Reflecting on Model-based Code Generators Using Traceability Information

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

***Abstract*—Model-based code generators use model-to-model and model-to-text transformations to systematize the construction of software systems. They integrate rule-based and template-based transformation languages to translate high-level specifications into executable code and scripts. Given the complexity and heterogeneity of the underlying transformation languages, flexible traceability tools are needed to collect and visualize information about their architecture and operational mechanics. In this paper, we present ChainTracker, a traceability analysis environment for model-based code generators. ChainTracker helps developers to reflect on the composition of model transformations during the different stages of their construction and maintenance. Furthermore, we describe a family of software-engineering tasks that developers have to complete during the construction of model-based code generators, and how ChainTracker makes the execution of these tasks less error prone and less cognitively challenging.**

I. Introduction

Model-based code generators use models as a vehicle to capture a system's specification. Domain-specific languages with high-level semantics enable developers to create models using well-formed constructs [1]. In turn, these models are input into model-transformation compositions to derive executable and/or deployment artifacts, such as source-code text and configuration scripts [2]. By raising the level of abstraction for the software specification and making extensive reuse of executable assets, model-based code-generators make the construction of software systems less error prone, less expensive, and, potentially, less challenging to non-computer experts [3].

The role of model-based code generators is very similar to that of compilers, translating a general-purpose programming language to machine code. In this sense, a code generator for a domain-specific language uses model transformations that inject execution semantics to the high-level models, and produces code based on templates that have been previously engineered for reuse. Domain-specific languages are typically associated with graphical or textual notations; using these notations, developers can express system specifications in terms with application-domain semantics. Such definitions are ruled by metamodels that constrain how the concepts relate to each other. In the transformation process, *model-to-model* transformation languages 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]. Lastly, *model-to-text* transformations take this intermediate models, and produce code in a general-purpose language, such as Java or C++, that can be compiled and executed [5].

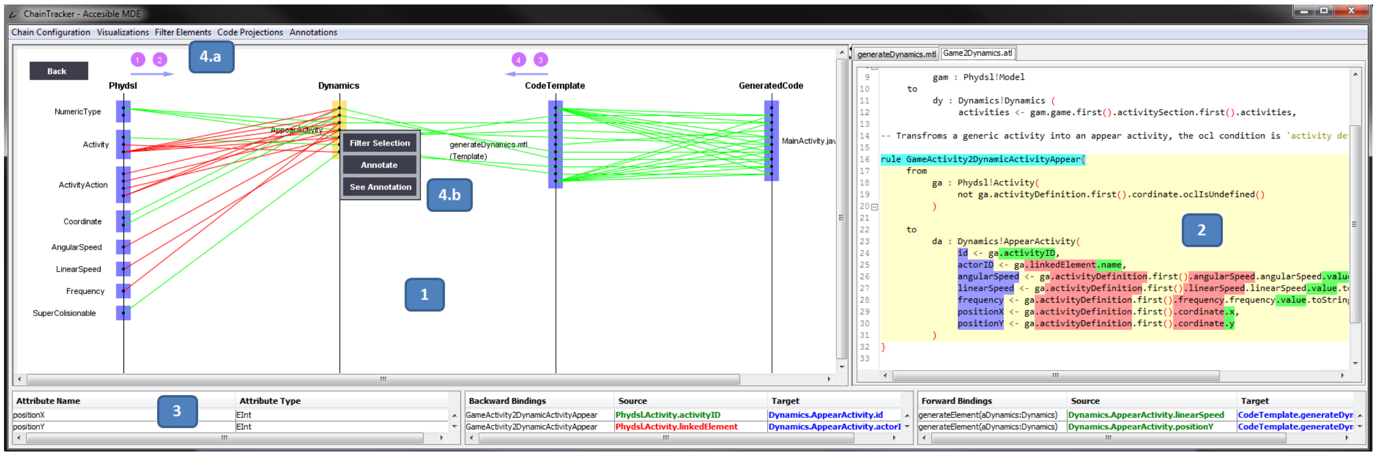
Numerous challenges remain open to make the construction and maintenance of model-based code-generators less tedious and less brittle to evolution [6] [7]. Despite of the multiple research addressing the technical aspects of building code-generators, relative few researchers investigate the core software-engineering problems behind their complex technology ecosystems.

In this paper, we reflect on two challenges that affect developers when building and maintaining code generators, namely, (a) the platform evolution of code generators, and (b) the overwhelming complexity of code-generation architectures. The former refers to the difficulties that developers face when the to-be-generated systems must fulfill new requirements and the generation artifacts must also synchronously evolve to reflect those changes. The latter stems from the fact that non-trivial model transformations are hard to understand, given that they heavily rely on textual scripts with cryptic operational semantics, and are composed in complex transformation chains that hard to mentally visualize and study.

The contribution of this paper is twofold. We first present *ChainTracker*, a visualization and trace-analysis environment that uses traceability information about model-transformation compositions, to support developers maintaining the complex operational mechanics behind model-based code generators. Second, we discuss how *ChainTracker* supports a number of software-engineering tasks involved in the construction of model-based code generators, ranging from tracing tangled code-generation artifacts, to summarizing information that assesses the quality of the generation process.

II. Related Work

During the life-cycle of a model-based code-generator, the generated code is often manually improved [7] [8]. The goals of these code modifications are bug fixing, code-performance optimization, and functionality extensions [9][10]. This process is known as platform evolution in model-driven engineering [8]. Due to the high complexity of the model-transformation compositions that live at the core of the code generators, and the complexity of the generated code itself,



```
        gam : Phydsl!Model
    to
        dy : Dynamics!Dynamics (
            activities <- gam.game.first().activitySection.first().activities,

-- Transfroms a generic activity into an appear activity, the ocl condition is 'activity de

rule GameActivity2DynamicActivityAppear{
    from
        ga : Phydsl!Activity(
            not ga.activityDefinition.first().cordinate.oclIsUndefined()
        )

    to
        da : Dynamics!AppearActivity(
            id <- ga.activityID,
            actorID <- ga.linkedElement.name,
            angularSpeed <- ga.activityDefinition.first().angularSpeed.angularSpeed.valu
            linearSpeed <- ga.activityDefinition.first().linearSpeed.linearSpeed.value.t
            frequency <- ga.activityDefinition.first().frequency.frequency.value.toStrin
            positionX <- ga.activityDefinition.first().cordinate.x,
            positionY <- ga.activityDefinition.first().cordinate.y
        )
}
```

Attribute Name	Attribute Type
positionX	EInt
positionY	EInt

Backward Bindings	Source	Target
GameActivity2DynamicActivityAppear	Phydsl.Activity.activityID	Dynamics.AppearActivity.id
GameActivity2DynamicActivityAppear	Phydsl.Activity.linkedElement	Dynamics.AppearActivity.actorI

Forward Bindings	Source	Target
generateElement(aDynamics:Dynamics)	Dynamics.AppearActivity.linearSpeed	CodeTemplate.generateDyn
generateElement(aDynamics:Dynamics)	Dynamics.AppearActivity.positionY	CodeTemplate.generateDyn

Fig. 1. ChainTracker - Main Screen (1) Transformation-composition Branch View; (2) Transformation Script Viewers; (3) Transformation In/Out Bindings Information Tables; (4.a) Filter and Transformation-script Visualization Options; (4.b) Context-aware menu to isolate artifacts related to metamodel elements or generated textual artifacts.

backwardly propagating code refinements to the generation architecture is a challenging task [11]. Some of the questions that developers have to answer when facing platform evolution include, *what portions of code have been changed in the generated codebases?* and, *assuming that a code change should indeed be propagated, what elements of the underlying models and transformations should be revised?*.

Most state-of-the-art tools that support platform evolution in model-based code-generators tackle the problem using traceability information. Van Amstel et al. [12][13] have used traceability information of *model-to-model* transformations to visualize their operational mechanics. Other researchers [14] [15] [16] have used traceability information to assess the quality of model-transformation compositions. More recently, in [17], Santiago et al. introduced iTrace, a tool that uses a *transformation-enrichment* strategy, similar to Jouault's proposal in [18], to collect traceability models at runtime. Nevertheless, to the best of our knowledge, current proposals fail to provide comprehensive tools that support end-to-end platform-evolution analysis, including analysis of the modified generated code and collection of traceability information from both *model-to-model* and *model-to-text* transformations. Furthermore, current proposals fail to provide effective tools that present the traceability information in an interactive and integrated tool, effectively supporting developers in the exploration of generation architectures.

In summary, most of the existing proposals support very specific maintenance and construction scenarios. Furthermore, they often provide theoretical frameworks with demonstrative examples, rather than generic tools than can be shipped and used by developers in fully-featured model-based code generators.

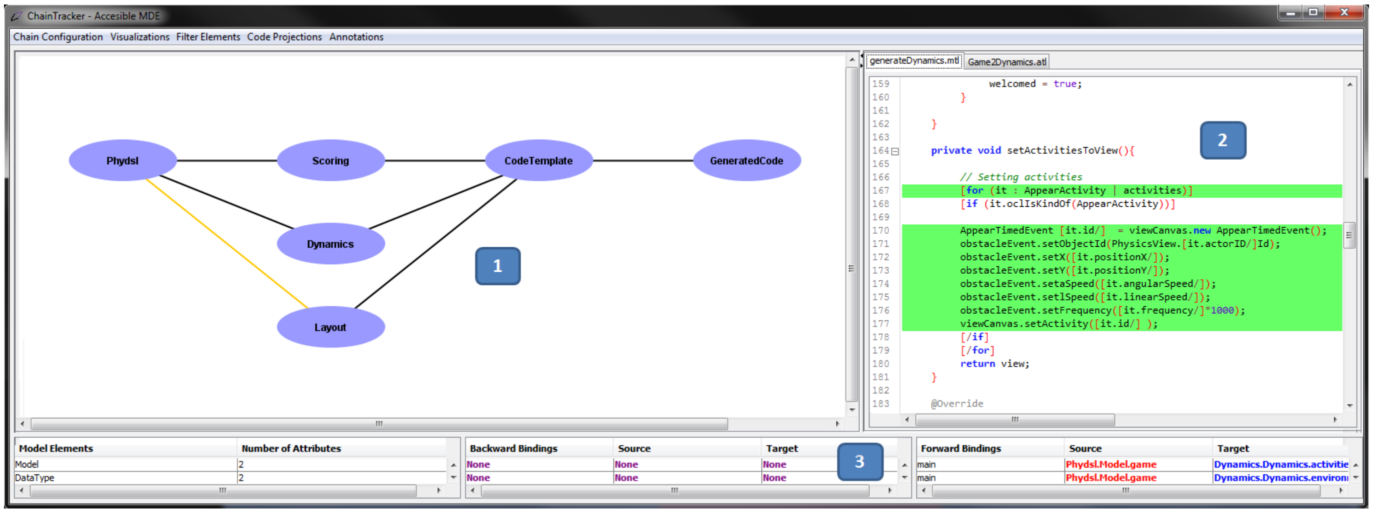
III. ChainTracker

Model-based code-generators are complex; they are built using multiple technology platforms including language workbenches that allow the definition of domain-specific languages [19], rule-based transformation languages that enable the specification of *model-to-model* transformations (such as in ATL [20]), and template-based *model-to-text* transformation technologies (such as Acceleo [21]) that allow the derivation of source-code files [5].

ChainTracker is an integrated analysis environment designed to support developers of model-based code generators by making model-driven engineering technologies more efficient, less error prone, and less cognitively challenging. *ChainTracker* provides an integrated analysis environment to visualize models and heterogeneous *model-to-model* and *model-to-text* transformation compositions, through interactive canvases and source-code viewers. Furthermore, *ChainTracker* collects and summarizes traceability information about the scripts of a code generator, and helps developers to identify maintenance hotspots that answer to platform-evolution scenarios. In [22], we presented a general conceptual framework, generalizable to all rule-based and template-based model-transformation compositions, that formalizes how to collect and model traceability information in model-based code generators. In [23], we showcased *ChainTracker* as an integrated tool that implemented our generalizable framework using static-analysis techniques. We have now further enhanced *ChainTracker* to deal with complex *model-to-model* and *model-to-text* transformation execution semantics that heavily rely on non-trivial OCL expressions [24]. The most recent incarnation of *ChainTracker* uses natural-language processing techniques to collect traceability information in a lightweight and generalisable fashion for rule-based and template based transformation languages built-in OCL. The purpose of this paper is to showcase how *ChainTracker* provides a comprehensible analysis environment to solve real software-engineering tasks that developers have to complete during the construction of model-based code generators.

IV. Solving Developers' Tasks Using ChainTracker

ChainTracker is conceived to address three types of software-engineering tasks around model-based code gener-



```
                welcomed = true;
            }

        }

        private void setActivitiesToView(){

            // Setting activities
            [for (it : AppearActivity | activities)]
            [if (it.oclIsKindOf(AppearActivity))]

            AppearTimedEvent [it.id/]  = viewCanvas.new AppearTimedEvent();
            obstacleEvent.setObjectId(PhysicsView.[it.actorID/]Id);
            obstacleEvent.setX([it.positionX/]);
            obstacleEvent.setY([it.positionY/]);
            obstacleEvent.setaSpeed([it.angularSpeed/]);
            obstacleEvent.setlSpeed([it.linearSpeed/]);
            obstacleEvent.setFrequency([it.frequency/]*1000);
            viewCanvas.setActivity([it.id/] );
            [/if]
            [/for]
            return view;
        }

        @Override
```

Model Elements	Number of Attributes
Model	2
DataType	2

Backward Bindings	Source	Target
None	None	None
None	None	None

Forward Bindings	Source	Target
main	Phydsl.Model.game	Dynamics.Dynamics.activitie
main	Phydsl.Model.game	Dynamics.Dynamics.environ

Fig. 2. ChainTracker - Main Screen (1) Transformation-composition Overview; (2) Transformation Script Viewers; (3) Metamodel Information Tables

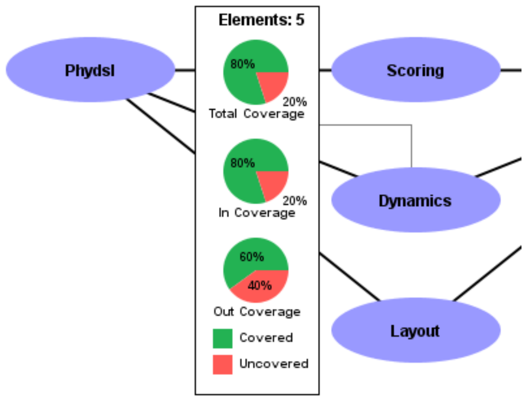


Fig. 3. ChainTracker - Metamodel Coverage Metrics

ators: *1. information discovery*, *2. information summarization*, and *3. information filtering and isolation*. Without *ChainTracker*, these tasks have to be performed manually over transformation-source scripts [11].

Information discovery tasks involve 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. 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 form a static point of view.

As a concrete example, let us briefly discuss *PhyDSL*, a model-based code generator for physics-based games [25][26]. Figure 1 illustrates how *ChainTracker* uses the traceability information of *PhyDSL* to enable the investigation of how source-metamodel elements are transformed into different intermediate metamodels (including how their attributes are being manipulated by transformation rules) and into final textual files. *ChainTracker* enables developers to not only explore the visualizations of the *model-to-model* and *model-to-text* transformations, but also to project the visualizations into the analyzed source-scripts using code-highlighters, thus isolating pieces of code of interest (Figure 1.2)

Information summarization tasks require developers to measure generic information from the code-generation architecture to identify branches of a transformation composition, quantify the coverage of a model transformation, and measure the size of the underlying metamodels. Summarizing information about the code-generator enables developers to assess and, potentially, optimize its overall design and correctness [4]. Figure 2 presents *ChainTracker*'s *Overview* mode, in which traceability information is used to provide a bird's-eye-view of the transformation composition. This view enables developers to abstract the complexity of individual and isolated transformation scripts into a single picture that summarizes the architecture of a given generation architecture. The blue nodes represent source and target artifacts of transformations (metamodels, in the case of *model-to-model* transformations, and text files, in the case of *model-to-text* transformations) and the edges represent transformation modules that operate over them.

Developers can click on the edges to pinpoint transformation rules that live inside of specific transformation branch or module modules (Figure 2 - yellow "sticky notes"). Furthermore, developers can zoom into a branch of interest (as shown in Figure 1). By clicking on individual nodes, developers can access summarized information about the coverage of a metamodel at different stages of the composition (see Figure 2). Coverage information is important since it allows developers to assess which parts of the source/target model elements have not been used in the generation of code, thus needing to be removed or included in the scope of the transformation modules [14]. Maximizing the coverage of model transformations makes transformation modules less convoluted and less error prone while, at the same time, freeing metamodels from unused semantic constructs. In *ChainTracker*, coverage metrics are summarized in contextual pie-charts, where the *in-coverage* metric reflects on the percentage of elements in a metamodel that are effectively targeted by transformation bindings in the composition, and the *out-coverage* metric represents the percentage of elements in a metamodel used as a source in transformation bindings to either generate code, or to create an intermediate model by the transformation-composition under analysis.

Information filtering and isolation tasks involve connecting and isolating elements of the code-generation architecture, in order to find relationships between metamodel elements, metamodel attributes, transformation bindings, and generated lines of code. These tasks are performed when developers are assessing the impact of platform-evolution scenarios. *ChainTracker* includes a variety of filtering mechanism to isolate metamodel elements and trace how elements relate across the end-to-end transformation chains, including domain-specific models, intermediate metamodels, and generated source-code files (see Figure 1, at 4.a and 4.b). We are currently extending *ChainTracker* to include symbolic and static analysis of generated code, in order to automatically suggest *when* and *how* refined portions of generated code need to be backwardly propagated to the code-generators, using the traceability information already collected and visualized in *ChainTracker*. In *http://hypatia.cs.ualberta.ca/~guana/chaintracker.html* you can find a video that showcases a developer conducting a variety of the aforementioned tasks using *ChainTracker*.

V. Conclusions

In this paper, we presented *ChainTracker*, a traceability collection and visualization analysis environment for model-based code generators. *ChainTracker* enables developers to solve three types of software-engineering tasks around model-based code generators: *1. information discovery, 2. information summarization,* and *3. information filtering and isolation*, when dealing with the platform evolution of code generators and the analysis code-generation architectures. The novelty of our work is twofold. First, in *ChainTracker* traceability information is used to reduce the cognitive challenges developers face when using disparate model-driven engineering technologies. Second, *ChainTracker* is an integrated environment that uses traceability information to allow developers interactively explore model-transformation scripts. Our future research agenda aims at conducting empirical studies with developers using *ChainTracker*, to accurately validate how our tool and traceability collection techniques help developers when using model-driven engineering technologies in real code generators.

Acknowledgements

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

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] L. Kuzniarz, J. L. Sourrouille, and M. Staron, "Workshop on quality in modeling," 2007.

[16] J. Gray, Y. Lin, and J. Zhang, "Automating change evolution in model-driven engineering," *Computer*, vol. 39, no. 2, pp. 51–58, 2006.

[17] 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.

[18] 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.

[19] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2010, pp. 301–304.

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.