Interactive Dependency Graphs for Model Transformation Analysis

Andreas Rentschler, Per Sterner

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
`andreas.rentschler@kit.edu, per.sterner@student.kit.edu`

Abstract. Model transformations are pivotal artifacts in model-driven software projects. As with any software product, maintenance significantly contributes to the lifecycle costs. This is particularly true for model transformations. We present *Transformation Analysis*, a tool that supplements existing transformation editors under Eclipse with a graph view on the program's data and control dependencies. The tool has been designed to extract and deliver select information to maintainers to help them understand behavior and locate relevant code quicker. To accomplish this goal, the integrated view is interactively navigable and offers preconfigured filter criteria.

1 Introduction

Together with metamodels, model transformations play an important role as first-class artifacts in model-driven software projects. As with any software product, changing requirements and new enhancements can significantly contribute to the lifecycle costs. This is particularly true for model transformations. Missing documentation of a transformation requires maintainers to first obtain or reobtain a basic understanding of the transformations behavior. This requires to study the underlying control flow. Furthermore, for adjusting the transformation to new or changing requirements, e.g. changes in involved models, and for tracing bugs, relevant places in the code need to be located in a preliminary step. Without a useful structuring of the transformation into submodules that encapsulate single concerns, maintainers are even more challenged to locate relevant concerns in the code. At the present time, in the area of model transformations, there are only few tools known for supporting the process of maintenance and understanding. Some language workbenches [1], for example Spoofax, provide more sophisticated techniques for transformation maintenance than Eclipse-based tools. Still, none of them integrates data and control flow information from static analysis into one single interactive view.

We present *Transformation Analysis* for Eclipse, a novel tool that supplements existing transformation editors under Eclipse with a graph view on the program's data and control dependencies. The tool has been designed to extract and deliver select information to maintenancers, in order to help them understand behavior and locate relevant code more efficiently. To accomplish this goal, the integrated view is interactively navigable and offers preconfigured filter criteria.

An early version of our tool served as a protoypical implementation of a visual analytics process [2] for QVT-Operational (QVT-O) transformations under Eclipse[1] that we presented at ICMT'13 [3]. In an empirical study described there, we showed that the tool is indeed able to increase efficiency. In the meantime, we added support for QVT-Declarative[2] and mediniQVT[3], both derivatives of QVT-Relations (QVT-R).

This paper is structured as follows. First, we provide a small motivating scenario in Section 2 from which we conclude requirements for the tool to implement. In Section 3, we present the implementation, and Section 4 describes how the tool helps to tackle both challenges from the motivated scenario. Finally, Section 5 concludes.

2 Application Scenario

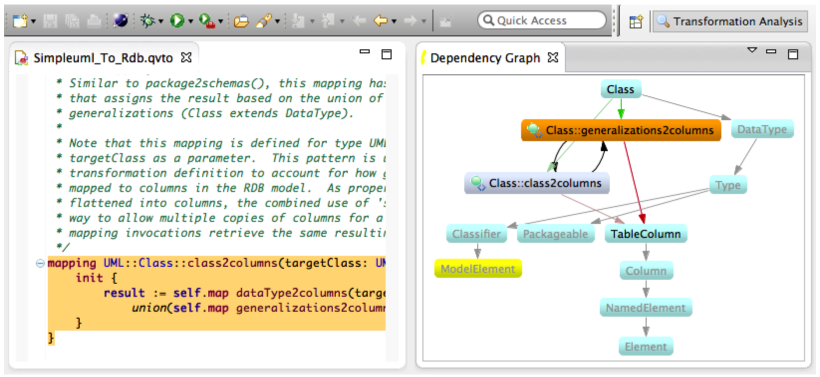
Generally, there are two basic activities for maintainers to do, understanding what the transformation does (comprehension of behavior), and carrying out various maintenance tasks. As a running example, we observe the UML2RDBMS usecase.

Comprehension. Similar to code in a procedural programming language, statements in QVT's imperative dialect QVT-Operational (QVT-O) are structured into mapping methods that can call other mappings, constructors and query methods. Regarding QVT's relational dialect, rules can similarly have dependencies to other rules and queries. To understand the transformation's behavior, developers need to first study these explicit and implicit control flow dependencies. Such information can be extracted and visualized automatically. While some transformations may be small enough to be readily comprehensible, in practice, transformations easily comprise a hundred and more mappings, depending on the size and heterogenity of the metamodels involved.

Maintenance. Whereas some maintenance tasks are similar to those carried out for programs in general-purpose languages, other tasks are domain-specific, e.g. adaptation to evolving metamodels (co-evolution). Co-evolution means that, for example, if new concepts are added to related metamodels, the transformation needs to be enhanced to support these concepts. Because of the often complex structure of metamodels, it is not easy to locate places in the code that deal with a certain class and subclasses thereof. If we would like to change the way class attributes are mapped to the RDBMS domain, we must identify all the mappings that deal with class `Property` or subclasses thereof. Existing Eclipse-based editors only support text-based search-and-replace functionality that is prone to return false positives and false negatives.

To improve the efficiency in understanding transformations and locating relevant places in the code, we designed a tool that provides a graphical view on mappings and other top-level methods, metamodel elements and attributes, and occuring control and data dependencies.

[1] `http://www.eclipse.org/mmt/qvto`
[2] `http://www.eclipse.org/mmt/qvtd`
[3] `http://projects.ikv.de/qvt`



```
 * Similar to package2schemas(), this mapping ha:
 * that assigns the result based on the union of
 * generalizations (Class extends DataType).
 *
 * Note that this mapping is defined for type UMl
 * targetClass as a parameter.  This pattern is
 * transformation definition to account for how
 * mapped to columns in the RDB model.  As proper
 * flattened into columns, the combined use of ':
 * way to allow multiple copies of columns for a
 * mapping invocations retrieve the same resulti
 */
mapping UML::Class::class2columns(targetClass: UM
    init {
        result := self.map dataType2columns(targe
            union(self.map generalizations2colum
    }
}
```

Fig. 1. The graph view augmenting the existing textual view of Eclipse QVT-O

3 The Dependence Graph View

We implemented software for the Eclipse platform that adds a *Dependency Graph* view that is able to interact with existing QVT editors under Eclipse.

Live Dependence Information. When editing a transformation in the Eclipse editor, dependence information is extracted from the parsed abstract syntax tree. Similar to the commonly supported outline views for Eclipse code editors, information is updated live. A graph structure is built, where nodes reflect top-level constructs of the program and referenced metamodel elements. In the QVT dialects, top-level constructs are mappings, helpers, constructors, relations, and queries. Referenced metamodel elements are modeled as classes and class attributes. The extraction algorithm recognizes different types of links to reflect inheritance between classes, containment between classes and attributes, and also read-only and modifying accesses to data elements.

Graph-based View. The graph represents the extracted dependence information visually as a node-link diagram. Visualization and automatic layouting is realized with the *Eclipse Zest* framework[4]. Control units are displayed as gray nodes. If one unit calls another, a black arrow connects both, pointing into the calling direction. Multiple calls are condensed to one arrow. Data elements are displayed in green. If a control unit reads or modifies an element, either a green arrow points to the control unit, or a red arrow points from the control unit to the accessed element. If an attribute is accessed, a dashed line is leading to a class element which contains the attribute. Multiple reads/writes are condensed to one arrow. Inheritance relationships among classes are depicted as a gray arrow pointing to the super class. They currently selected node is highlighted in orange. Nodes and links that are not in direct context of the actively selected element are slightly faded out to a lighter color.

The graph from Figure 1 is computed for mapping `TopClassToTable`. In its immediate context, there is one caller (`Package2Schema`), and two callees (`ClassToTableMarker` and `ClassToTableActual`). It maps elements of type `Class` to elements of type `Table`. Element `Class` subclasses `Classifier`.

[4] `http://www.eclipse.org/gef/zest/`

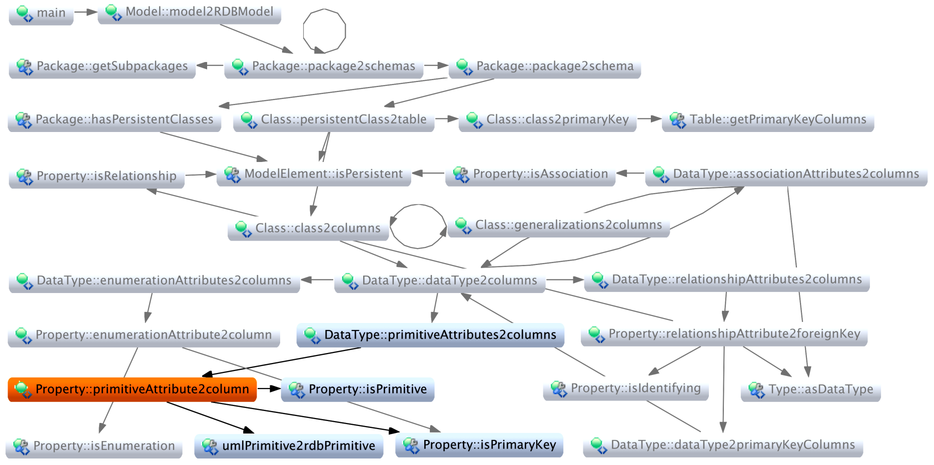

main
Model::model2RDBModel
Package::getSubpackages
Package::package2schemas
Package::package2schema
Package::hasPersistentClasses
Class::persistentClass2table
Class::class2primaryKey
Table::getPrimaryKeyColumns
Property::isRelationship
ModelElement::isPersistent
Property::isAssociation
DataType::associationAttributes2columns
Class::class2columns
Class::generalizations2columns
DataType::enumerationAttributes2columns
DataType::dataType2columns
DataType::relationshipAttributes2columns
Property::enumerationAttribute2column
DataType::primitiveAttributes2columns
Property::relationshipAttribute2foreignKey
Property::primitiveAttribute2column
Property::isPrimitive
Property::isIdentifying
Type::asDataType
Property::isEnumeration
umlPrimitive2rdbPrimitive
Property::isPrimaryKey
DataType::dataType2primaryKeyColumns

Fig. 2. Control dependence graph of UML2RDBMS (Filter F1)

Configurable Filters. The graph is processed by a chain of filter functions. There are four predefined filter configurations, they can be activated from a context menu that appears when right-clicking the graph canvas:

F1: Show control dependencies for the currently loaded transformation program.
F2: Show control dependencies in direct context of a selected control unit.
F3: Show control and data dependencies in direct context of a selected control or data unit.
F4: Same as filter three, but in addition, condense data dependencies to the class-level, i.e., if an attribute is accessed, the parent class is shown instead.

Synchronization with Textual View. The graph is navigable. A double-click on one of the control nodes leads to the corresponding method in the code editor. Our graph view is synchronized with the textual view, so the graphical view always reflects the current editing context.

4 Application Scenario Revisited

Now we take a second look at the UML2RDBMS scenario, this time we elaborate how comprehension and maintenance tasks can be supported by the tool. Download instructions for the tool are available at the tool's website.[5] The site also links to a screencast which illustrates how each of the presented requirements is reflected in the tool.

As example transformation for the UML2RDBMS usecase, we choose the implementation that belongs to the samples from the Eclipse QVT-O distribution.
Understanding the Call Dependencies. To understand the logic of a transformation, it is inevitable to track dependencies between the mappings. With the help of filter F1, a diagram of the call structure is no longer needed to be drawn using paper and pencil. Figure 2 demonstrates the result with mapping `primitiveAttribute2Column` selected in the text editor.

[5] `http://sdqweb.ipd.kit.edu/wiki/Transformation_Analysis`

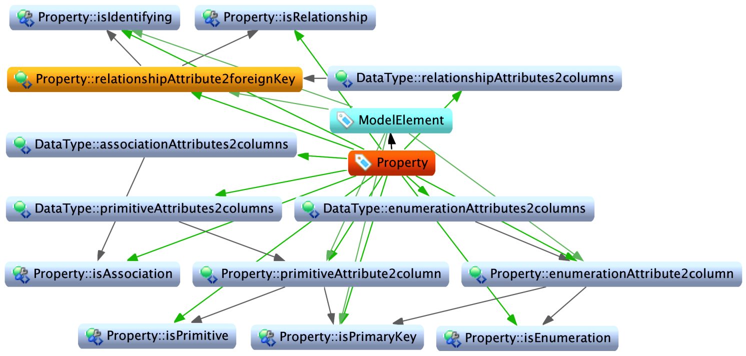

Property::isIdentifying
Property::isRelationship
Property::relationshipAttribute2foreignKey
DataType::relationshipAttributes2columns
ModelElement
DataType::associationAttributes2columns
Property
DataType::primitiveAttributes2columns
DataType::enumerationAttributes2columns
Property::isAssociation
Property::primitiveAttribute2column
Property::enumerationAttribute2column
Property::isPrimitive
Property::isPrimaryKey
Property::isEnumeration

Fig. 3. Data dependencies of UML2RDBMS in context of class `Property` (Filter F4)

Understanding Data Dependencies for Metamodel Evolution. The second scenario is about understanding and modifying transformation logic that is dealing with certain UML concepts in the source domain. In our view, we can apply filter F4 to obtain all the mappings and helper functions that deal with class `Property` (Figure 3).

5 Conclusion

In this paper, we demonstrated a novel tool that implements the visual analytics process in the particular case of model transformation maintenance. Our tool processes dependence information based on filter criteria and provides interactive visual feedback to maintainers. We subsequently demonstrated that the tool can help to ease comprehension and concern location in the introductory scenario.

Of course, static analysis brings certain weaknesses, as well – analysis abstracts from concrete input models, and OCL's reflection capabilities on types cannot be fully considered. Nevertheless, we found the tool's effect on the overall efficiency impressive: According to our experiences from a practical course on model-driven techniques, students whom we introduced to the tool in a short tutorial session promptly integrated it into their workflow. Some of them even utilized the graph representation for documentation purposes. In the future, refactoring operations could add additional value to our graph view.

Acknowledgements. This research has been funded by the German Research Foundation (DFG) under grant No. RE 1674/5-1.

References

1. Erdweg, S., van der Storm, T., Völter, M., et al.: The State of the Art in Language Workbenches. Conclusions from the Language Workbench Challenge. In: 6th Int. Conf. on Software Language Engineering (SLE'13). LNCS, Springer (2013)
2. Telea, A., Ersoy, O., Voinea, L.: Visual Analytics in Software Maintenance: Challenges and Opportunities. In: Int. Symp. on Visual Analytics Science and Technology (EuroVAST'10), Bordeaux, France, Eurographics Association (2010) 75–80
3. Rentschler, A., Noorshams, Q., Kapova, L., Reussner, R.: Interactive Visual Analytics for Efficient Maintenance of Model Transformations. In: 6th Int. Conf. on Model Transformation (ICMT'13). LNCS, Springer (June 2013)