How Do Developers Solve Software-engineering Tasks on Model-based Code Generators? An Empirical Study Design

Victor Guana and Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
{guana, stroulia}@ualberta.ca

***Abstract*—Model-based code-generators are complex in nature; they are built using a variety of tools such as language workbenches, and model-to-model and model-to-text transformation languages. Due to the highly heterogeneous technology ecosystem in which code generators are built, understanding and maintaining their architecture pose numerous cognitive challenges to both novice and expert developers. Most of these challenges are associated with tasks that require to trace and pinpoint generation artifacts given a life-cycle requirement. We argue that such tasks can be classified in three general categories: (a) information discovery, (b) information summarization, and (c) information filtering and isolation. Furthermore, we hypothesize that visualizations that enable the interactive exploration of model-to-model and model-to-text transformation compositions can significantly improve developers' performance when reflecting on a code-generation architecture, and its corresponding execution mechanics. In this paper we describe an empirical study conceived (a) to understand the performance of developers (in terms of time and precision) when asked to discover, filter, and summarize information about a model-based code generator, using classic integrated development environments and editors, and (b) to measure and compare the developers' effectiveness on the same tasks using state-of-the-art traceability visualizations for model-transformation compositions.**

I. Introduction

Model-based code generation refers to a software-engineering methodology for building systems that systematically differ from each other [1][2]. In effect, code generators are frameworks for building applications from code semantics that have been engineered for reuse. However, code generators can be difficult to understand since they are typically composed of numerous elements, whose complex interdependencies pose cognitive challenges for developers performing design, implementation, and maintenance tasks [3][4].

Model-based code generators integrate rule-based *model-to-model* transformation languages (such as ATL [5] and EGL [6]) and template-based *model-to-text* transformation languages (such as Acceleo [7]) to translate high-level system specifications into executable code and scripts [8][9]. At the core of a model-based code generator, *model-to-model* and *model-to-text* transformations are composed in so-called model-transformation chains (MTCs) [10].

Given the complexity and heterogeneity of the technologies involved in a code generator, developers who are trying to inspect and understand the code-generation process have to deal with numerous different artifacts. As a concrete example, in a code-generator maintenance scenario, a developer might need to find all chained *model-to-model* and *model-to-text* transformation bindings, that originate a buggy line of code to fix it [11]. This task is error prone, if not virtually impossible, when done manually. We believe that flexible traceability tools are needed to collect and visualize information about the architecture and operational mechanics of code generators, to reduce the cognitive challenges that developers face during their life-cycle. With the purpose of tackling this challenge, we have developed ChainTracker [12][13], a tool that enables developers to better understand how model-based code generators are built, using interactive traceability visualizations and code projections. ChainTracker gathers and visualizes *model-to-model*, and *model-to-text* traceability information for ATL and Acceleo model-transformation compositions (Figure 1).

Whether at design or maintenance time, developers are constantly trying to solve software-engineering tasks on model-based code generators. We argue that such tasks can be classified in three categories: (a) information discovery, (b) information summarization, and (c) information filtering and isolation. In this paper we describe an empirical study conceived *(a) to understand the performance of developers when asked to discover, filter, and summarize information about a model-based code generator, using classic integrated development environments and editors*, and *(b) to measure and compare the performance of developers executing the same set of tasks using state-of-the-art visualizations for code-generator traceability information*.

This study will enable us to analyze the performance of developers when reflecting on a model-based code generator to achieve various software-engineering goals. Furthermore, it will increase our understanding of how advanced development environments and traceability visualization tools, such as ChainTracker, can help developers to design, study and maintain a code generator. This study includes a comparative analysis of developers' performance in answering questions

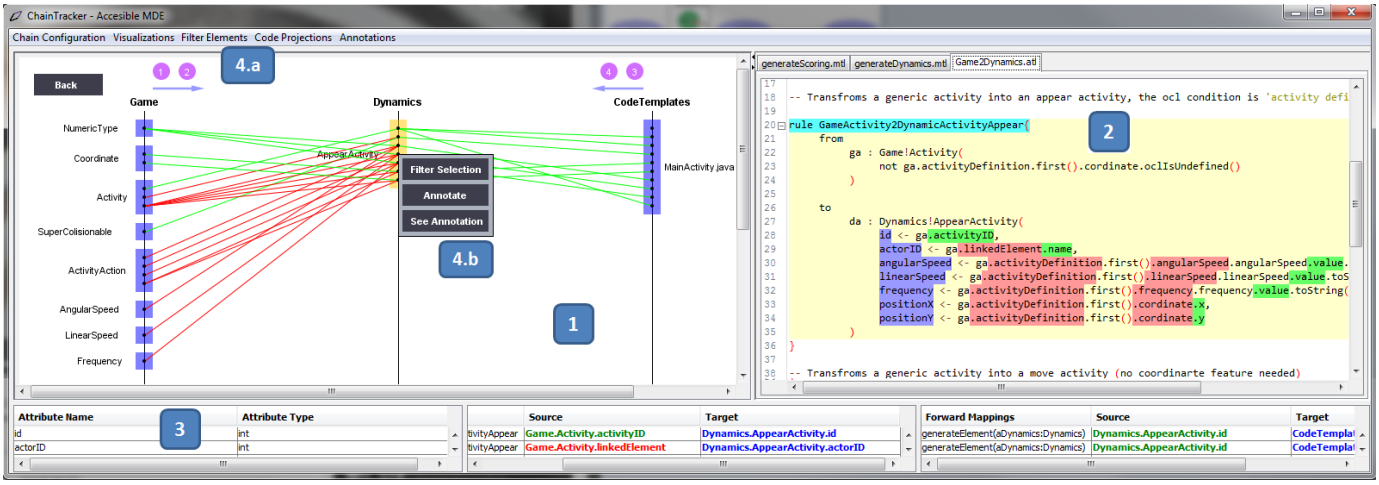

Fig. 1. ChainTracker's Main Screen (1) Transformation-composition Branch View; (2) Transformation Script View; (3) Binding Information Tables; (4.a) Filtering and Visualization Options; (4.b) Context-dependent menu to isolate artifacts related to metamodel elements, or generated textual sources.

with ChainTracker versus using existing code editors (i.e. Eclipse). Furthermore, in this study developers' performance is understood in terms of the time taken to answer a question and the correctness of their answers.

In this paper, we first present our study research questions and hypothesis. Second, we describe the study subject systems. Third, we present a detailed description of the families of tasks developers will solve in the study, including question templates that can be reused by the community. Finally we introduce the protocol of the study, its expected treats to validity, and expected contributions.

II. Research Questions and Hypothesis

In the life-cycle of a code generator developers ask multiple questions to optimize and maintain its infrastructure. Particularly, once a code-generator has been built, developers face multiple scenarios of evolution [14]. The two most important among them are *metamodel evolution*, in which changes are needed to the domain-specific language that interfaces with the end user, or to the intermediate metamodels that modularize the code-generation process, in order to improve the language expressiveness, and *platform evolution*, where the generated code needs to be refined with different purposes, such as fixing a bug or optimizing the performance of a generated codebase. In the latter scenario, the generator's *model-to-text* and, in some cases *model-to-model* transformations, need to be modified in order to reflect such refinements in a systematic way. Indeed, evolutionary scenarios in model-based code generators motivate questions about their architecture and execution mechanics, such as:

- *Where does this generated feature come from?*
- *What chained generation artifacts would be affected if a model element were removed or modified?*
- *What is the coverage of the transformation rules in each stage of my generation process?*
- *What portions of code have evolved in the generated codebases? and, assuming that code changes should indeed be included in future genertion instances, what elements of the underlying models and transformations should be revised?*

To answer these questions, developers need to have a thorough understating of the generation architecture. In this study we hypothesize that the interactive exploration of *model-to-model* and *model-to-text* transformation scripts can significantly improve developers' performance when reflecting on a code-generator architecture. Furthermore, we believe that the tasks that developers perform when reflecting on the design and execution mechanics of a generator can be classified in three general categories. Let us now briefly discuss each one of them.

1. Information Discovery Tasks: The developer's intent when performing this family of tasks is to explore the code generator to identify its major components, and to understand how the underlying transformation scripts are organized from a static point of view. This type of task involves locating individual elements of the code-generator's architecture, i.e., individual metamodel elements, transformation rules, and collections of transformation bindings, inside the generator's source scripts. These tasks are commonly performed when a developer is dealing with legacy code generators that need to be reused or optimized.

2. Information Summarization Tasks: The purpose of these tasks is for developers to measure generic information of the code-generation architecture, such as to quantify the coverage of a model transformation, or to measure the size of its metamodels. Summarizing information about the code generator allows developers to assess, and potentially improve, its overall design and correctness [15].

3. Information Filtering and Isolation Tasks: These tasks are generally performed when developers are assessing the impact of *platform-evolution* scenarios. They involve tracing and isolating elements of the code-generation architecture

from a dynamic perspective, in order to find dependency relationships between metamodel elements, metamodel attributes, transformation bindings, and generated pieces of code.

Considering the above types of tasks we believe developers solve when answering questions about a model-based code generator, we intent to investigate two research questions:

- *Q1: How do developers approach the process of answering questions that involve the discovery, filtering, and summarization of artifacts that constitute a code generator?*
- *Q2: Do developers answer questions more accurately when solving tasks that involve information discovery, filtering, and summarization using the interactive traceability visualizations provided by ChainTracker?*

On the basis of the above questions we outlined two corresponding null hypotheses.

- H_{Q1}: Developers spend an equal amount of time when solving tasks that involve information discovery, filtering, and summarization of a model-based code generator using ChainTracker as they do using Eclipse editors.
- H_{Q2}: Developers provide equally accurate answers, in terms of task solution correctness, using ChainTracker as they do using Eclipse editors.

III. Subject Systems

The subject systems of our study are two model-based code generators developed in our research laboratory: PhyDSL (System Subject 1) and ScreenFlow (System Subject 2). PhyDSL [16][17] is a model-based code generator for mobile physics-based 2D games (see Figure 2). It is built in a textual domain-specific language, and a multi-branched model-transformation composition that includes three *model-to-model* transformations implemented using ATL, and three template-based *model-to-text* transformations written in Acceleo. PhyDSL is currently used to create cost-effective and fully-featured mobile games with rehabilitation purposes. PhyDSL is now being used in the construction of mobile games used by the Faculty of Rehabilitation Medicine at the University of Alberta, the Glenrose Rehabilitation Hospital in Edmonton, Canada, and the Knowledge Media Design Institute at the University of Toronto. ScrenFlow[1] is a code generator for Android application skeletons with interface-navigation logic, from graphic user interface storyboard specifications (see Figure 3). ScreenFlow is composed by a textual domain-specific language, and a single-branched (i.e. linear) model-transformation composition that includes one *model-to-model* transformation, and one *model-to-text* transformation, written in ATL and Acceleo respectively. ScreenFlow is used by novice Android application developers in rapid software prototyping environments such as hackathons.

[1]a complete description and demo video of ScreenFlow can be found at http://goo.gl/IGqLTv

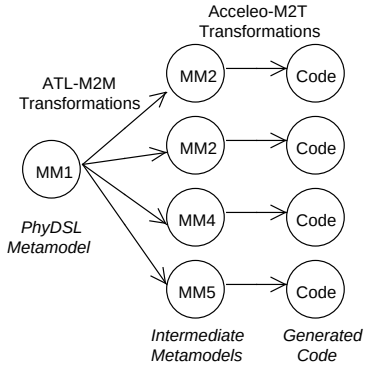

Fig. 2. PhyDSL's multi-branched model-transformation composition architecture: four model-to-model (M2M) ATL transformations, and four model-to-text (M2T) Acceleo transformations chained in four transformation branches.

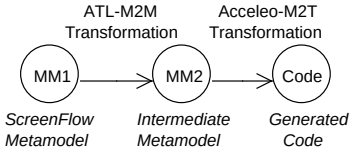

Fig. 3. ScreenFlow's linear model-transformation composition architecture: one model-to-model (M2M) ATL transformation, and one model-to-text (M2T) Acceleo transformation, chained in one transformation branch.

ATL *model-to-model* transformations and Acceleo *model-to-text* transformations are two widely adopted model-transformation technologies in both industry and academic environments. We believe that ATL and Acceleo exemplify the semantic complexity of state-of-the-art transformation languages built on top of model manipulation languages such as OCL [18], thus generalizing the complexity behind modern model-based code generators.

IV. Dependent and Independent Variables

Considering the hypotheses H_{Q1} and H_{Q2}, we have two dependent variables in our study:

- Var_A: Time developers spend solving each task.
- Var_B: Developers' accuracy in terms of task solution correctness.

TABLE I
Study Independent Variables

Subject System	Tasks ChainTracker	Tasks Eclipse
Subject System 1	$VarCT_1$	$VarEE_1$
Subject System 2	$VarCT_2$	$VarEE_2$

The four independent variables of the study are $VarCT_1$, $VarEE_1$, $VarCT_2$, and $VarEE_2$ (see Table I). The first two define the set of questions developers will solve using PhyDSL as the subject system (a model-based code generator with a branched transformation composition). The last two

specify the set of questions to be solved using ScreenFlow as a subject system (a model-based code generator with a linear transformation composition). We believe that by using two subject systems with different levels of complexity (eight transformations in PhyDSL vs. two of ScreenFlow) the study will be able to investigate if the compositional architecture of the generator affects developers when conducting software-engineering tasks.

V. Detailed Hypothesis

Taking into account our two high-level null hypotheses, our two subject systems, and the variables of our study, let us briefly discuss the set of detailed null hypotheses that this study will try to reject. They all share the following general form:

$$H_x Var_{xy} : \tilde{V}ar_x CT_y = \tilde{V}ar_x EE_y$$

- where x is A or B in place for hypothesis H_{A0} and H_{B0} related to Var_A-time and Var_B-accuracy respectively;
- $\tilde{V}ar_x CT_y$ is the median of our study dependent variables, where $_x$ indicates the developer's time and precision, when solving tasks using the interactive visualizations provided by ChainTracker;
- $\tilde{V}ar_x EE_y$ is the median of our study dependent variables, where $_x$ indicates the developer's time and precision when solving tasks using off-the-shelf Eclipse script editors for ATL and Acceleo;
- and $_y$ (1, or 2) refers to the result of a dependent variable obtained from developers solving tasks on the *System Subject 1: PhyDSL,* and the *System Subject 2: ScreenFlow,* respectively.

In summary, four detailed null-hypothesis will be investigated in this study. While $H_A Var_{A1}$ and $H_A Var_{A2}$ compare the median time spent by developers solving tasks that involve information discovery, filtering, and summarization on single and multi-branched model-based code generators *(i.e. developers spend an equal amount of time solving questions using ChainTracker as they do using Eclipse editors for single and multi-branched code generators)*, hypothesis $H_B Var_{A1}$ and $H_B Var_{B2}$ compare the median accuracy (in terms of task solution correctness) of developers conducting software-engineering tasks on single and multi-branched model-based code generators, respectively *(i.e. developers provide equally accurate answers using ChainTracker as they do using Eclipse editors for single and multi-branched code generators)*.

VI. Study Protocol

The protocol of the study will be divided in two main stages that involve two independent working sessions with two different sets of participants, and two exit surveys that will assess the participants' experience during the study (see Figure 4).

Stage 1: The first stage involves a working session with 15 developers. Each participant will be asked 30 questions about a subject model-based code generator. In this stage, information about the time spent by developers answering each question will be collected using an in-house survey application. In this first stage, developers will solve the first half of the tasks using off-the-shelf ATL and Acceleo code editors in Eclipse, and the second half using ChainTracker.

Stage 2: The second stage consists of a second working session with a new group of 15 developers. They will be asked to answer the same set questions as developers in Stage 1. Developers' performance will also be collected using our in-house survey application. In this second stage developers will be instructed to solve the first half of the tasks using ChainTracker, and the second half using code editors in Eclipse.

At the end of each working session, developers will be asked to complete a survey on the usability of ChainTracker and their general experience during the session. Let us now briefly discuss how the working sessions will be structured.

A. Working Sessions

During Stages 1 and 2, each developer will be assigned an individual working station consisting of a desktop computer in which Eclipse and ChainTracker will be installed and deployed. This computer will also have an in-house system capable of monitoring the participant's activity such as mouse clicks and keystrokes events. Indeed, the difference between the working sessions of Stage 1 and 2 is the order of the tools that developers will use to solve the given tasks. Both stages will be divided in four parts.

Part 1. The participants will receive a 20 minute high-level presentation of Eclipse, ChainTracker, and the purpose of the study. A demonstration of Eclipse's features and user interface will be given through a typical scenario of information discovery, filtering, and summarization on both of the subject systems. A similar demonstration will be conducted using ChainTracker's interactive visualizations, and code-projection features. Finally, participants will be asked to sign the informed consent form of the study.

Part 2. Participants will be pointed to our in-house survey application where they will answer questions about their experience with modeling tools, and their overall software-development expertise. More specifically, the questionnaire will cover i) the developers' number of years of software-development experience; ii) their experience using integrated development environments; iii) their experience using modeling tools to document software system implementations; and iv) whether they been exposed to model-transformation technologies before. A predefined list of options including popular development environments, modeling tools, and model-transformation technologies will be presented to the participants along with open fields that will receive alternative answers.

Part 3. Participants will have a five-minute break.

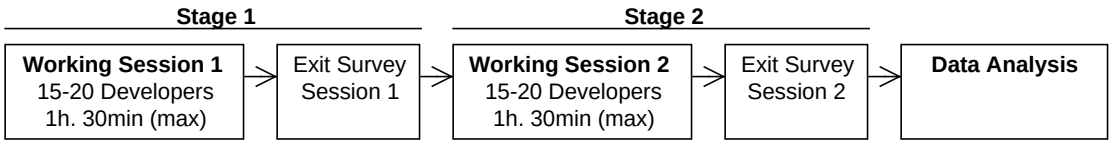

Fig. 4. Protocol Stages

Part 4. Participants will be pointed to the second part of the survey that will ask them to solve tasks with Eclipse and ChainTracker. The mechanics of this part are as follows:

1) Each participant will be presented with a question about a subject system (see Section VI-B).
2) The participant will use ChainTracker or Eclipse to find information relevant to the question, and answer the question.
3) The participant will submit her/his answer.
4) The participant will be directed to the next question. A total of 30 questions will be asked to every participant covering each one of our proposed families of tasks; 15 questions will be related to the Subject System 1, and 15 to the Subject System 2.

B. Question Templates

An appendix containing a collection of template questions that involve information discovery, filtering, and summarization tasks on model-based code generators can be found at: http://goo.gl/BDXpUO. Some examples of our template questions are shown below.

- Find a template's upstream *model-to-model* transformation dependencies: What transformation rules are upstream related to the template line of code *[line-id]* in the *[template-script-name]* script?
- Identify the transformation rule that contains a given metamodel binding: What transformation rule contains the *[metamodel-binding]* binding in *[model-to-model-transformation-name]*?
- Evaluate how well a metamodel is used in a transformation composition: What percentage of the *[metamodel-name]* metamodel is been covered by the transformation composition?

C. Data Analysis

Due to the nature of the variables and the limited number of data points we will apply a Mann-Whitney "U" non-parametric statistical test to study the hypothesis propositions. We will adopt an alpha level with a p-value lower than 5%, thus we will consider an acceptable probability of 0.05 for Type-I error, i.e. rejecting the null hypothesis when it is true.

VII. Participants

The study solicits participants of any age and gender with at least three years of programming experience. The experience requirement is tightly related to the technical tasks that developers will perform during the duration of the study. Due to the limited experience of potential participants with model-driven engineering technologies, the study will not enforce model-driven engineering experience as a fundamental requirement, however candidates with experience on model-transformation technologies will be preferred. We believe that graduate and undergraduate students are potentially interested in acquiring different skills through the use of experimental tools that enhance their software-development abilities. Therefore the study solicits participants enrolled in advanced software engineering courses in academic institutions. This study will be also advertised in venues such as the International Conference on Model Driven Engineering Languages and Systems (MODELS) and the International Conference on Model Transformation (ICMT) to potentially conduct additional virtual working sessions with highly skilled professionals on model-driven engineering technologies.

VIII. Potential Threats to Validity

Construct validity *(Do we measure what is intended?)* In this study we will measure the performance of developers answering questions about a model-based code-generating system. We understand developers' performance in terms of the time they take answering each question and their correctness. We have developed an in-house survey application that presents participants with the questions and measures the time from when the question is showed to the participant to the moment when the participant has submitted an answer. Furthermore, we have carefully instantiated our question templates on our subject systems, and the correctness of each expected answer has been validated by three model-transformation experts. We do not foresee any significant threats to the construct validity of study.

Internal validity *(Are there unknown factors which might affect the outcome of the experiments?)* We have identified two main threats to the internal validity of this study. First, the limited number of participants and their heterogeneous expertise on model-driven development technologies may limit the validity of the study. This study, however, is planned to be conducted with a minimum of 30 developers with at least three years of software-development experience. Considering that model-driven engineering technologies (such as model-transformation languages and modeling tools) are still in their infancy, and are yet to be adopted by the software engineering community at large, our pool of participants are representative of a community in which the majority of developers designing and maintaining code generators are novice, or at least not highly experienced, on model-driven engineering tools.

Furthermore, this study hopes to capture the interest of the model-driven engineering community and conduct additional virtual working sessions with highly-skilled model-driven professionals around the world. Indeed, having a diverse pool of participants will be highly valuable to the generalizability and statistical soundness of the study. Second, we are aware of the learning curve of ChainTracker and how its accessibility might affect developers when trying to answer questions on the subject systems. In order to minimize the impact of this threat to validity, we have included an introductory tutorial at the beginning of our working sessions' protocol (Section VI-A). The tutorial will showcase different question-solving scenarios using ChainTracker and Eclipse. Furthermore, during the last year we have iterated over ChainTracker's graphic user interface, running informal focus groups in order to make its features accessible and intuitive for developers.

External validity *(To what extend is it possible to generalize the findings?)* The subject systems of our study are two model-based code generators implemented using ATL, a rule-based *model-to-model* transformation language, and Acceleo, a template-based *model-to-text* transformation technology. Therefore any conclusions drawn from this study can not be fully generalized to the performance of developers solving software engineering tasks on model-based code generators built using other model-transformation technologies. However, both Acceleo and ATL are widely used in academia and industry, and more importantly, both languages are aligned to the *Query/View/Transformation (QVT)* standard for *model-to-model* transformations [19], and the *Model to Text Transformation Language (MOF)* standard for *model-to-text* transformations [20] proposed by the Object Management Group (OMG), respectively. Therefore the observations of this study can potentially be generalized to developers performing the same set of tasks in generators, with similar size and architecture, built using languages that comply with the same set of standards.

IX. Expected Contributions

The contributions of this study are twofold. First our study will be the first of its kind to investigate how developers approach the process of answering questions that reflect on the design and execution mechanics of model-based code generators. It is our strong belief that by gaining insight on the human aspects of model-driven software development, the community will be able to propose tools that make the construction of code generators less error prone and less cognitively challenging, thus potentially increasing the adoption of model-driven engineering techniques as a whole. Second, our study will increase the understanding on how developers can solve software-engineering tasks on model-based code generators, more accurately and efficiently, using interactive traceability collection and visualization tools such as ChainTracker. This study will gather information necessary to enhance the current features of ChainTracker, and to create new ones that further support developers in their daily tasks. Furthermore, we believe this study is a novel contribution to the community and plays an important role in the collective endeavour to improve and boost the adoption of model-driven engineering among software engineering researchers and practitioners.

Acknowledgements

This work was supported by The Killam Trust, NSERC (the Discovery and the IRC program), the GRAND NCE and IBM Canada.

References

[1] U. Aßmann, J. Knoop, and W. Zimmermann, "Model-based code-generators and compilers-track introduction," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer, 2014, pp. 386–390.

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

[3] 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.

[4] 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.

[5] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[6] D. Kolovos, R. Paige, and F. Polack, "The epsilon transformation language," *Theory and Practice of Model Transformations*, pp. 46–60, 2008.

[7] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," 2006.

[8] A. Bragança and R. J. Machado, "Transformation patterns for multi-staged model driven software development," in *Software Product Line Conference, 2008. SPLC'08. 12th International*. IEEE, 2008, pp. 329–338.

[9] 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. Citeseer, 2003, pp. 1–17.

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

[11] V. Guana and E. Stroulia, "Backward propagation of code refinements on transformational code generation environments," in *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on*, 2013, pp. 55–60.

[12] 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.

[13] V. Guana, K. Gaboriau, and E. Stroulia, "Chaintracker: Towards a comprehensive tool for building code-generation environments," in *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Press, 2014.

[14] 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.

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

[16] 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.

[17] V. Guana, E. Stroulia, and V. Nguyen, "Building a game engine: A tale of modern model-driven engineering."

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

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

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