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 (ICSM14). Victoria, BC. September 28th to October 3rd

ChainTracker: Towards a Comprehensive Tool for Building Code-Generation Environments

Victor Guana
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
guana@ualberta.ca

Kelsey Gaboriau
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
gaboriau@ualberta.ca

Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
stroulia@ualberta.ca

***Abstract*—Code-generation environments have emerged as a new mechanism for building software systems in a systematic manner. At their core, model-driven engineering technologies such as *model-to-model* and *model-to-text* transformations are effectively used to build generation engines. However, due to the complexity of *model-to-model* and *model-to-text* transformation scripts, which is exacerbated as they are composed in complex transformation chains, developers face technical and cognitive challenges when architecting, implementing, and maintaining code-generation environments. In this paper we present *ChainTracker*, a visualization and trace analysis tool for *model-to-model* and *model-to-text* transformation compositions. *ChainTracker* aims to support developers of code-generation environments by making the usage of model-driven engineering technologies more efficient, less error prone, and less cognitively challenging.**

I. Introduction

Code-generation environments systematize and (partially) automate the process of building software systems, aiming at improved software specifications, more efficient software construction, increased software flexibility, and reduced coding errors [1]. These environments adopt model-driven engineering (MDE) technologies, relying on textual and graphical domain-specific languages for software specification, and on model transformations for code generation [2].

Domain-specific languages encapsulate a cohesive set of constructs, in terms of which one can specify a software system. Model transformations provide the translation mechanism through which to reify the abstractions expressed in these languages, and to translate them into textual artifacts such as code and deployment scripts. The transformations work by injecting execution semantics into the application specifications, using a composition of *model-to-model* and *model-to-text* transformation scripts [1]. Each transformation iteratively maps the information contained in its input models into its output models, reducing its abstraction level; at the end of the transformation chain, the final set of executable artifacts is obtained. The main advantage of this software-construction methodology lies in the simplified semantics that can be used to specify a software system from a particular perspective, and the reusable (and previously engineered) execution artifacts that can be derived from them.

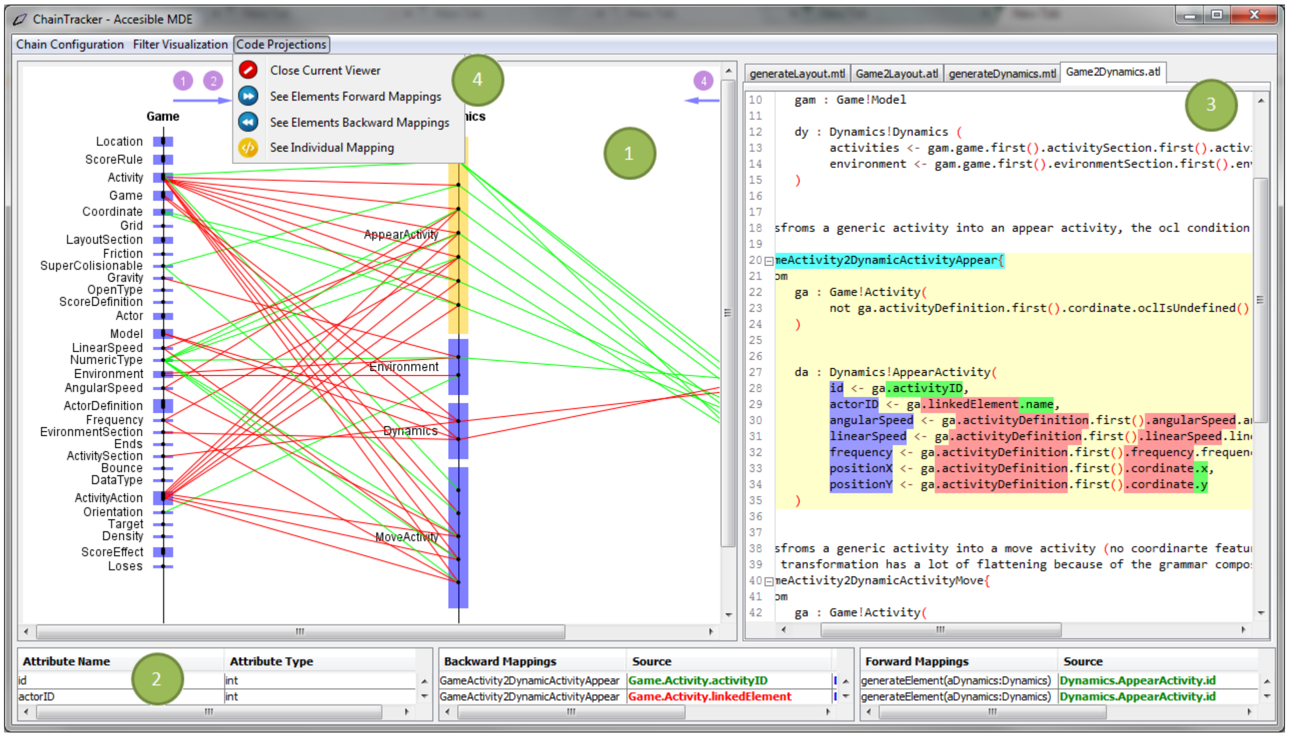
In 2011, a report on the adoption and usage of MDE techniques for building software systems [3] exposed that, even though code-generation environments significantly increase development productivity, practitioners face multiple challenges when maintaining these environments and evolving them to answer to new generation requirements. Furthermore, due to the complexity of the technologies behind *model-to-model* and *model-to-text* transformations, building new code-generation environments is expensive, error prone and cognitively challenging for developers [4]. Most of these difficulties are due to the fact that non-trivial model transformation are hard to visualize, and that, more often than not, multiple transformation languages have to be integrated to create a code-generation environment with real generation capabilities.

In rule-based *model-to-model* transformation languages –such as ATL[5], RubyTL[6], and ETL[7]– transformation mappings are specified using rules that take input-model patterns (also called source metamodels), and produce output-model patterns (or target metamodels) that generally modify (i.e., reduce, split, merge, or augment) the information represented a given input model. In the case of *model-to-text* transformation languages –such as EGL[7] and Acceleo[8]– input models are used to inject model information in predefined code templates. They are usually implemented as modules of metacode and expansion rules, that select and print string pattens with valid code semantics, or non-executable text such as XML[9].

In this paper we present *ChainTracker*, an integrated visualization and trace analysis tool for *model-to-model* (M2M) and *model-to-text* (M2T) transformation compositions. *ChainTracker* is designed for developers of code-generation environments, making their usage of model-driven engineering technologies more efficient, less error prone, and less cognitively challenging. *ChainTracker* helps developers to understand how model-transformation compositions work by enhancing their development experience with rich interactive visualizations, model-transformation traceability information, M2M and M2T mapping filtering, and smart transformation script highlighting. The current *ChainTracker* prototype and a video showcasing its features is available at http://hypatia.cs.ualberta.ca/~guana/chaintracker.html

II. Related Work

Precious little related research exists on development tools that help developers to architect and implement code-generation environments using model-driven engineering technologies. In [10], van Amstel et al. present a visualization



```
   gam : Game!Model

   dy : Dynamics!Dynamics (
        activities <- gam.game.first().activitySection.first().activi
        environment <- gam.game.first().evironmentSection.first().env
   )


sfroms a generic activity into an appear activity, the ocl condition

meActivity2DynamicActivityAppear{
om
   ga : Game!Activity(
        not ga.activityDefinition.first().cordinate.oclIsUndefined()
   )


   da : Dynamics!AppearActivity(
        id <- ga.activityID,
        actorID <- ga.linkedElement.name,
        angularSpeed <- ga.activityDefinition.first().angularSpeed.an
        linearSpeed <- ga.activityDefinition.first().linearSpeed.lin
        frequency <- ga.activityDefinition.first().frequency.frequen
        positionX <- ga.activityDefinition.first().cordinate.x,
        positionY <- ga.activityDefinition.first().cordinate.y
   )


sfroms a generic activity into a move activity (no coordinarte featu
 transformation has a lot of flattening because of the grammar compo
meActivity2DynamicActivityMove{
om
   ga : Game!Activity(
```

Attribute Name	Attribute Type
id	int
actorID	int

Backward Mappings	Source
GameActivity2DynamicActivityAppear	Game.Activity.activityID
GameActivity2DynamicActivityAppear	Game.Activity.linkedElement

Forward Mappings	Source
generateElement(aDynamics:Dynamics)	Dynamics.AppearActivity.id
generateElement(aDynamics:Dynamics)	Dynamics.AppearActivity.id

Fig. 1. ChainTracker - Main Screen (1) Model-transformation Composition Visualizer; (2) Model Element Information Tables; (3) Transformation Code Viewer (M2M); 4) Code Projections.

tool for ATL-M2M model-transformations. The tool portraits transformation mappings of transformation compositions using a hierarchical projection of model-transformation scripts. Similarly, von Pilgrim et al. [11] introduce a visualization tool for UniTL [12]. The proposal uses 3D projections to visualize a collection of 2D models that represent transformation-composition results. Both tools, however, lack capabilities to integrate M2M and M2T transformations in a consolidated view. Furthermore, the visualizations are not connected to the transformation scripts they represent. *ChainTracker* visualizes model-transformation compositions that include both M2M and M2T transformations, specifically, our current implementation is tailored for ATL and Acceleo M2M and M2T transformation technologies, but can be easily extended to other transformation technologies. Moreover, *ChainTracker* provides smart visualizations together with a code-viewer that allows developers to link visualization elements with code elements in the transformation scripts, using different information resolutions and filtering mechanisms.

III. Building and Maintaining Code-generation Engines

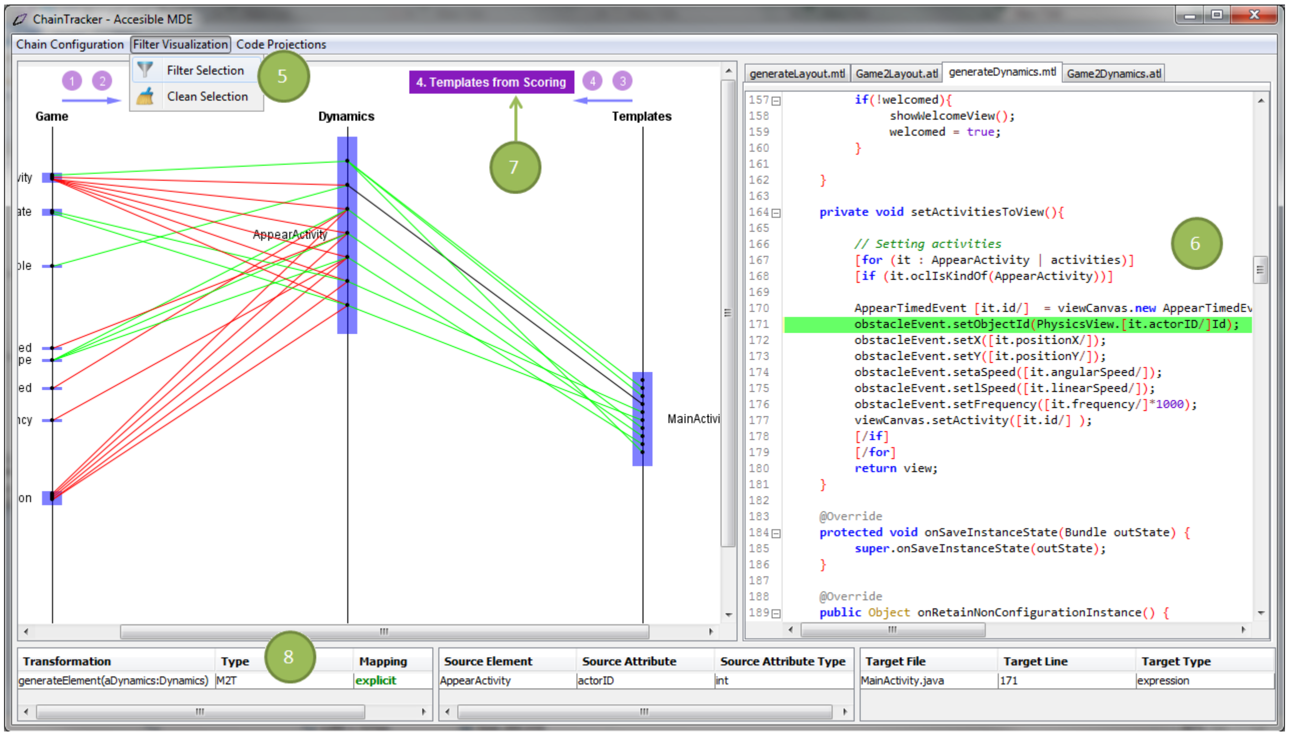
ChainTracker helps developers answer the following types of questions during the construction and maintenance of model-transformation compositions.

- *Where does this code feature come from?*
- *What downstream and upstream elements would be affected if a model element were modified?*
- *What is the coverage of the transformation rules in each stage of my transformation composition?*
- *How is my transformation chain affected if the generated code evolves and changes have to be propagated?*

ChainTracker consists of two major components. The *backend* is responsible for parsing and interpreting M2M and M2T transformation scripts, and identifying *explicit* and *implicit* transformation mappings between the input and output metamodels of each transformation stage. In [13], we presented a generic conceptual framework to gather traceability information in ruled-based M2M transformation languages, and template-based M2T languages. Furthermore, we showcased the concrete implementations of two lightweight language parsers and interpreters for ATL and Acceleo 3.0, as examples of popular and widely adopted M2M and M2T model-transformation technologies. In our framework, while explicit transformation mappings represent how model elements are directly mapped in transformation rules, implicit mappings gather information about intermediate model elements, and relations, used in mapping navigation statements to obtain concrete mapping values. *ChainTracker*'s *frontend* uses the traceability information collected in the *backend* of the tool, and provides three major interactive visualization components. a) The model-transformation composition visualizer (Figure 1 – point 1), b) the mapping/element information tables (Figure 1 – point 2), and c) the transformation code viewer (Figure 1 – point 3).

IV. Understanding Model-transformation Compositions

ChainTracker receives multiple M2M and M2T transformation scripts as input, with their source and target metamodels as Ecore files. *ChainTracker*'s visualizer (Figure 1 –



```
        if(!welcomed){
            showWelcomeView();
            welcomed = true;
        }

    }

    private void setActivitiesToView(){

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

        AppearTimedEvent [it.id/]  = viewCanvas.new AppearTimedEv
        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
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
    }

    @Override
    public Object onRetainNonConfigurationInstance() {
```

Transformation	Type	Mapping
generateElement(aDynamics:Dynamics)	M2T	explicit

Source Element	Source Attribute	Source Attribute Type
AppearActivity	actorID	int

Target File	Target Line	Target Type
MainActivity.java	171	expression

Fig. 2. ChainTracker - (5) Filters; (6) Transformation Code Viewer (M2T); (7) Branch Identifiers; (8) Mapping Information Tables.

point 1) portraits the structure of *model-to-model* and *model-to-text* transformation compositions, with a parallel-coordinate visualization for hyperdimensional data [14]. In this visualization, vertical lines represent models, blue rectangles on these lines represent model elements, and black dots inside the rectangles portray element attributes. Additionally, the visualization presents all the implicit and explicit transformation mappings involved in the creation of a target model from a set of source models, using red and green lines respectively. As model-transformations grow in size and complexity, so does the number of elements in the visualization, and it becomes increasingly difficult to understand what they represent. *ChainTracker* includes filtering mechanisms (see Figure 2 – point 5) for isolating forward and backward mappings connected to individual model elements, or sets of model elements, that when selected change in color from blue to yellow.

ChainTracker's visualizer also takes into account the branching structure of complex transformation compositions. In some cases, a single model can serve as input to multiple transformation scripts. For example, imagine a model that defines the gameplay design of a video game, this model can serve as input to multiple transformation scripts that create models for independent concerns of the gameplay design, such as layout, scoring rules, and game dynamic behaviors. In this case, the three independent target models will have the same source model, and as a consequence, three individual set of mappings need to be visualized. Furthermore, branching in transformation compositions may happen at any point of the transformation composition architecture. To that end, *ChainTracker* provides branching identifiers (Figure 2 – point 7). By hovering over the identifiers, developers can find information about the existing forward or backward transformation branches. By clicking on them, the visualization changes to portray how the composition models and mappings look from that specific point of view. This feature is particularly useful when developers are trying to understand how M2T mappings create different target textual files, from multiple source models.

ChainTracker's visualization highlights only the most important information of a transformation composition, and hides the details about model elements and transformation mappings. However, the mapping/element information tables display detailed information for model elements (Figure 1 – point 2) and individual transformation mappings (Figure 2 – point 8). The tables are context dependent; if the user selects a model element, three independent tables display information about the names and types of an element's attributes, its backward mappings (transformation mappings that have as target the selected element), and forward mappings (transformation mappings that use the selected element as the source for the creation of another element). If the user selects a transformation mapping, information about the transformation type, transformation module, and source and target attributes will be displayed. Particularly, if the selected mapping is a M2T mapping, information regarding the location inside M2T code-templates where the mapping occurs, and the type of expression used for the generation (loop or simple expression) is also presented.

V. Understanding Model-to-Model and Model-to-Text Transformations

A key aspect we considered in *ChainTracker*'s design and implementation is the need of developers for linking visualizations, filtering mechanisms, and detailed information about elements of the composition, with the actual transformation scripts that they represent. To meet this requirement, *ChainTracker* includes what we call *code projections* (Figure 1 – point 4). They work together with the code-viewer in order to highlight information *from the visualizations, to the transformation-script code*.

The *code projections* work using three different *lenses* that project and highlight different transformation statements depending on the selection context. Firstly, when one or many model elements are selected in the visualization, the *Elements' Backward Mappings* lens allows developers to open a code-viewer for each transformation script that contains transformation rules that mapped model elements into them, and then highlights implicit and explicit mappings following the same color schema used in the visualization, blue for model elements and attributes, and red/green for implicit and explicit mappings (see Figure 1 – point 3). Similarly, using the *Elements' Backward Mappings* lens, mappings in the transformation scripts, that use the selected element(s) as source for a transformation, are highlighted. Lastly, the *Individual Mapping* lens, allows developers to select and locate information about an individual mapping in the transformation source code (Figure 2 – point 6).

VI. Experience Report

ChainTracker has been actively used on building and maintaining *PhyDSL*, a code-generation environment that uses model-driven engineering technologies to create 2D physics-based games for mobile devices. *PhyDSL*'s generation architecture is composed by an initial model that captures domain-specific information about 2D gameplay definitions, together with three ATL-M2M transformation scripts that create independent models for a game's layout, scoring mechanisms, and game actor behaviors, and three Acceleo-M2T code-templates capable of generating Java/Android source code. In summary, *PhyDSL*'s code-generation environment contains 25 *model-to-model* transformation rules, 85 individual model mappings, and 105 *model-to-text* code injection statements. Figures 1 and 2 showcased the "Game Definition" to "Game Dynamics" M2M transformation branch of *PhyDSL*, and "Game Dynamics" to "Code" M2T mappings respectively.

During our experience building and maintaining *PhyDSL*, we encountered difficulties dealing with ATL-M2M transformation scripts that heavily use helper and called rules. Similarly, Acceleo-M2T statements with import statements were failed to process during parsing and mapping recollection. We are currently working on improving *ChainTracker*'s *backend* in order to support more complex transformation rules with rule inter-procedural calls (for helper and lazy rules in the case of ATL), and complex import and OCL expressions (in the case of our Acceleo 3.0 parser implementation). Furthermore, we would like to include code-interaction capabilities that allow developers to simply highlight portions of generated code, and trace them back to M2T and M2M transformation mappings. A preliminary idea on our research agenda towards this direction can be found in [15].

VII. Conclusions

In this paper we presented *ChainTracker*, an integrated development tool that helps developers understand model-transformation compositions. Based on a parallel-coordinates visualization, *ChainTracker* supports a rich interaction with model-transformation traceability information, transformation mapping filtering, and smart transformation-script highlighting through *code projections* and *lenses*. *ChainTracker* is designed to make the usage of model-driven engineering technologies more efficient, less error prone, and less cognitively challenging.

Our future work will explore two primary avenues of investigation. First, we want to extend *ChainTracker* to deal with more complex, and modularized, M2M and M2T scripts. Second, we are currently planning empirical studies that will examine how *ChainTracker* effectively improves the experience of developers, and reduce their cognitively challenges, when building code-generation environments using model-driven engineering technologies.

References

[1] S. Mellor, T. Clark, and T. Futagami, "Model-driven development: guest editors' introduction." *IEEE software*, vol. 20, no. 5, pp. 14–18, 2003.

[2] K. Czarnecki, "Overview of generative software development," in *Unconventional Programming Paradigms*. Springer, 2005.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[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] 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.

[11] 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, pp. 17–32.

[12] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers, "Uniti: A unified transformation infrastructure," in *Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 31–45.

[13] V. Guana and E. Stroulia, "Chaintracker, a model-transformation trace analysis tool for code-generation environments," in *Proceedings of the 7th International Conference on Model Transformation (ICMT14)*, 2014.

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

[15] 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.