ChainTracker, a Model-Transformation Trace Analysis Tool for Code-Generation Environments

Victor Guana and Eleni Stroulia

Department of Computing Science
University of Alberta
Edmonton, Canada
{guana,stroulia}@ualberta.ca

Abstract. Model-driven engineering is advocated as an effective method for developing families of software systems that systematically differ across well defined dimensions. Yet, this software construction paradigm is rather brittle at the face of evolution. Particularly, when building code-generation environments, platform evolution scenarios force developers to modify the generated code of individual generation instances in an ad-hoc manner. Thus violating the systematicity of the original construction process. In order to maintain the code-generation environment synchronized, code refinements have to be traced and backwardly propagated to generation infrastructure, so as to make these changes systematically possible for all systems that can be generated. This paper presents *ChainTracker*, a general conceptual framework, and model-transformation composition analysis tool, that supports developers when maintaining and synchronizing evolving code-generation environments. *ChainTracker* gathers and visualizes *model-to-model*, and *model-to-text* traceability information for ATL and Acceleo model-transformation compositions.

1 Introduction

Code-generation environments automate and systematize the process of building families of software systems. They typically rely on one or more domain-specific languages, and a set of model transformations that reify the abstractions expressed in the domain models and generate executable code [1]. The transformations work by injecting execution semantics into the initial problem specification, through a composition of *model-to-model* and *model-to-text* transformation modules.

Like all software, code-generation environments are bound to evolve [2]. Recent empirical studies revealed that practitioners face challenges when new requirements arise, and changes have to be introduced in either the source code of a generated application, or the domain-specific languages and the model-transformation compositions involved in the code-generation process [3].

Although, in principle, developers avoid modifying the code of a system after it is generated, approximately 40% end up having to do so [3][2] and, when they do, they have to spend copious amounts of time inspecting how changes

D. Di Ruscio and D. Varró (Eds.): ICMT 2014, LNCS 8568, pp. 146–153, 2014.
© Springer International Publishing Switzerland 2014

impact models and transformations, so changes can be backwardly propagated to the generation environments, and later reused in the generation of future systems. So far, little progress has been made towards supporting developers when performing these modifications during the construction and maintenance of code-generation environments.

The work we describe in this paper makes two novel contributions. The first is a general conceptual framework that formalizes how to model and collect traceability information in code-generation environments with model-transformation compositions that use (i) rule-based transformation languages to implement *models-to-model* transformations, an (ii) template-based languages to implement *model-to-text* transformations, distinguishing between explicit and implicit traceability links. The second contribution of our work is *ChainTracker*, a model-transformation composition analysis tool that supports developers when maintaining and synchronizing evolving code-generation environments. *ChainTracker* gathers and visualizes *model-to-model*, and *model-to-text* traceability information for ATL [4] and Acceleo [5] model-transformation compositions (as examples of the above rule-based *model-to-model* and template-based *model-to-code* transformation languages).

2 Background and Related Work

In principle, traceability information can be used in multiple ways, including to assess metamodel coverage in a code-generation environment, to verify model-transformation correctness, and to reduce the cognitive challenges when understanding a model-transformation chain [6][7]. However, most of the time, traceability information is collected manually or through experimental tools. More importantly, all current tools are unable to examine the *model-to-text* transformations, ignoring the last step in the model-transformation composition and effectively relying on developers for mapping code changes to their upstream dependent generation infrastructure.

Let us now review in some detail current approaches to traceability in model-driven engineering. Falleri, et al. [8] propose an imperative language in order to create trace models inside individual model-transformation modules. In this proposal, developers have to insert traceability constructs inside the transformation code to gather the traceability information of a transformation module. Similarly, Jouault [9] presents a strategy to keep track of ATL trace links by extending model-transformation rules with ATL constructs that build a traceability model conforming to a traceability metamodel proposed by the same author.

Van Amstel et al. [10] present a tool that gathers and visualizes traceability information of transformation compositions. In this case, the implemented tool makes explicit the mappings between source and target elements of a transformation, highlighting the hierarchical structure of both metamodels and ATL transformation modules. Jouault's proposal does not provide insights on possible visualization mechanisms to reduce the cognitive challenges of coping with massive amounts of information derived from complex model-transformation

compositions. Furthermore, none of the proposals presented above provide any type of support to collect or visualize traceability information for *model-to-text* transformations.

In Section 3 we present *ChainTracker*'s implementation architecture and visualization mechanisms. *ChainTracker* works as a third-party tool that analyses model-transformation compositions (that include *model-to-text* mappings), keeping the semantics of transformation rules intact, and providing an orthogonal set of metamodels that contain traceability information by statically interpreting a set of transformation rules that have been composed in order to generate code. In Section 3, we also introduce the concept of implicit traceability links (not covered by the current proposals). Implicit traceability links augment the traceability analysis by identifying indirect relations between source and target metamodels. This information provides additional support to developers when analysing the impact of changes in metamodels and transformations, that need to be synchronized after generated code refinements.

3 The ChainTracker Architecture

As shown in Figure 1, the architecture of *ChainTracker* consists of four main components: the *ATL Parser*, the *Tuple Extractor*, the *Acceleo Parser*, and the *Tuple Visualizer*. *ChainTracker* receives as input all the relevant transformations of a model-transformation composition to be analysed (ATL scripts for *model-to-model* and Acceleo scripts in the case of *model-to-text* transformations).

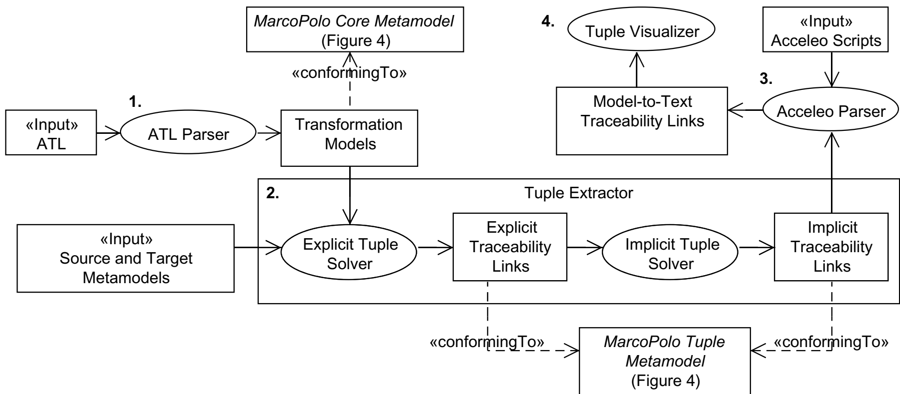

Fig. 1. ChainTracker Implementation Architecture

3.1 A Transformation Composition Example

We will illustrate the *ChainTracker* process using a simple model-transformation composition example. The goal of the composition is to refactor the elements of a model conforming to the $MetamodelA$, and produce a model conforming to $MetamodelB$, both portrayed in Figure 2. Then, the composition generates a

Java class that contains attributes initialized using elements of the latter model. Listings 1.1 and 1.2 present our *model-to-model* and *model-to-text* transformation examples respectively.

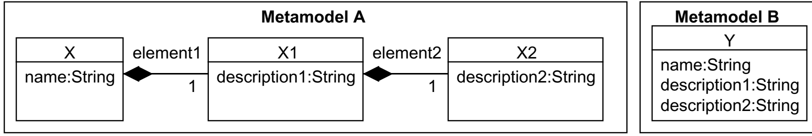

Fig. 2. Metamodel A (source) and Metamodel B (target) examples

3.2 The ATL Parser

The main functionality of the *ATL Parser* is to read, parse, and simplify a set of ATL transformation scripts. *ChainTracker* uses the reflexive capabilities of ATL's virtual machine to obtain the XMI-AST representation of a set of ATL scripts. *ChainTracker* implements a programmatic transformation that takes the XMI model of an ATL script, and produces a simplified representation that contains all the information relevant for the traceability link recollection. The resulting model conforms to *MarcoPolo*, a metamodel that we have designed in order to highlight transformation mappings in rule-based and template-based transformation languages (Figure 3). *MarcoPolo* is composed by two main packages, *MarcoPolo Core* and *MarcoPolo Tuple*. In this particular case, *MarcoPolo Core* is conceived to manage the complexity of transformation tuples that represent ATL transformation mappings. Effectively, we use *MarcoPolo "to find our way"* through the traceability links of a model-transformation composition.

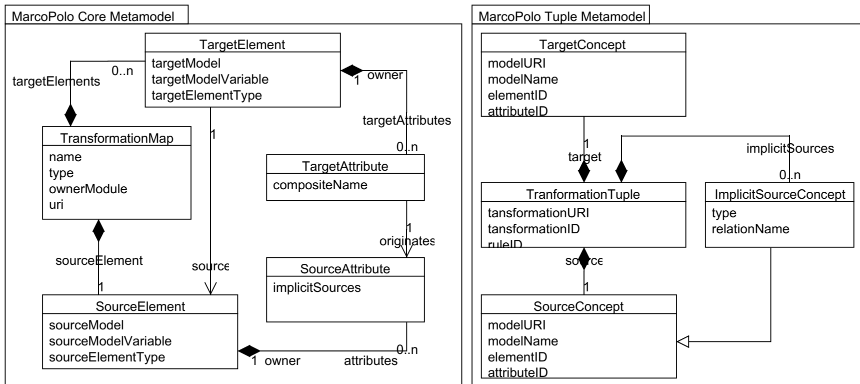

Fig. 3. MarcoPolo Metamodel

In *MarcoPolo Core*, we see each transformation module as a 3-tuple (TM, TE, se), where TM is the set of transformation rules, and TE a collection of its target-model elements. TE is defined as a tuple (TA, se) in which TA is a set of

target attributes ta, and se a unique source-model element. Furthermore each source-model element se contains a set, namely SA, that represent multiple source attributes sa. Finally, ta is modelled as a nested tuple (ta, sa) establishing a one-to-one mapping from a target attribute to a source attribute. Following this definition, it can be seen that in *MarcoPolo* the origins of a target attribute come from one, and only one, source attribute. However, the attribute's *implicit source* concept could have pointers to other intermediate source elements that participate in the creation of a target element as explained below.

```
1 module A2B;
2 create OUT : B from IN : A;
3 rule X2Y {
4   from
5     x : A!X
6   to
7    y : B!Y (
8      name <- x.name,
9      description1 <- x.element1.description1,
10     description2 <- x.element1.element2.description2)}
```

Listing 1.1. ATL - A2B Transformation Module Example

In our example, after the $X2Y$ matched rule is parsed (Listing 1.1), a model conforming to *MarcoPolo Core* is produced with the following (ta, sa) tuples:

- $(Y : name, (X : name))$
- $(Y : description1, (X : element1/description1))$
- $(Y : description2, (X : element1/elenment2/description2/))$

On cursory examination, these tuples would be identified as all the traceability links that map the elements of the $MetamodelA$ into elements in of the $MetamodelB$. However, even though there are one-to-one mappings between the target and source attributes in the transformation, there are many more dependency links between the source and target metamodels. For example, the creation of the Y : *description*2 attribute in the $MetamodelB$, depends not only on the attribute $X2$: *description*2 of the $MetamodelA$, but also on the model associations *element*1 and *element*2, and the element $X1$ as well. If any of the associations changes, or if the element $X1$ disappears, the transformation $X2Y$ will be broken. In effect, there are two types of traceability links that need to be preserved and made visible: *explicit* and *implicit* traceability links. The former type reflects the dependencies between the endpoints of the mappings in a transformation rule (as shown above); the latter type includes the dependencies between metamodel elements and associations used to navigate or query the source metamodel, and select information relevant during the creation of a target attribute.

In order to be able to detect implicit traceability links, *MarcoPolo Core* includes the *implicit source* attribute as a part of the source attribute concept. The *implicit source* represents the relative path that a mapping rule follows when navigating source model concepts in order to create a target attribute (see Figure 3). After the ATL modules are parsed, the *implicit source* contains a chain of meta-associations and meta-attributes, often extracted from OCL expressions.

For example, in the context of the $X2Y$ rule, the *implicit source* value for the source attribute $X2 : description2$ is $X : element1/element2/description2/$. Notice how the *implicit source* does not include information about where the *element*1 and *element*2 associations come from, and if there is an intermediate element that binds them, in this case $X1$. Given that both OCL and ATL model-navigation expressions are solved in execution time, this information is not explicitly available in the ATL abstract syntax model. *ChainTracker*'s *Tuple Extractor* implements an ATL interpreter that takes the source attribute context together with its *implicit source*, in order to identify where the intermediate associations and intermediate attributes come from.

3.3 The Tuple Extractor

The main functionality of the *Tuple Extractor* component is to analyze every source-to-target mapping and identify sets of explicit and implicit traceability links. For that purpose, the *Tuple Extractor* takes as input a set of models conforming to *MarcoPolo Core* that represent all the mappings between source and target models implemented in a transformation script. It also takes all the intermediate metamodels used in the composition as input and output patterns.

The *Tuple Extractor* consists of two sub-modules (Figure 1). While the *explicit tuple solver* takes a set of *MarcoPolo* instances and extracts all the explicit transformation links for a given transformation mapping. The *implicit tuple solver* finds the intermediate or navigated concepts involved in a given transformation rule. These concepts can be either metamodel elements or associations. In our example, the *implicit tuple solver* will take a (ta, sa) tuple such as $(Y : description2, (X : element1/elenment2/description2/))$, and through a recursive exploration of the $A2B$ source metamodel, it will discover the three implicit traceability links:

- $(Y : description2, X : element1)$ Association *element1* that belongs to X
- $(Y : description2-> X1 : element2)$ Association *element2* that belongs to X1
- $(Y : description2-> X2 : description2)$ Element *X2* and *description2* attribute

The final result of the *Tuple Extractor* module is a set of *MarcoPolo Tuple* instances that portray the explicit and implicit traceability links of a given set of ATL transformation scripts.

3.4 The Acceleo Parser

So far we have described how *ChainTracker* collects traceability information from *model-to-model* transformations. The *Acceleo Parser* identifies transformation tuples that map model elements into text artifacts. It takes an Acceleo script together with the metamodel that the script uses as input, and statically analyses its code-injection statements. *Model-to-text* traceability links are modelled in the form of tuples with the following structure $((startLineID, endLineID), (moduleID, fileID, sourceModelID, source\ ElementID))$. In

the tuples, *startLineID* and *endLineID* specify the initial and final code line identifiers where a specific source element is queried for a code injection statement, or used in an Acceleo model navigation construct.

```
1  [module B2Java('http://ualberta.edu.cs.ssrg.cge.b')]
2  [template public generateElement(yB : Y)]
3  [comment @main/]
4  [ file ('Generated.java', false , 'UTF−8')]
5   public class Generated {
6      [for (it : Y | yB)]
7       private Y [it.name/];
8      [/for]
9         public Generated (){
10        [for (it : Y | yB)]
11           [it.name/] = new Y([it.description1/], [it.description2 /]);
12        [/for]}}
13 [/ file ]
14 [/template]
```

Listing 1.2. Acceleo 3.0 - B2Java Transformation Module Example

After analysing the Acceleo *model-to-text* transformation script presented in Listing 1.2, the *Acceleo Parser* identifies traceability links such as $((13, 13), (B2Java, Generated.java, MetamodelB, Y : description1))$

3.5 The Tuple Visualizer

In order to communicate the traceability information to developers, *ChainTracker* includes a web-based traceability-visualization tool implemented in the *Tuple Visualizer*. Figure 4 presents the visualization of the traceability link tuples obtained using *ChainTracker*'s *Tuple Extractor*, and the *Acceleo Parser* for our *A2B (model-to-model)* and *B2Java (model-to-text)* composition example.

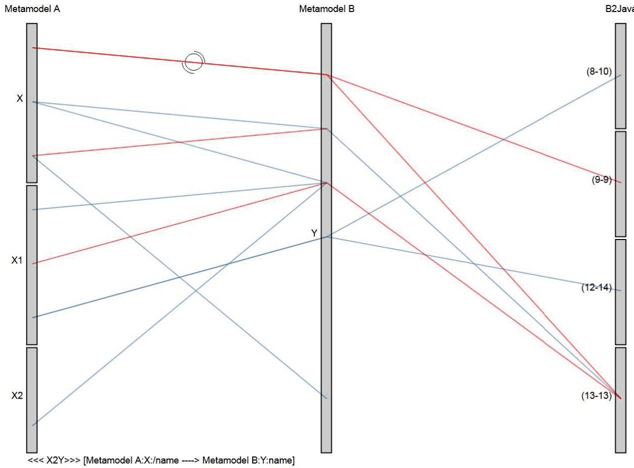

Fig. 4. Model-Transformation Composition Traceability Visualization

In Figure 4, red lines represent explicit traceability links according to *MarcoPolo*'s definition, and blue lines represent implicit traceability information of the composition. The details of the transformation tuples behind the links can be obtained by hovering the cursor over a link.

4 Conclusions and Future Work

In this paper we described *ChainTracker*, a tool designed to support the maintenance and evolution of code-generation environments. In the face of an environment's platform evolution, *ChainTracker* can support developers to trace ad-hoc modifications, from the generated code to is generation environment, thus enabling corresponding changes to the generation infrastructure so as to make these changes systematically possible for all systems that can be generated. *ChainTracker* is currently aware of the ATL and Acceleo transformation syntaxes, which it parses to extract traceability information in its syntactically simpler *MarcoPolo* metamodel. The second contribution of our work, beyond *ChainTracker* itself, is the conceptual framework underlying the design of the tool that formalizes how we model, and collect traceability information in code-generation environments, distinguishing between explicit and implicit links and capturing both in *MarcoPolo*. We believe that this framework is general and can support the extension of *ChainTracker* to deal with other transformation technologies, beyond ATL and Acceleo.

References

1. Czarnecki, K.: Overview of generative software development. In: Banâtre, J.-P., Fradet, P., Giavitto, J.-L., Michel, O. (eds.) UPP 2004. LNCS, vol. 3566, pp. 326–341. Springer, Heidelberg (2005)
2. Van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. In: Proceedings 1st International Workshop on Model-Driven Software Evolution, pp. 41–49 (2007)
3. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of mde in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 471–480. ACM (2011)
4. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
5. Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., Allilaire, F.: Acceleo user guide (2006)
6. Guana, V.: Supporting maintenance tasks on transformational code generation environments. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 1369–1372. IEEE Press (2013)
7. Guana, V., Stroulia, E.: Backward propagation of code refinements on transformational code generation environments. In: 2013 International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), pp. 55–60 (2013)
8. Falleri, J., Huchard, M., Nebut, C., et al.: Towards a traceability framework for model transformations in kermeta (2006)
9. Jouault, F.: 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)
10. van Amstel, M., Serebrenik, A., van den Brand, M.: Visualizing traceability in model transformation compositions. In: Pre-Proceedings of the First Workshop on Composition and Evolution of Model Transformations (2011)