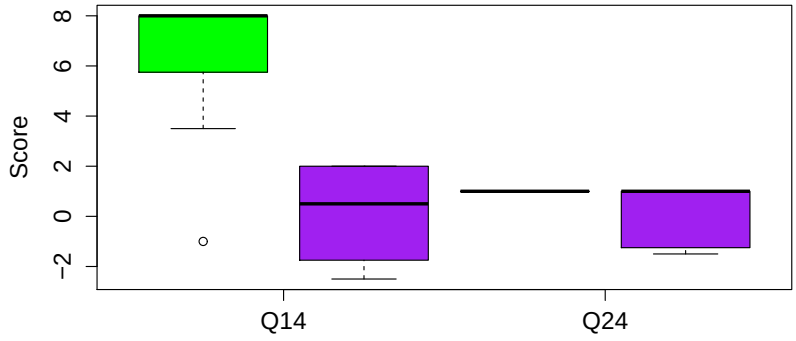
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
8	6.07	3.54	0.125	1.92
1	1.00	0.00	0.125	1.21

Median Time Comparison	
CT	**EC**
317.00	**263.50**
74.00	142.50

Median Score Comparison	
CT	**EC**
8.00	0.50
1.00	1.00



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

Table A.16: Session B (PhyDSL) - Results: Generation Dependencies in MTCs (Property Level)

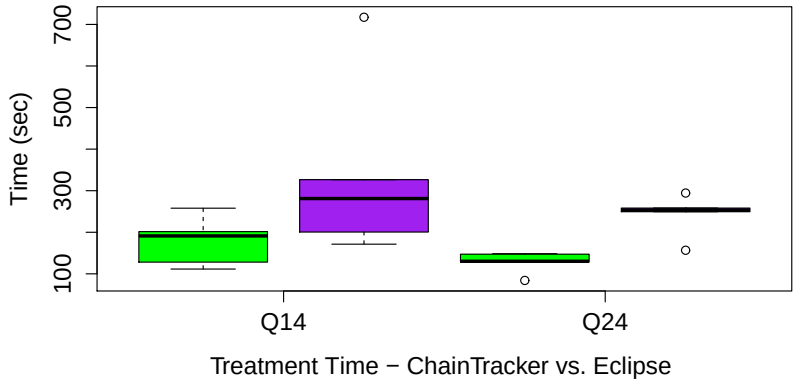
Results of Mann-Whitney Test (Two-Tailed) Time and Correctness			
N.	**Question**	**Score (p-value)**	**Time (p-value)**
Q14	Considering the entire transformation chain, what metamodel properties does the generation of line 76 in ScoringManager.java depend on?	**0.0066***	0.1508
Q24	Considering the entire transformation chain, what metamodel properties does the generation of line 18 in ControlManager.java depend on?	**0.0248***	**0.0079***

Treatments Time: Means and SD.				
N.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
Q14	177.60	59.00	339.00	220.20
Q24	127.40	25.97	242.00	51.29

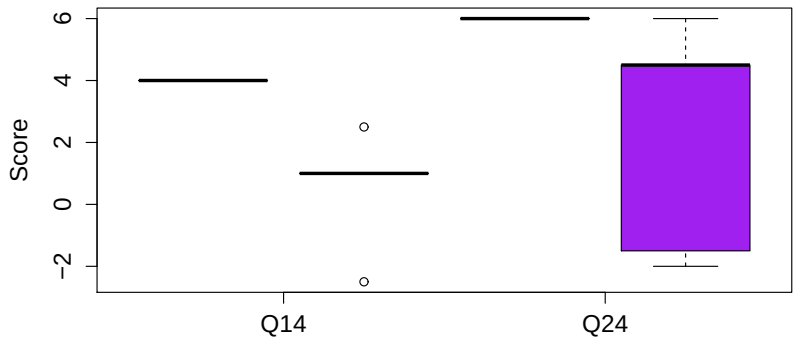
Treatments Score: Mean and SD.				
Max.	**CT Mean**	**CT SD**	**EC Mean**	**EC SD**
4	4.00	0.00	0.60	1.85
6	6.00	0.00	2.30	3.75

Median Time Comparison	
CT	**EC**
191.00	281.00
130.00	253.00

Median Score Comparison	
CT	**EC**
4.00	1.00
6.00	4.50



Treatment Time − ChainTracker vs. Eclipse



Treatment Score − ChainTracker vs. Eclipse

A.3 Library to Anonymous Index Traceability Links

Listing A.1 portrays a snippet with traceability links of the *Library to Anonymous Index* transformation chain. The complete file can be found at `http://hypatia.cs.ualberta.ca/~guana/ct/b2p-traces.json`.

```
{
    "metamodels": [
        {
          "name": "Book",
          " location ":   "./ Book.ecore"
          "elements ":  [
                {
                  "name": "Library ",
                  "  attributes  ":  [
                        {
                          "name": "books",
                          "type ":  "Book",
                          "id ":  "1"
                        }
                   ]
          }
        },
        {
          "name": " Publication ",
          " location ":   "./ Publication . ecore"
          "elements ":  [
                {
                  "name": "Database",
                  "  attributes  ":  [
                        {
                          "name": " publications ",
                          "type ":  " Publication ",
                          "id ":  "10"
                         },...
          ]
         },...
    ],
    " traces ":  [
        {
          "M2M": [
              {
                "id ":  "tlm1 ",
                " location  ":"./ Book2Publication. atl ",
                "type ":  " explicit  ",
                " line ":  "12",
                " targetID ":  "10",
                "sourceID":  "1"
              },   ....
          ],
          "M2T": [
              {
                "id ":  " tlt1  ",
                " location  ":"./ Publication2HTML.mtl",
                "type ":  " explicit  ",
                " line ":  "8",
                "sc ":  "1",
                "wc":  "39",
                "sourceID":  "10"
               },...
          ]
        }
    ]
}
```

Listing A.1: Library to Anonymous Index Traceability Links (JSON)