ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 28 – October 3, 2014 • Valencia (Spain)

AMT 2014 – Analysis of Model Transformations Workshop Proceedings

Juergen Dingel, Juan de Lara, Levi Lúcio, Hans Vangheluwe (Eds.)

Published on Oct 2014 v1.0

Editors' addresses:

Juergen Dingel
Queens University, Canada

Juan de Lara
Universidad Autónoma de Madrid, Spain

Levi Lúcio
McGill University, Canada

Hans Vangheluwe
University of Antwerp, Belgium and McGill University, Canada

Organizers

Juergen Dingel	Queen's University (Canada)
Juan de Lara	Universidad Autónoma de Madrid (Spain)
Levi Lúcio	McGill University (Canada)
Hans Vangheluwe	University of Antwerp (Belgium) and McGill University (Canada)

Program Committee

Marsha Chechik	University of Toronto (Canada)
Juergen Dingel	Queen's University (Canada)
Alexander Egyed	University of Linz (Austria)
Joel Greenyer	University of Hannover (Germany)
Holger Giese	Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam (Germany)
Reiko Heckel	University of Leicester (UK)
Dimitris Kolovos	University of York (UK)
Kevin Lano	King's College London (UK)
Juan de Lara	Universidad Autónoma de Madrid (Spain)
Tihamér Levendowsky	Vanderbilt University (USA)
Levi Lúcio	McGill University (Canada)
Alfonso Pierantonio	University of L'Aquila (Italy)
Perdita Stevens	University of Edinburgh (UK)
Gianna Reggio	University of Genua (Italy)
Gabriele Taentzer	University of Marburg (Germany)
Antonio Vallecillo	University of Malaga (Spain)
Hans Vangheluwe	University of Antwerp (Belgium) and McGill University (Canada)
Dániel Varró	Budapest University of Technology and Economics (Hungary)
Manuel Wimmer	Vienna University of Technology (Austria)

Additional Reviewers

Ábel Hegedüs
David Lindecker
Bentley Oakes
Rick Salay

Table of Contents

Preface

To facilitate the processing and manipulation of models, a lot of research has gone into developing languages, standards, and tools to support model transformations. A quick search on the internet produces more than 30 different transformation languages that have been proposed in the literature or implemented in open-source or commercial tools. The increasing adoption of these languages and the growing size and complexity of the model transformations developed require a better understanding of how all activities in the model transformation life cycle can be optimally supported.

Properties of an artifact created by a model transformation are intimately linked to the model transformation that produced it. In other words, to be able to guarantee certain properties of the produced artifact, it may be very helpful, or even indispensable, to also have knowledge of the producing transformation. As the use and significance of modeling increase, the importance that the model transformations produce models of sufficient quality and with desirable properties increases as well; similarly, as the number and complexity of model transformations grows, the importance that transformations satisfy certain non-functional requirements and that life cycle activities for model transformations such as development, quality assurance, maintenance, and evolution are well supported grows as well.

The central objective of the AMT workshop is to provide a forum for the discussion and exchange of innovative ideas for the analysis of model transformations, broadly construed. Analyses might support a variety of model transformation activities including the development, quality assurance, maintenance and evolution by facilitating, for instance,

– the detection of typing errors, anti-patterns, dead code, transformation slices, likely invariants, or performance bottlenecks;
– the informal, semi-formal, or formal establishment of properties related to correctness or performance;
– test suite evaluation through code coverage determination;
– code completion and generation;
– the evolution of metamodels;
– impact analysis;
– refactoring.

Another objective of the workshop is to help clarify which transformation analysis problems can be solved with the help of existing analysis techniques and tools developed in the context of general-purpose programming languages and source code transformation languages, and which analysis problems require new approaches specific to model transformations. The exchange of ideas between the modeling community on the one hand and the programming languages community and source code transformation community on the other hand thus is another objective of the workshop.

In this third edition, AMT received 16 submissions, out of which 9 were accepted. The workshop also held a keynote speech by Ronan Barrett from Ericsson on exploring the non-functional properties of model transformation techniques used in industry. We are grateful to all authors, attendees, program committee members, external reviewers and local organizers for helping make AMT 2014 a success.

October 2014 Juergen Dingel, Levi Lúcio, Hans Vanghewluwe and Juan de Lara

Keynote

Exploring the non-functional properties of model transformation techniques used in industry

Ronan Barrett

Ericsson

Authoring model transformations is arguably the most resource intensive effort in the whole Model Driven Engineering (MDE) chain. The non-functional properties (NFP) of the techniques used to realize these transformations has an enormous impact on the quality of the model based systems being developed. It is always tempting, and someone will invariable offer, to conjure up a quick script to implement a transform. However, transformations are more often than not creatures with a long life span. They evolve in ways you would never have expected and so require excellent extensibility, readability and maintainability. They must of course also perform as well as their previous incarnation with the same or better levels of quality. These non-functional properties are not synonymous with hastily written scripts. We know from experience that making bad choices early on will cost later. In this talk we will share our experiences of authoring transformations using a number of different open source transformation techniques and how we have tried, and succeeded in most cases, to meet our NFP obligations.

Ronan Barrett *received his Ph.D from the School of Computing at Dublin City University, Ireland, in 2008. He is a Senior Specialist in Modeling Technologies & Transformations at Ericsson, Sweden. Since completing his Ph.D. Ronan has worked on applying model driven engineering concepts and technologies in industry. He has worked extensively with Eclipse based open source modeling technologies, working closely with the open source community and internally within Ericsson. Ronan has a wealth of experience in writing model transformations and designing domain specific language tools that meet demanding non-functional requirements. He has published a number of academic papers in the area of model driven engineering and has also presented at open source conferences like EclipseCon Europe.*

Remodularizing Legacy Model Transformations with Automatic Clustering Techniques

Andreas Rentschler, Dominik Werle, Qais Noorshams,
Lucia Happe, Ralf Reussner

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{rentschler, noorshams, happe, reussner}@kit.edu,
dominik.werle@student.kit.edu

Abstract. In model-driven engineering, model transformations play a critical role as they transform models into other models and finally into executable code. Whereas models are typically structured into packages, transformation programs can be structured into modules to cope with their inherent code complexity. As the models evolve, the structure of transformations steadily deteriorates, and eventually leads to adverse effects on the productivity during maintenance.
In this paper, we propose to apply clustering algorithms to find decompositions of transformation programs at the method level. In contrast to clustering techniques for general-purpose languages, we integrate not only method calls but also class and package dependencies of the models into the process. The approach relies on the Bunch tool for finding decompositions with minimal coupling and maximal cohesion.
First experiments indicate that incorporating model use dependencies leads to results that reflect the intended structure significantly better.

1 Introduction

The idea behind model-driven software engineering (MDSE) is to move the abstraction level from code to more abstract models. Although the principal aim of model-driven techniques is to improve the productivity, maintenance of models and particularly of transformation programs for mapping these models to less abstract models and finally to executable code remains costly. Studies on long-term experiences from industrial MDSE projects give evidence for maintenance issues that arise from constantly evolving models [1, p. 9].

As the complexity of models grows, model transformations tend to become larger and more complex. If transformation programs are not properly structured into well-understandable artifacts, understanding and maintaining model transformations is worsened.

However, as opposed to object-oriented code where data is encapsulated by the concept of classes, transformation units must consider not only the methods provided and required by a module, but also the scope of model elements that are used by a module to implement a particular concern. We recently proposed a module concept tailored for model transformation languages which introduces information hiding through an explicit interface mechanism [2]. Per interface, scoping of model elements can be defined on the package and class level. Further on, only those methods are accessible that are defined either locally or in one of the imported interfaces.

Although it is possible to use the added language concept to develop transformations with a modular design, according to our own experience, many existing

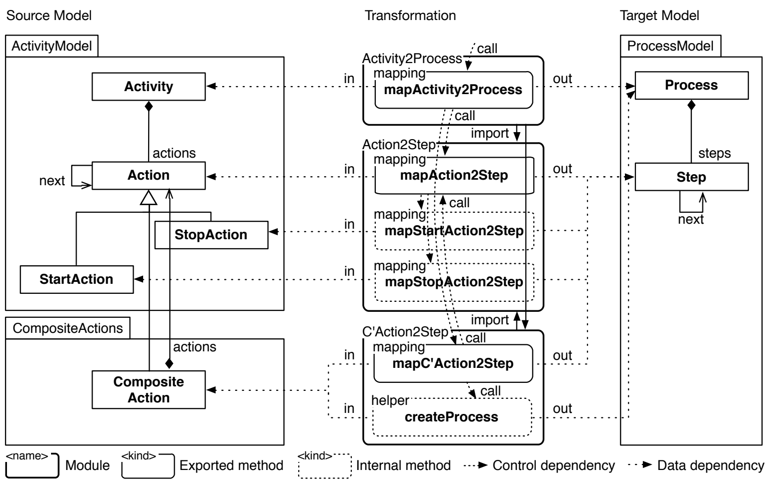


Fig. 1: Activity2Process transformation – Method and model scoping

transformations have been built monolithically, or their modular structure had been deteriorating over time. As it has been observed for software in general, deriving a module structure manually from legacy code can be cumbersome without an in-depth knowledge of the code. At the present time, there is no approach to derive such a structure from transformation programs automatically.

Existing clustering approaches [3] are able to derive module structures from source code. But in contrast to ordinary programs, transformation programs traverse complex data structures. Most model-to-model and model-to-code transformations are structured based on the source or target model structure. As we will show, model use relationships should be taken into account by automatic clustering approaches to produce useful results.

In this paper, we propose to carry out automatized cluster analysis based on a dependence graph that includes not only method calls, but also model use dependencies and structural dependencies among model elements.We use the Bunch tool [4], a software clustering framework that searches for clusters with minimal coupling and maximal cohesion. By integrating model information into the search process, found clusters are (near-)optimal regarding the scope of both methods and model elements.

Next, Section 2 motivates our previously published modularity concept for transformations as a way to improve maintainability, and presents methods how experts tend to structure transformation programs. Section 3 briefly introduces the Bunch tool, a prominent software clustering technique that is used in this paper. In Section 4, we present a novel approach for clustering model transformations. Section 5 presents relevant work that is related to our own work, and Section 6 concludes the paper and points out potential further work on the topic.

2 Modular Model Transformations

To explain how model transformations are structured in a way that improves maintainability, we are going to use a minimalistic example transformation Activity2Process implemented in QVT-Operational (QVT-O) [5], which maps activity diagrams to process diagrams (Fig. 1).

A transformation between the two models would be implemented with five mapping methods. Each method is responsible for mapping one class of the source domain to semantically equivalent elements in the target domain. Because the target model is less abstract, as it does not offer composite steps, hierarchical activity models are flattened by the transformation.

When decomposing the sample transformation into modules, we may identify three different concerns: The first module is responsible for mapping the container elements and to trigger the rest of the mappings. A second module can be assigned to the task of mapping actions to process elements. Internally, the module does further deal with start and stop actions. A third module can be made responsible for mapping the extended concept of composite actions to basic steps.

With our recently proposed module concept for model transformations (cf. [2]), it is possible to define this decomposition in a way that improves maintainability.

Explicit interfaces. We introduce a new language concept to declare module interfaces. With explicit interface declarations, it is possible to hide implementation details behind interfaces. For instance, the second module in Fig. 1 relies on two mapping functions that are only used locally and can be thus kept internal, `StartAction2Step` and `StopAction2Step` (marked by a dashed frame). By omitting these from the module's interface declaration, they remain invisible for the other two modules that import this module.

Method access control. Only method implementations that are either defined locally, or that are declared by one of the imported interfaces can be called. The first module in the example, for instance, is only able to access mappings `Activity2Process` and `Action2Step`.

Model visibility control. The second module in the Activity2Process scenario must only have access to three model elements in the source domain, `Action`, `StartAction`, and `StopAction`, and `Step` in the target domain. These classes can be specified in the module's interface declaration. It is statically checked that an implementation of the interface does not access elements outside of this scope. Scoping of model elements can also be declared at package-level, so the third module could list package `CompositeActions` as accessible.

Information-hiding modularity helps to improve understandability and maintainability, as the scope of a module can be directly grasped from its declared interface. Internal functions for querying model elements are hidden behind the interface, making it easier to understand the functionality provided by a module.

Keeping the scope of models and the number of modules that are imported at a minimum is obviously a prime concern; internally, mappings in a module may have arbitrary references to each other. This relates to two software metrics to measure the quality of a module decomposition, favoring a low degree of method and data interconnectivity between modules and a high degree of intraconnectivity of methods within a module (*low coupling* and *high cohesion*). In an optimal decomposition, each module encapsulates a single concern with a minimal model scope, and model scopes overlap for as few modules as possible.

By observing transformations that had been manually implemented by experts, we can distinguish three classic styles of how a transformation is structured [6].

Source-driven decomposition. In this case, for objects of each class in the source domain, objects of one or more classes are generated in the target domain (one-to-many mappings). Transformations where models are transformed to models that are equally or less abstract usually fall into this category. The Activity2Process transformation is a typical candidate for a source-driven decomposition. It traverses the tree-like structured activity model, and each node embodies an own high-level concept that is mapped to target concepts.

Target-driven decomposition. When objects of a particular class in the target domain are constructed from information distributed over instances of multiple classes in the source domain (many-to-one mappings), a target-driven decomposition is deemed more adequate. Transformations from low-level to high-level concepts (synthesizing transformations) use this style.

Aspect-driven decomposition. In several cases, a mixture of the two applies. Aspect-driven decompositions are required whenever a single concern is distributed over multiple concepts in both domains (many-to-many mappings). In-place transformations (i.e., transformations within a single domain) that replace concepts with low-level concepts often follow this style, particularly if operations are executed per concern and affect multiple elements in the domain.

Any of these styles – and preferably also mixtures – must be supported by an automatic decomposition analysis in order to produce meaningful results.

3 Automatic Software Clustering

The principal objective of software clustering methodologies is to help software engineers in understanding and maintaining large software systems with outdated or missing documentation and inferior structure. They do so by partitioning system entities – including methods, classes, and modules – into manageable sub systems. A survey on algorithms that had been used to cluster general software systems has been carried out by Shtern et al. [3]. They describe various classes of algorithms that can be used for this purpose, including algorithms from graph-theory, constructive, hierarchical agglomerative, and optimization algorithms.

In this paper, we employ the Bunch tool, a clustering system that uses one of two optimization algorithms, hill climbing or a genetic algorithm, to find near-optimal solutions [4]. Bunch operates on a graph with weighted edges, the so-called *Module Dependency Graph* (MDG). Nodes represent the low-level concepts to be grouped into modules, and may correspond to methods and classes. As a fitness function for the optimization algorithms, Modularization Quality (MQ) is used, a metric that integrates coupling and cohesion among the clusters into a single value. Optimization starts with a randomly created partitioning, for which neighboring partitions – with respect to atomic move operations – are explored.

According to Mitchell et al. [4], a dependency graph is a directed graph $G = (V, E)$ that consists of a set of vertices and edges, $E \subset V \times V$. A partition (or clustering) of G into n clusters (n-partition) is then formally defined as $\Pi_G = \bigcup_{i=1}^{n} G_i$ with $G_i = (V_i, E_i)$, and $\forall v \in V\ \exists_1 k \in [1, n], v \in V_k$. Edges E_i are edges that leave or remain inside the partition, $E_i = \{\langle v_1, v_2 \rangle \in E : v_1 \in V_i \wedge v_2 \in V\}$.

The MQ value is the sum of the cluster factors CF_i over all $i \in \{1, \ldots, k\}$ clusters. The cluster factor of the i-th cluster is defined as the normalized ratio between the weight of all the edges within the cluster, intraedges μ_i, and the sum of weights of all edges that connect with nodes in one of the other clusters, interedges $\epsilon_{i,j}$ or $\epsilon_{j,i}$. Penalty of interedges is equally distributed to each of the affected clusters i and j:

$$MQ = \sum_{i=1}^{k} CF_i, \quad CF_i = \begin{cases} 0, & \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \frac{1}{2} \sum_{\substack{j=1 \\ j \neq i}}^{k} (\epsilon_{i,j} + \epsilon_{j,i})}, & \text{otherwise} \end{cases}$$

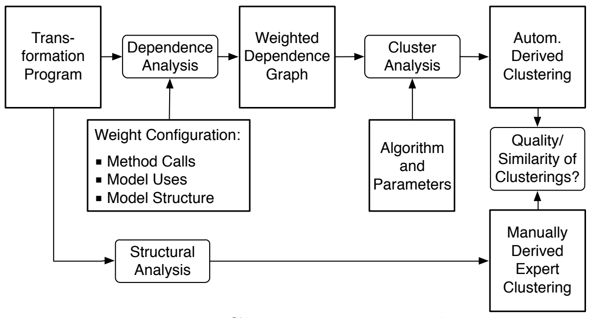


Fig. 2: Clustering approach

Bunch does not differentiate between types of nodes, although edges can be given different weights. Other software clustering approaches exist, a survey by Maqbool et al. [7] lists ARCH, ACDC [8], LIMBO, and others. We decided for Bunch, because it uses classic low-coupling and high-cohesion heuristics that match the information-hiding property we are heading for, and because it has gained a good reputation so far [7].

4 Clustering Model Transformations

The methodology of our automatic clustering approach for model transformations follows to a wide extent the typical procedure of software clustering approaches in general. It comprises three steps (Fig. 2). In the first step, dependence information is statically analyzed and extracted from the source files, resulting in a weighted dependence graph. It is crucial to choose appropriate weights for the types of dependencies that are going to be extracted. The graph serves as input for the cluster analysis. Before running cluster analysis as the second step, an appropriate algorithm must be chosen, and the algorithm's parameters are to be configured. In the third and last step, the automatically derived clustering has to be analyzed. One option is to compare results with the existing modular decomposition that is automatically extractable from the source files, for instance using some of the available similarity measures. However, developers may also compare clusterings derived with alternative weights, either manually, or using similarity or quality metrics. This whole procedure can be repeated with different configurations. Developers planning to refactor the present code manually to obtain an improved modular structure can base their decisions on the computed clusterings.

In the following sub sections, we will address any of the peculiarities when dealing with model transformations. The Activity2Process scenario from Section 2 serves as a running example.

4.1 Dependence Analysis

A preliminary step in any graph-based clustering approach is to extract dependence information from software systems in a graph-based form. When dealing with general-purpose programming languages, various source code analysis tools are available to choose from. However, as we want to extract dependencies from languages specific to the domain of model transformations, we must build our own tools. We use static analysis, i.e., only information that is immediately available at the syntactic level is used, whereas dynamic information that results from (partial) execution of the source code is not used. In the context of transformation programs, we consider not only dependencies among methods, but in addition the structure of involved models and model use dependencies.

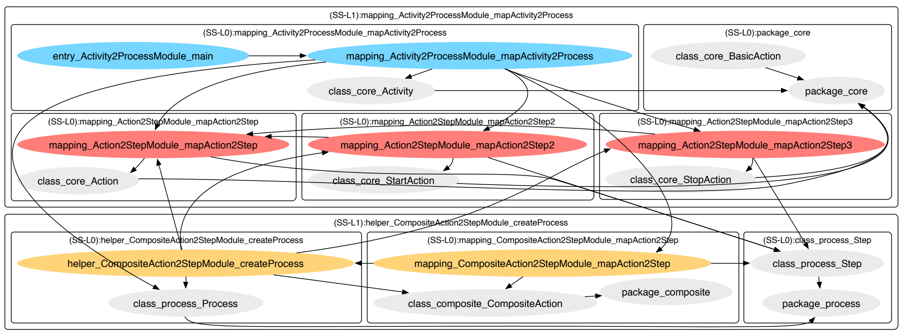

(SS-L1):mapping_Activity2ProcessModule_mapActivity2Process
(SS-L0):mapping_Activity2ProcessModule_mapActivity2Process
entry_Activity2ProcessModule_main
mapping_Activity2ProcessModule_mapActivity2Process
class_core_Activity
(SS-L0):package_core
class_core_BasicAction
package_core
(SS-L0):mapping_Action2StepModule_mapAction2Step
mapping_Action2StepModule_mapAction2Step
class_core_Action
(SS-L0):mapping_Action2StepModule_mapAction2Step2
mapping_Action2StepModule_mapAction2Step2
class_core_StartAction
(SS-L0):mapping_Action2StepModule_mapAction2Step3
mapping_Action2StepModule_mapAction2Step3
class_core_StopAction
(SS-L1):helper_CompositeAction2StepModule_createProcess
(SS-L0):helper_CompositeAction2StepModule_createProcess
helper_CompositeAction2StepModule_createProcess
class_process_Process
(SS-L0):mapping_CompositeAction2StepModule_mapAction2Step
mapping_CompositeAction2StepModule_mapAction2Step
class_composite_CompositeAction
package_composite
(SS-L0):class_process_Step
class_process_Step
package_process

Fig. 3: Activity2Process transformation – Bunch-derived clustering based on class-level dependencies

Implementation structure. Any method that is present in one of the source files is represented by a single node $v_i \in V$ in the graph $G = (V, E)$. For instance, QVT-O defines four different types of methods, namely helpers, mappings, queries, and constructors; these are all translated to nodes in the graph.

Method call dependencies are extracted as follows. For any two nodes $v_i, v_j \in V$ in the graph where each represents a distinct method, $v_i \neq v_j$, a directed edge points from v_i to v_j, $\langle v_i, v_j \rangle \in E$, iff the method represented by v_i calls or otherwise references the method represented by v_j. In QVT-O, a single call (indicated by keyword `map`) may refer to multiple methods in the case of method dispatching, and references may arise from reuse dependencies (keywords are `disjunct`, `merge`, `override`, and `extend`).

Model structure. Any package and class in one of the models used by the transformation is represented by a distinct node in the graph.

Package containment is extracted as follows. For any two nodes $v_i, v_j \in V$ in the graph where each represents a distinct model element, $v_i \neq v_j$, a directed edge points from v_i to v_j, $\langle v_i, v_j \rangle \in E$, iff v_i represents a class or package and v_j represents a package that directly contains that class or package.

Additionally, inheritance and reference relationships among classes are defined. For any two nodes $v_i, v_j \in V$ in the graph where each represents a class, $v_i \neq v_j$, a directed edge points from v_i to v_j, $\langle v_i, v_j \rangle \in E$, iff v_i represents a class that inherits from or references instances of another class represented by v_j.

Model use dependencies. For any two nodes $v_i, v_j \in V$ in the graph where v_i represents a method and v_j a class or package, $v_i \neq v_j$, a directed edge points from v_i to v_j, $\langle v_i, v_j \rangle \in E$, iff the method represented by v_i implicitly or explicitly refers to one of the classes or packages of the involved models. We distinguish model use dependencies with read-access and write-access.

In QVT-O, read dependencies occur as both context and in/inout parameters, or within the implementation body for each of the intermediate *Object Constraint Language* (OCL) expression's inferrable type; write dependencies occur in the form of a mapping's result parameter and explicit instantiations via new or object operator. We provide an alternative extraction method that reduces class-level dependencies to package-level dependencies.

Weight configuration. To guide the clustering algorithm, the influence of dependence relations can be regulated manually. For this purpose, a weighting function $w : E \to \mathbb{N}_0$ assigns positive numbers to the edges in the graph. Depending on the type of dependency represented by the respective edge, we use four weights:

W_{write} for write-access dependencies to classes and packages, W_{read} for read-access dependencies to classes and packages from one of the method's parameters, W_{call} for method call dependencies, and $W_{package}$ for containment of classes and packages to their directly containing package.These weights constitute a particular weight configuration, vector $WC := \langle W_{write}, W_{read}, W_{call}, W_{package} \rangle \in \mathbb{N}_0^4$.

Choosing a weight of zero naturally results in the respective type of edge being ignored by the clustering algorithm. Choosing values $W_{write} \gg W_{read}$ promotes a mainly target-driven decomposition, whereas values $W_{write} \ll W_{read}$ enforce a mainly source-driven decomposition.

4.2 Cluster Analysis

Once dependence information has been extracted from the source files in form of a graph, and weights have been configured accordingly, cluster analysis can be performed on the obtained graph structure in a follow-up step.

Algorithm and parameters. Bunch supports three clustering algorithms, exhaustive search, hill climbing, and a genetic algorithm. In this paper, we use Bunch's hill climbing algorithm which appeared to produce more stable results. We use a consistent configuration, with population size set to 100, the minimum search space set to 90%, leaving 10% of the neighbors selected randomly.

Fig. 3 depicts the graph that had been extracted from the Activity2Process example. Colored nodes represent the transformation's methods, and gray nodes mark the transformation's model elements. Boxes mark a two-level partitioning created by Bunch – L0 stands for the lower and more detailed level, whereas L1 partitions subsume one or more L0 partitions. For this clustering, a weight configuration $\langle 1, 15, 5, 15 \rangle$ had been used. With a sufficiently higher weight for read than for write dependencies, $15 \gg 1$, a source-driven decomposition had been performed. Therefore, mapping methods have been grouped together with their respective source model elements (`Activity`, `Action`, etc.) on L0. Two of the clusters solely contain model elements and can be ignored. In the L1 partition, two clusters remain: One cluster aggregates `Activity2Process` and `Action2Step` methods, the other cluster aggregates `CompositeAction2Step` methods. The reference to class `CompositeAction` may have primarily induced the algorithm to correctly group the respective methods together. When comparing the Bunch-derived L0 partition with our handmade partitioning illustrated by colors blue, red and yellow (cf. Fig. 3), we can observe that both partitions are highly similar. Bunch, however, decided to agglomerate the red and blue L0 clusters to a single L1 cluster. Developers may think about adopting Bunch's recommendation and merge clusters `Activity2Process` and `Action2Step`.

4.3 Structural Analysis

The main objective of the approach is to gain a better understanding of the code, but also to agree on a modular decomposition that fosters understandability and that can be used to restructure the code. To achieve this goal, in this last step, the existing modular structure and partitions computed by the algorithm on different parameters are compared against each other regarding their modularization quality and structural differences. Although this is a manual step that requires to find a compromise on two or more partitions and to refine the solution based on expert knowledge, developers can profit from a set of metrics.

To include the legacy modular structure of the code into the assessment, an automatized structural analysis is used that extracts this kind of information.

Table 1: Activity2Process – Manual vs. derived clustering

Configuration	**Statistics**		**Similarity to expert clustering**			
	# Clusters	MQ index	Precision	Recall	EdgeSim	MeCl
Expert clustering						
Derived manually	3	1.067	100%	100%	100	100%
Method-call dependencies only						
Hill Climbing, $WC = \langle 0,0,1,0 \rangle$	2	1.214	20.00%	100%	54.54	60%
Class-level dependencies						
Hill Climbing, $WC = \langle 1,15,5,15 \rangle$	2	1.083	33.33%	100%	72.72	85%

Modularization Quality. Quality metrics can be used for a quick estimation of the quality of a particular partition. In context of the Bunch approach, it makes sense to observe the MQ index that Bunch uses to assess partitions when searching for a (quasi-)optimal partition. The MQ value can be computed for both method and model dependencies (which it has been optimized for), but also for method dependencies alone.

We use three similarity measures to quantify the similarity of a sample clustering with the expert clustering, Precision/Recall, EdgeSim, and MeCl. The latter two had been specifically built for the software domain by Mitchell et al., all three are supported by the Bunch tool. Other measurements that are used in other contexts include MojoFM [9] and the Koschke-Eisenbarth metric [10].

Precision/Recall. Precision is calculated as the percentage of node pairs in a single cluster of a sample clustering that are also contained within a single cluster in the authoritative clustering. Recall, on the other hand, is defined as the percentage of node pairs within a single cluster in the authoritative clustering that are also node pairs within a single cluster in the sample clustering [3]. Edges are not considered, and the metric is sensitive to number and size of clusters [11].

EdgeSim. The EdgeSim similarity measure [11] calculates the normalized ratio of intra and intercluster edges present in both partitions. Nodes are ignored.

MeCl. The MergeClumps (MeCl) metric is a distance measure [11]. Starting with the largest subsets of entities that had been placed in each of the partitions into the same clusters, a series of merge operations, needed to convert one partition into the other, is calculated. Both directions are considered, and the largest number of merge operations (in a normalized form) is taken as the MeCl distance.

We used the above measurements to compare quality and similarity of manually and two automatically derived partitions in the Activity2Process example. We computed a partition based on method-level dependencies alone, and another partition based on method and class-level dependencies (Tab. 1). Due to the small number of nodes in the input graphs, the output partition per dependence graph produced was identical for five independent runs.

The expert clustering – the one manually done – comprises three clusters, whereas both derived clusterings comprise two. The method-level clustering produced the best MQ value. Despite having a slightly worse modularization quality, the partition derived from class-level dependencies still produces an (albeit marginally) better MQ value than that of the expert clustering.

Even more importantly, for this example, all three metrics agree that model-use dependencies result in a partition more similar to the expert clustering than a partition derived from method-call dependencies alone. The still relatively low precision of 20% and 33.33% can be attributed to the fact that two clusters correspond to a single one in the derived clustering.

5 Related Work

There is only work on statically analyzing model transformation programs for visualization purposes, whereas software cluster analysis has not been applied to model transformation programs in particular.

Model transformation analysis. Some work has been done on extracting dependence information from model transformation programs for graphical viewing. Van Amstel et al. [12] extracted method call and model use dependence information and used hierarchical edge bundling diagrams for presentation. Similar work had been done by us to support model transformation maintenance, as we employ navigable node link diagrams that are embedded into the development environment. Our view encompasses both method call and model use dependencies, including inheritance, reference, and containment relationships among classes and packages. Automatic clustering of these graphs obtained from static analysis, however, has not been carried out so far by either work.

Software cluster analysis. Software clustering approaches mainly focus on recovering an architecture from code written in general-purpose programming languages. Hence their view consists of procedures and call relationships, modules and use dependencies, or classes and their relationships.

Other information to discover a modular structure had been put into consideration as well, including the change history [13], omnipresent objects [14], or transactions (repeated use of a set of classes by other classes indicates that they form a single purpose) [15]. Furthermore, a combination of control and data dependencies as a source of information to discover a hidden modular structure in procedural and object-oriented code had been studied over the last three decades [16,17,18,19]. We apply a similar technique to model transformation languages, though in our specific case we additionally exploit the subtleties of UML/MOF-compliant modeling languages as data description language, for instance hierarchically structured data elements.

Nevertheless, when it comes to the application of automatic clustering techniques to model transformation programs, no previous work is known to us.

6 Conclusions and Outlook

Together with models, model transformations belong to the core assets of software developed according to the model-driven paradigm. Much of the recent work in this area has focused on reuse aspects of transformations, neglecting maintainability as an equally important concern. To manage the inherent complexity of transformation programs, well-approved language concepts can be used, including information hiding modularity. In practice, however, transformation programs lack structure, or their structure has slowly eroded over time.

This work proposes to transfer software clustering techniques to the specific domain of model transformation programs. Based on automatically derived clusterings, developers have to spent less effort in understanding, maintaining and refactoring the code. As the example demonstrates, we were able to automatically derive clusterings that exhibit high similarity with manual decompositions. To reach this goal, we had to integrate structural information of the models and model use dependencies of the transformation language's concepts, and we had to guide the clustering algorithm by weighting the input dependencies to match the type of transformation at hand.

We are currently working on a case study with a larger, more realistic transformation, with promising results so far. However, quality of the results obtained highly depends on the weight vector which is still configured manually. It would be interesting to explore methods to determine this vector automatically. Further on, more details could be used to guide the clustering process. We currently extract data dependence information at the type-level, whereas dataflow analysis could help to detect cohesiveness between methods more accurately.

Acknowledgements. This research has been funded by the German Research Foundation (DFG) under the Priority Programme SPP 1593.

References

1. Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H., Heldal, R.: Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In: MODELS '13. LNCS, Springer (2013) 1–17
2. Rentschler, A., Werle, D., Noorshams, Q., Happe, L., Reussner, R.: Designing Information Hiding Modularity for Model Transformation Languages. In: Proc. 13th Int'l Conf. on Modularity (AOSD'14), ACM (2014) 217–228
3. Shtern, M., Tzerpos, V.: Clustering Methodologies for Software Engineering. Adv. Soft. Eng. (2012)
4. Mitchell, B.S., Mancoridis, S.: On the Automatic Modularization of Software Systems Using the Bunch Tool. IEEE Trans. Software Eng. **32**(3) (2006) 193–208
5. Object Management Group: MOF 2.0 Query/View/Transformation, version 1.1. URL `www.omg.org/spec/QVT/1.1/` (2011)
6. Lawley, M., Duddy, K., Gerber, A., Raymond, K.: Language Features for Re-use and Maintainability of MDA Transformations. In: Proc. OOPSLA Wksp. on Best Practices for Model-Driven Software Development. (2004)
7. Maqbool, O., Babri, H.A.: Hierarchical Clustering for Software Architecture Recovery. IEEE Trans. Software Eng. **33**(11) (2007) 759–780
8. Tzerpos, V.: Comprehension-driven Software Clustering. PhD thesis, Univ. of Toronto (2001)
9. Wen, Z., Tzerpos, V.: An Effectiveness Measure for Software Clustering Algorithms. In: Proc. 12th Int'l Wksp. on Prg. Compr. (IWPC'04), IEEE (2004) 194–203
10. Koschke, R., Eisenbarth, T.: A Framework for Experimental Evaluation of Clustering Techniques. In: Proc. Int'l Wksp. on Prg. Compr. (IWPC '00), IEEE (2000) 201–210
11. Mitchell, B.S., Mancoridis, S.: Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements. In: Proc. IEEE Int'l Conf. on Sw. Maint. (ICSM '01), IEEE (2001) 744–753
12. van Amstel, M., van den Brand, M.G.J.: Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In: Proc. 4th Int'l Conf. on Model Transformations (ICMT '11). LNCS, Springer (2011) 108–122
13. Beyer, D., Noack, A.: Clustering Software Artifacts Based on Frequent Common Changes. In: Proc. Int'l Wksp. on Prg. Compr. (IWPC '05), IEEE (2005) 259–268
14. Wen, Z., Tzerpos, V.: Software Clustering based on Omnipresent Object Detection. In: Proc. 13th Int'l Wksp. on Prg. Compr. (IWPC '05), IEEE (2005) 269–278
15. Sindhgatta, R., Pooloth, K.: Identifying Software Decompositions by Applying Transaction Clustering on Source Code. In: Proc. 31st Annual Int'l Computer Software and Applications Conference (COMPSAC '07), IEEE (2007) 317–326
16. Hutchens, D., Basili, V.: System Structure Analysis: Clustering with Data Bindings. IEEE Trans. Software Eng. **SE-11**(8) (1985) 749–757
17. Liu, S.S., Wilde, N.: Identifying Objects in a Conventional Procedural Language: An example of data design recovery. In: ICSM '90, IEEE (1990) 266–271
18. Chu, W., Patel, S.: Software Restructuring by Enforcing Localization and Information Hiding. In: Proc. 18th Int'l Conf. on Sw. Maint. (ICSM '92), IEEE (1992)
19. Siff, M., Reps, T.W.: Identifying Modules via Concept Analysis. IEEE Trans. Software Eng. **25**(6) (1999) 749–768

Unit Testing of Model to Text Transformations

Alessandro Tiso, Gianna Reggio, Maurizio Leotta

DIBRIS – Università di Genova, Italy
alessandro.tiso | gianna.reggio | maurizio.leotta@unige.it

Abstract. Assuring the quality of Model Transformations, the core of Model Driven Development, requires to solve novel and challenging problems. Indeed, testing a model transformation could require, for instance, to deal with a complex software artifact, which takes as input UML models describing Java applications and produces the executable source code for those applications. In this context, even creating the test cases input and checking the correctness of the output provided by the model transformation are not easy tasks. In this work, we focus on Model to Text Transformations and propose a general approach for testing them at the unit level. A case study concerning the unit testing of a transformation from UML models of ontologies into OWL code is also presented.

1 Introduction

Model to Text Transformations (shortly M2TTs) are not only used in the last steps of a complete model driven software development process to produce the code and the configuration files defining a new software system, but can be the way to allow any kind of user to perform tasks of different nature using visual models instead of the scarcely readable text required by software tools (e.g. the textual input for a simulator of business processes may be generated by a UML model made of class and activity diagrams).

Testing model transformations is a complex task [6], since the complexity of the inputs and the time required to produce them (e.g. complete UML models instead of numbers and strings), and the difficulties in building an effective oracle (e.g. it may have to provide whole Java programs instead of a result of such programs). Selecting input models for model transformations testing is harder than selecting input for programs testing because they are more difficult to be defined in an effective way. Hence, also giving adequacy criteria for model transformations is again more difficult than in the case of programs. Furthermore, also the well-established kinds of tests, e.g. acceptance and system, are either meaningless for model transformations or have to be redefined.

M2TTs may be quite complex and large. For example, a transformation from UML models composed of class diagrams and state machines, where the behaviour of any element is fully defined by actions and constraints, to running Java applications built using several frameworks (e.g. Hibernate, Spring, JPA). Thus, it is important to be able to perform tests also on subparts of an M2TT. First, we need to identify the nature of such subparts, and then define what testing them means. Using the classical terminology, we need to understand whether a form of unit testing is possible for M2TTs.

In this paper we present a proposal for testing M2TTs at the unit level. We then indicate how to implement the executions of the introduced unit tests for M2TTs coded using Acceleo [1], and how this form of unit testing may be integrated in the development method for model to text transformation *MeDMoT* proposed by some of the authors in [10, 11, 12].

We use the M2TT U-OWL as running example to illustrate our proposal. It is a transformation from profiled UML models of ontologies to OWL definitions of ontologies. An ontology model is composed by a class diagram (the StaticView) defining the structure of the ontology in terms of categories (plus specializations and associations among them), and possibly by an object diagram (the InstancesView) describing the information about the instances of the ontology (i.e. which are the individuals/particulars of the various categories, the values of their attributes, and the links among them). The U-OWL targets are text files describing an ontology using the RDF/XML for OWL concrete syntax. This transformation has been developed following *MeDMoT*. Specifically, we have given the U-OWL requirements, designed the transformation and then implemented it using Acceleo. It is composed by 8 modules, 21 templates and 8 queries.

The paper is structured as follows. In Sect. 2 we present our proposal for M2TTs unit testing including suggestions for implementing their execution and getting a report, and in Sect. 3 how this kind of testing may be integrated within *MeDMoT*. The results of the unit testing of U-OWL are summarized in Sect. 4. Finally, related work and conclusions are respectively in Sect. 5 and 6.

2 Unit Testing of Model to Text Transformations

2.1 Testing Model to Text Transformations

The *Model to Text Transformations* considered in this paper map models[1] (both graphical and textual) to *Structured Textual Artifacts* (shortly STAs) having a specific form. An STA is a set of text files, written using one or more concrete syntaxes, disposed in a well-defined structure (physical positions of the files in the file system).

In previous works [10, 11, 12] we have proposed new kinds of tests suitable for model to text transformations.

- **Conformance tests** are made with the intent of verifying whether the M2TT results comply the requirements imposed by the target definition. Generally this means that the produced STA has the required structure and form. Considering the U-OWL case this means that the files produced can be loaded into a tool for OWL like Protege[2] without errors (because it accepts only well-formed OWL files).
- **Semantic tests** are made with the intent of verifying whether the target of the M2TT has the expected semantics. In the U-OWL case a semantic test may check, for example, if an OWL class has all the individuals represented in the UML model, or that an OWL class is a sub-class of another one iff such relationship was present in the UML model.
- **Textual tests** are made with the intent of verifying whether the STAs produced by the M2TT have the required form considering both the structuring in folders and files and the textual content of the files.

[1] conform to a meta-model
[2] `http://protege.stanford.edu/`

2.2 Unit Tests for Model to Text Transformations

M2TTs may be quite large and complex. For this reason, they may be composed of several sub-transformations. Each sub-transformation may be in turn composed of other sub-transformations, each of them taking care of transforming some model elements. The various sub-transformations may be arranged in a call-graph (we use a diagram similar to the structure chart, that we call *functional decomposition diagram*), where the nodes are labelled by the sub-transformations themselves and the arcs represent calls between sub-transformations (see, e.g. Fig. 1).

In what follows, we consider only M2TTs equipped with a functional decomposition diagram; moreover, we assume that their design specifications, which provide information on the STAs produced by their sub-transformations, are available.

To perform the unit testing of an M2TT we must identify the units composing it. That will be the subjects of the unit tests. It is natural to choose as subject of unit tests the various sub-transformations of the M2TT, introduced by the functional decomposition diagram, considered in isolation.

In the context of testing classical software, the parts considered by unit tests, in general, cannot be run (e.g. a method or a procedure), thus special stubs have to be built (e.g. JUnit test cases for Java class methods). In general, in the case of M2TTs the "parts" composing a transformation return STA fragments that in many cases do not even correspond to well-formed constructs in the transformation target; to build some stubs for them to be able to perform conformance and semantic testing may be utterly complex and time consuming (think for example what does it mean to build a stub for a configuration file for the Spring framework, or referring to the U-OWL case study, to build a stub for an RDF/XML definition of an individual). To propose a general technique for unit testing of M2TTs, we should consider the sub-transformation results as purely textual artifacts, and so we use only textual tests, which is one of the proposed new kinds of tests (see Sect. 2.1).

To build a unit test for a given sub-transformation, say *ST*, we need other ingredients: the test input and the oracle function.

The *test inputs* for *ST* are model elements. For example, the sub-transformation of U-OWL **TAttribute** (see Fig. 1) transforms a UML class attribute into an OWL fragment

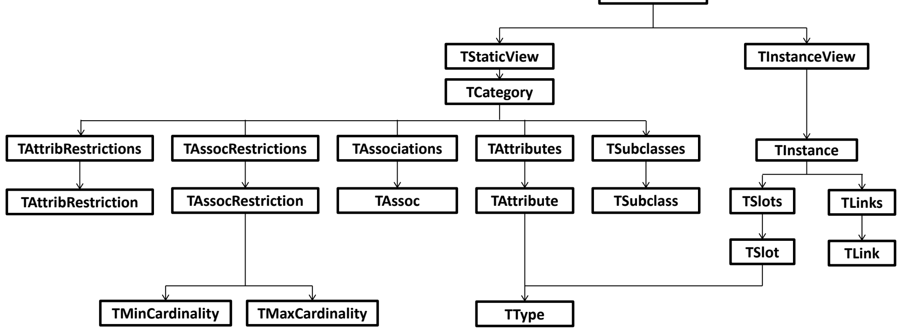


Fig. 1. Functional decomposition diagram for the U-OWL M2TT

(more precisely, in the definition of an OWL data type property). Thus, to test **TAttribute** we need to use as inputs UML attributes.

There are substantially two ways to obtain the model elements to be used in the unit tests: (i) they can be built by instantiating the right meta-class (in the case **TAttribute**, the meta-class in the UML meta-model which represents class attributes) or (ii) they can be extracted from a model using an appropriate query. We opted for (ii) because in this way input models can be easily written by the tester. Furthermore, if the model considered contains more than one model element of the desired type (e.g. class attribute), it is possible to easily improve the test coverage of the input space. Indeed, we can retrieve all these elements in only one query, and thus test the sub-transformation using different values.

The *oracle function* has to evaluate the correctness of the result of *ST* on the chosen input. Usually, this is done by comparing the actual output given by the *ST* with the expected output. In the case of the M2TTs the expected output of a sub-transformation is a fragment of an STA (for example some text files arranged in different folders, a text file, a text fragment). It is clear that generating this kind of expected outputs is very difficult and time consuming if not almost impossible; consider the case of the transformation from UML models to Java applications, the expected outputs may be sets of files containing the complete code of various Java classes or long and cryptic configuration files needed by the used framework (e.g. Hibernate).

Thus, we formulate the oracles in terms of properties on the expected STA fragments. These properties are about the structure of the STA fragment, for example which files and folders must be present and how they are named, and about the text files content. The latter may just require that the output text matches a given regular expression.

Following the classical testing techniques, we define a test case for a sub-transformation as a pair consisting of a specific input and a property on the expected output; we have preferred instead to define generic test cases that are properties on the expected output parameterized on the input of *ST*. To allow the expression of more powerful tests, we then decided that an oracle is composed by one or more conditions on the *ST* parameters that allow to select the proper parameterized regular expression for checking the correctness of the *ST* output.

A test suite is composed by a set of input models and by a list of test specifications (written by the tester), one for each *ST*, such as the one shown in Fig. 2. Obviously, at each test specification corresponds a test case in the test suite implementation. For simplicity in this case, we adopted a simplified regular expression language in which

Sub-Transformation

TAttribute(*attrName:*String,*OwnerClassName:*String)

Verification Pattern

```
<owl:DatatypeProperty rdf:ID = 'attrName'>
  <rdfs:domain rdf:resource = '#OwnerClassName'/>
  <rdfs:range rdf:resource = ***/>
</owl:DatatypeProperty>
```

Fig. 2. Unit Test Specification for **TAttribute**

the text that must be ignored is marked by a string containing three “*” characters (see Fig. 2).

2.3 Unit Tests Implementation

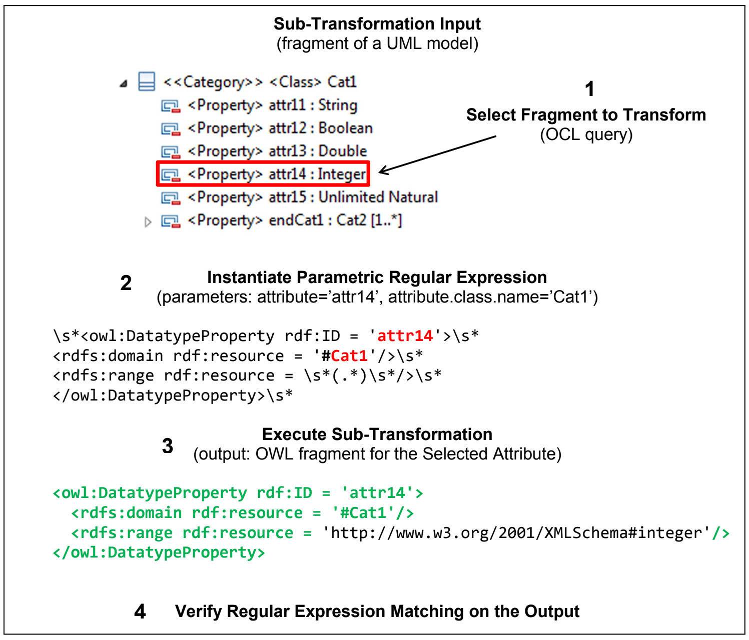
In what follows we assume that the considered M2TTs (and their sub-transformations) are implemented using the *Eclipse Platform* [4], taking advantage of the features offered by the *Eclipse Modeling Project* [3], such as:
– the *Eclipse Modeling Framework* [2] (EMF) that provides tools and runtime support for viewing and editing models, and
– Acceleo [1] that is an implementation of the OMG MOF Model to Text Language [5] plus an IDE and a set of tools to simplify the production of M2TTs.

Each M2TT is implemented as an Eclipse plugin, and it is composed by a set of Acceleo templates, which correspond to the sub-transformations appearing in the functional decomposition diagram.

The test suite is implemented as an Acceleo transformation, built by defining a template for each test case (thus, it is an Eclipse plugin). Obviously, the Eclipse plugin implementing the unit test suite must have a dependency to the plugin which implements the M2TT, so that a test case for *ST* can invoke the template corresponding to *ST*.

Let *TC* be the test case for the sub-transformation *ST*. The template implementing *TC* is defined as follows.

Fig. 3 shows the four logical steps performed by the Acceleo transformation implementing the test suite during its execution. First, an OCL query over the input model extracts the model elements suitable to be used as input for *ST* (e.g. **TAttribute**, see point

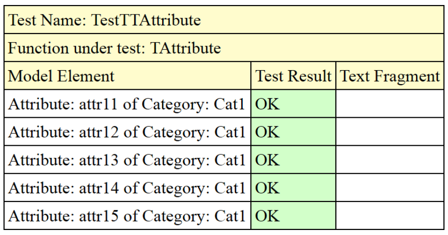

Sub-Transformation Input
(fragment of a UML model)
1
Select Fragment to Transform
(OCL query)
2
Instantiate Parametric Regular Expression
(parameters: attribute='attr14', attribute.class.name='Cat1')
3
Execute Sub-Transformation
(output: OWL fragment for the Selected Attribute)
4
Verify Regular Expression Matching on the Output

```
<<Category>> <Class> Cat1
    <Property> attr11 : String
    <Property> attr12 : Boolean
    <Property> attr13 : Double
    <Property> attr14 : Integer
    <Property> attr15 : Unlimited Natural
    <Property> endCat1 : Cat2 [1..*]
```

```
\s*<owl:DatatypeProperty rdf:ID = 'attr14'>\s*
<rdfs:domain rdf:resource = '#Cat1'/>\s*
<rdfs:range rdf:resource = \s*(.*)\s*/>\s*
</owl:DatatypeProperty>\s*
```

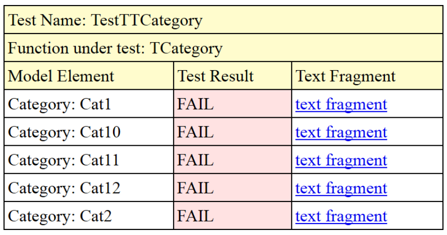
```
<owl:DatatypeProperty rdf:ID = 'attr14'>
  <rdfs:domain rdf:resource = '#Cat1'/>
  <rdfs:range rdf:resource = 'http://www.w3.org/2001/XMLSchema#integer'/>
</owl:DatatypeProperty>
```

Fig. 3. Unit Test Logical Schema



Test Name: TestTTAttribute		
Function under test: TAttribute		
Model Element	Test Result	Text Fragment
Attribute: attr11 of Category: Cat1	OK	
Attribute: attr12 of Category: Cat1	OK	
Attribute: attr13 of Category: Cat1	OK	
Attribute: attr14 of Category: Cat1	OK	
Attribute: attr15 of Category: Cat1	OK	

Fig. 4. HTML Unit Test Report: no failed tests



Test Name: TestTTCategory		
Function under test: TCategory		
Model Element	Test Result	Text Fragment
Category: Cat1	FAIL	text fragment
Category: Cat10	FAIL	text fragment
Category: Cat11	FAIL	text fragment
Category: Cat12	FAIL	text fragment
Category: Cat2	FAIL	text fragment

Fig. 5. HTML Unit Test Report: failed tests

1 in Fig. 3). Then, for each model element, a special kind of Acceleo query, called Java Services wrapper[3], is invoked. This kind of Acceleo query can call a Java method as if it was a simple query. The service wrapper implements the oracle of *TC*. It takes the following parameters: a string obtained by invoking *ST*, that is the actual output of the *ST*, and the same parameters (model elements) used by *ST*. The service wrapper, using the parameters representing the model element, builds the appropriate regular expression (point **2** of Fig. 3, the parametric portion of the regular expression is depicted as red text) by instantiating the general parametric regular expression appearing in *TC* (see, e.g. Fig. 2). Then, the service wrapper, using the instantiated regular expression, checks if the output of *ST* (point **3**) is correct (point **4**).

Fig. 3 bottom, shows the actual output of the **TAttribute** sub-transformation and the green text represents the correct match of the regular expression shown above while black text represents text that is not checked (e.g. in the considered case study, the type of the attributes is managed by the **TType** sub-transformation and thus is not checked during the unit testing of the **TAttribute** sub-transformation).

The result of the oracle function is then used to build a HTML report as it is shown in Fig. 4 and 5. In detail, Fig. 4 shows a report of a unit test over a *ST* successful on all the elements, meanwhile Fig. 5 shows a report of a unit test failing on some elements. In the latter case on the rightmost column there is a hyper-link to the text fragment generated by the sub-transformation under test. Thus, the developer can inspect the problematic output.

Summarizing, each unit test, which has as subject one of the sub-transformations: (1) selects the model elements composing the test input using appropriate queries over the input models, and for each model element (2) calls the oracle function that, invokes the sub-transformation under test obtaining the actual output that is verified using a regular expression built in conformance to what is defined by the M2TT design specification for the sub-transformation parameterized with the actual values of the sub-transformation parameters. As last step, using the oracle function result, we produce a report of the unit test, in a form suitable to be easily readable from the transformation developer, such as a HTML report.

Obviously, the plugin containing the test suite can be executed using different models as input.[4]

[3] see `http://www.obeonetwork.com/page/acceleo-user-guide`
[4] It can be done by configuring in an appropriate way Eclipse run-configurations.

3 Model to Text Transformations Unit Testing within *MeDMoT*

In the previous section we described a general approach for testing at the unit level an M2TT. This approach can be integrated within *MeDMoT*, our general method for developing M2TTs. Each instance of *MeDMoT* is a method to develop a model to text transformations (e.g. U-OWL). It is briefly described in [11, 12], and more details can be found in the Ph.D. thesis of one of the authors [10]. Here, we summarize the main concepts and phases of *MeDMoT*.

The input of a transformation (e.g. U-OWL) is a set of UML models conform to the UML meta-model extended with a specific profile (e.g. a profile for UML models of ontologies). The form of the source models is defined by a conceptual meta-model, i.e. a UML class diagram with a class whose instances correspond to all the possible source models. That meta-model is constrained by a set of well-formedness rules that precisely define the set of acceptable models. The target of a transformation is always an STA having a specific form (e.g. for U-OWL a set of text files describing an ontology).

In *MeDMoT* an M2TT is structured as a chain of transformations of different types, some from model to model and one from model to text. The input model is verified by a model to model transformation that checks if the input model satisfies the well-formedness rules constraining the transformation source (i.e. the input model). If the input model is well-formed, then it is refactored by one or more model to model transformations in an equivalent simplified model. This step helps to maintain the subsequent and last M2TT as simple as possible, for instance, by reducing the number of different constructs used in the model. Finally, the refactored model is transformed into an STA using an M2TT. We call the last transformation of this chain the Text Generation Transformation (shortly TGT), which is precisely the M2TT on which we perform the unit tests.

The design of each sub-transformation of the TGT is specified by means of relevant source-target pairs. Each pair is composed by a left side, that shows a template for the sub-transformation inputs, and a right side that shows the result of the application of this sub-transformation on model elements obtained instantiating such template, see e.g. Fig. 6.

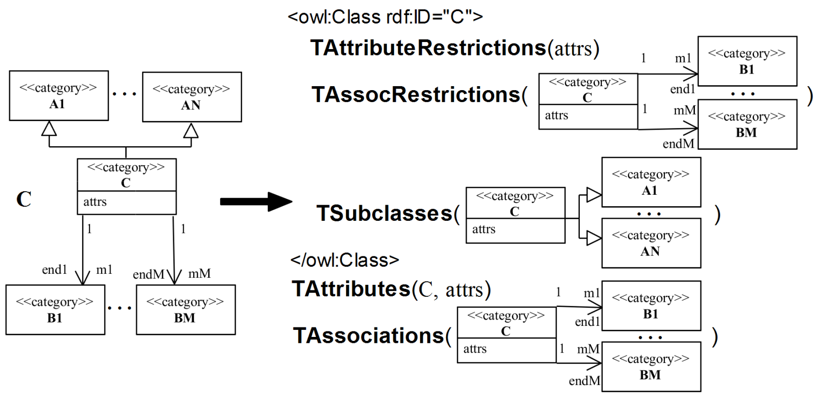


Fig. 6. Example of source-target pair defining the **TCategory** sub-transformation

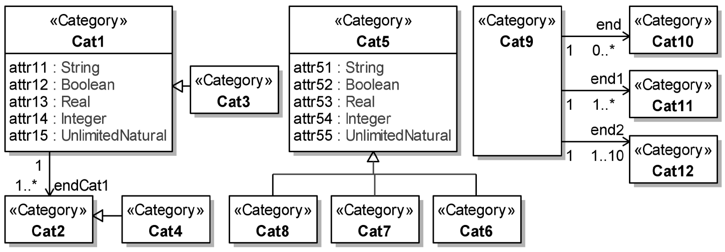


Fig. 7. Model 1 used to test U-OWL

Model	Classes	Attributes	Gen-Specs	Associations	Objects	Slots	Links
Model 1	12	10	5	4	0	0	0
Model 2	7	3	3	3	0	0	0
Model 3	2	10	0	2	4	20	4

Table 1. Simple Metrics of Test Models

To select the set of model elements to be used as input in our unit tests, we use an adequacy criteria defined as follows: at least an instance of the various templates of the M2TT design and at least an occurrence of all UML constructs used to build the source model must appear in the input models[5].

For example the model shown in Fig. 7 contains an instance of the pattern shown in the definition of **TCategory** (see Fig. 6). As we can see, class attributes are instantiated in the model with all the permitted primitive types, the multiplicities of the association ending roles (m1, mM) are instantiated in the model using different values such as 0..*, 1..* and 1..10.

4 Unit Testing of U-OWL

To perform the unit test of U-OWL we used three models, created in order to satisfy the adequacy criteria described in Sect. 3. One of these input models is shown in Fig. 7. The figures in Tab. 1 allow to grasp the size of the three models.

A summary of the results of the execution of all the unit tests (i.e. for the three models) is shown in Tab. 2. For each model we have reported the number of test cases executed on each sub-transformation and how many have failed. There are failed tests for the following sub-transformations: **TType**, **TOntology**, **TCategory**, **TAttribRestriction** and **TAssociation**. In the majority of the cases, all the tests have failed; this is due to the fact that some sub-transformations erroneously transform any possible input.

For the failed tests, by navigating the hyper link shown in the rightmost column of the HTML report (see Fig. 5) we can examine the output of the sub-transformations not satisfying the condition expressed by the tests, and thus we were facilitated to understand what the problem in their definitions is. **TType** does not produce any output when the type of an attribute is *double*. A quick inspection of the code implementing **TType** allows us to discover the error, **TType** does not transform *double*. In the same way, we are able to discover the reasons of the failure of the other tests. For instance, in the case of **TCategory**, the `owl:Class` tag is not closed in the proper way, while in the case

[5] This is a slightly different version of the criteria defined in our previous work [12].

	Model 1		Model 2		Model 3	
ST Under Test	Num. Tests	Num. failed	Num. Tests	Num. failed	Num. Tests	Num. failed
TType	10	2	5	1	30	6
TOntology	1	1	1	1	1	1
TCategory	12	12	7	7	2	2
TAttribRestriction	10	10	5	5	10	10
TAssocRestriction	4	0	3	0	2	0
TMinCardinality	4	0	3	0	2	0
TMaxCardinality	4	0	3	0	2	0
TAttribute	10	0	5	0	10	0
TAssociation	4	4	3	0	2	2
TSubClass	5	0	3	0	-	-
TInstance	-	-	-	-	5	0
TSlot	-	-	-	-	20	0
TLink	-	-	-	-	4	0

Table 2. Tests and Errors

of **TOntology** one of the name space declaration contains an URI different from the expected one. Finally, in the case of **TAssociation** the domain and range references are inverted in the generated OWL code.

Some of these errors cannot be revealed by the other kind of tests (such as the conformance and semantic test). Indeed, the error revealed by the unit test on **TOntology** cannot be revealed by any other kind of test (given that name space can be defined using any URI), meanwhile the error revealed by the unit test on **TAssociation** can be revealed only by a semantic test (given that, this kind of error produce a correct RDF/XML specification of an OWL object property, but it is not semantically compliant with the input model). All the material related (e.g. input models and HTML reports) can be found at `http://sepl.dibris.unige.it/2014-OWLTest.php`.

5 Related Work

Wimmer et al. [13] propose a technique to test model to text transformations based on tracts, that requires to transform an M2TT into a model to model one, by transforming what we have called STAs into models defined by a specific meta-model with meta-classes corresponding to folders, text files and their lines. The tests derived by tracts use the OCL extended with a "grep" function to define their oracles. The intent of this approach is quite similar to our since it consists in checking that the text produced by the M2TT has the required form. Moreover, tracts defined for sub-transformations may be used to build unit tests as those considered by our approach.

García-Domínguez et al. [7] present an approach for testing "model management tasks" within the Epsilon platform based on the unit testing framework EUnit; differently from our proposal, [7] does not consider model to text transformations, and, despite the name, does not seem to introduce any kind of "unit testing" not even for model to model transformations.

At the best of our knowledge there are no other works which deal specifically with M2TT testing or unit level model transformation testing (except our previous work [11]).

Esther Guerra in her work [8] considers model to model transformations and starting from a formal specification written using a custom specification language can derive oracle functions and generate a set of input test models that can be used to test the model transformation written using transML [9], a family of modelling languages proposed by the same author and others.

6 Conclusion

In this paper, we have presented an approach for the unit testing of model to text transformations, and up to our knowledge no other ones have been already proposed. The only requirement of our approach is that the transformation must be provided of: (1) a functional decomposition diagram showing the various sub-transformations composing it, and (2) a design specification for each sub-transformation reporting information on the textual artifacts it produces. Moreover, we have also presented a case study concerning the unit testing of a transformation from UML models of ontologies into OWL code performed following the proposed method.

As future work we plan to validate the power of the proposed unit testing approach for model to text transformations, together with the other kind of tests introduced in previous works, such as semantic and conformance (see [12]), by means of empirical experiments, for example based on fault-injection, and by applying them to other case studies concerning transformations larger and more complex than U-OWL.

References

1. Acceleo. `http://www.eclipse.org/acceleo/`.
2. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.
3. Eclipse Modeling Project. `http://www.eclipse.org/modeling/`.
4. Eclipse platform. `http://www.eclipse.org/`.
5. OMG MOF model to text transformation language (MOFM2T). `http://www.omg.org/spec/MOFM2T/1.0/`.
6. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.
7. A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A unit testing framework for model management tasks. In *Proceedings of 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS 2011, pages 395–409. Springer, 2011.
8. E. Guerra. Specification-driven test generation for model transformations. In Z. Hu and J. Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *LNCS*, pages 40–55. Springer Berlin Heidelberg, 2012.
9. E. Guerra, J. Lara, D. Kolovos, R. Paige, and O. Santos. Engineering model transformations with transml. *Software & Systems Modeling*, 12(3):555–577, 2013.
10. A. Tiso. *MeDMoT: a Method for Developing Model to Text Transformations*. PhD thesis, University of Genova, 2014.
11. A. Tiso, G. Reggio, and M. Leotta. Early experiences on model transformation testing. In *Proceedings of 1st Workshop on the Analysis of Model Transformations (AMT 2012)*, pages 15–20. ACM, 2012.
12. A. Tiso, G. Reggio, and M. Leotta. A method for testing model to text transformations. In *Proceedings of 2nd Workshop on the Analysis of Model Transformations (AMT 2013)*, volume 1077. CEUR Workshop Proceedings, 2013.
13. M. Wimmer and L. Burgueño. Testing M2T/T2M transformations. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, editors, *Proceedings of 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2013)*, volume 8107 of *LNCS*, pages 203–219. Springer, 2013.

On Static and Dynamic Analysis of UML and OCL Transformation Models

Martin Gogolla, Lars Hamann, Frank Hilken

Database Systems Group, University of Bremen, Germany
{gogolla|lhamann|fhilken}@informatik.uni-bremen.de

Abstract. This contribution discusses model transformations in the form of transformation models that connect a source and a target metamodel. The transformation model is statically analyzed within a UML and OCL tool by giving each constraint an individual representation in the underlying class diagram by highlighting the employed model elements. We also discuss how to analyze transformation models dynamically on the basis of a model validator translating UML and OCL into relational logic. One can specify, for example, the transformation source and let the tool compute automatically the transformation target on the basis of the transformation model without the need for implementing the transformation. Properties like injectivity of the transformation can be checked through the construction of example transformation pairs.
Keywords. Transformation model, Metamodel, UML, OCL, Model validator, Static and dynamic transformation model analysis.

1 Introduction

Model transformations are regarded as essential cornerstones for Model-Driven Engineering (MDE). Quality assessment and improvement techniques like transformation validation and verification are thus central for the success of MDE. Therefore, testing and analysis techniques for model transformations [2, 1] are obtaining more and more attention.

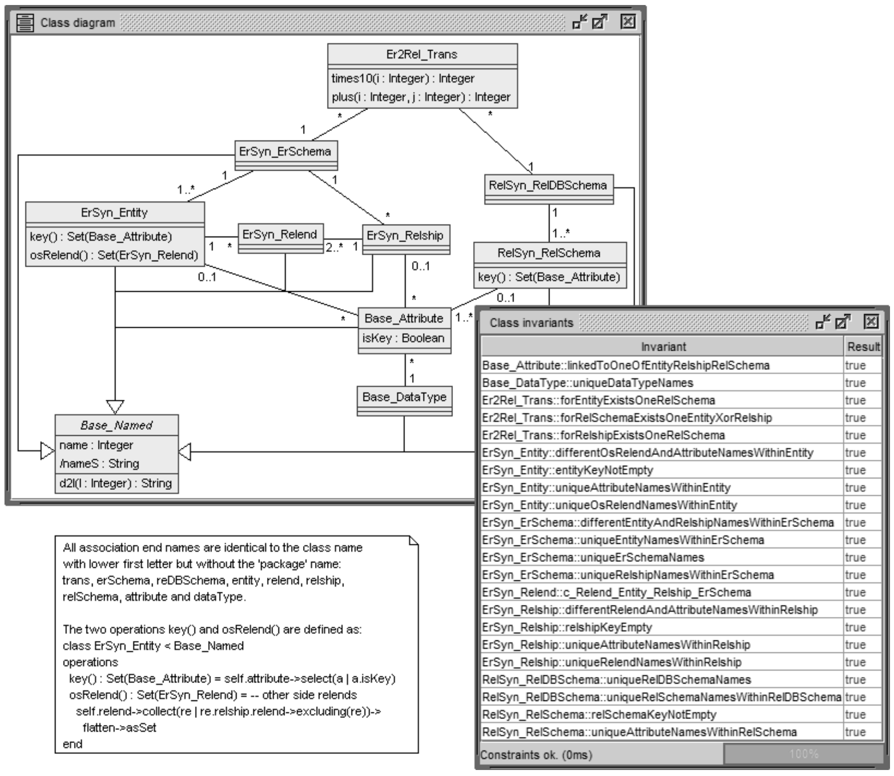
Here, we discuss model transformations in form of transformation models [3]. Transformation models are descriptive characterizations of mappings between a source and target metamodel. Our approach proposes to check the covering of constraints within transformation models statically in order to better understand the model, and to check for the completion of partially specified transformation pairs. We apply a so-called model validator in the tool USE (Uml-based Specification Environment) that searches for instances within a finite search space.

Our work has links to related approaches. Our contribution is based on Alloy [8] and Kodkod [10]. The implementation of the model validator that we employ is grounded on a translation of UML and OCL concepts into relational logic as described in [9]. Transformation models using the same example as here, however with different metamodels and focusing on refinement, have been studied in [4]. A general context of descriptive transformations employing UML and OCL is nicely described in [5]. The same example with focus on different transformation properties (as consistency and metamodel property preservation) and discussing solving and translation times has been studied in [7], but the covering and completion techniques developed here are not treated there.

The rest of this paper is structured as follows. Section 2 describes the running example. Section 3 sketches how to apply the model validator. Section 4 shows how transformation models can be statically inspected. Section 5 applies our technique for dynamically completing partial transformation model pairs. Section 6 closes the paper with a conclusion and future work.

2 Model Transformation Example

The running example in this paper is the well-known transformation between ER and relational database schemata. We study this transformation in form



```
All association end names are identical to the class name
with lower first letter but without the 'package' name:
trans, erSchema, reDBSchema, entity, relend, relship,
relSchema, attribute and dataType.

The two operations key() and osRelend() are defined as:
class ErSyn_Entity < Base_Named
operations
  key() : Set(Base_Attribute) = self.attribute->select(a | a.isKey)
  osRelend() : Set(ErSyn_Relend) = -- other side relends
    self.relend->collect(re | re.relship.relend->excluding(re))->
      flatten->asSet
end
```

Invariant	Result
Base_Attribute::linkedToOneOfEntityRelshipRelSchema	true
Base_DataType::uniqueDataTypeNames	true
Er2Rel_Trans::forEntityExistsOneRelSchema	true
Er2Rel_Trans::forRelSchemaExistsOneEntityXorRelship	true
Er2Rel_Trans::forRelshipExistsOneRelSchema	true
ErSyn_Entity::differentOsRelendAndAttributeNamesWithinEntity	true
ErSyn_Entity::entityKeyNotEmpty	true
ErSyn_Entity::uniqueAttributeNamesWithinEntity	true
ErSyn_Entity::uniqueOsRelendNamesWithinEntity	true
ErSyn_ErSchema::differentEntityAndRelshipNamesWithinErSchema	true
ErSyn_ErSchema::uniqueEntityNamesWithinErSchema	true
ErSyn_ErSchema::uniqueErSchemaNames	true
ErSyn_ErSchema::uniqueRelshipNamesWithinErSchema	true
ErSyn_Relend::c_Relend_Entity_Relship_ErSchema	true
ErSyn_Relship::differentRelendAndAttributeNamesWithinRelship	true
ErSyn_Relship::relshipKeyEmpty	true
ErSyn_Relship::uniqueAttributeNamesWithinRelship	true
ErSyn_Relship::uniqueRelendNamesWithinRelship	true
RelSyn_RelDBSchema::uniqueRelDBSchemaNames	true
RelSyn_RelDBSchema::uniqueRelSchemaNamesWithinRelDBSchema	true
RelSyn_RelSchema::relSchemaKeyNotEmpty	true
RelSyn_RelSchema::uniqueAttributeNamesWithinRelSchema	true

Constraints ok. (0ms) 100%

Fig. 1. Class diagram and invariants for example transformation model.

of a transformation model as introduced in [6, 3]. A transformation model is a descriptive model where the relationship between source and target is purely characterized by the (source,target) model pairs determined by the transformation. A transformation model consists in our approach of a plain UML class diagram with restricting OCL invariants. Typically, there is an anchor class for the source model, an anchor class for the target model, and a connecting class for the transformation. There are OCL invariants for restricting the source metamodel, for the target metamodel, and for the transformation.

In Fig. 1 the class diagram and the invariant names for the example are pictured. All details of the example can be found in [6]. The example transformation model has four parts: a base part with datatypes and attributes for concepts commonly employed in the ER and relational model; a part for ER schemata (`ErSchema`) with the concepts `Entity`, `Relship` (relationship), and `Relend` (relationship end); a part for relational database schemata (`RelDBSchema`) incorporating relational schemata (`RelSchema`); finally, a part for the transformation (`Trans`). [6] discusses also the semantics. Therefore, some classes here are marked in their names as belonging to the syntax (`ErSyn`, `RelSyn`).

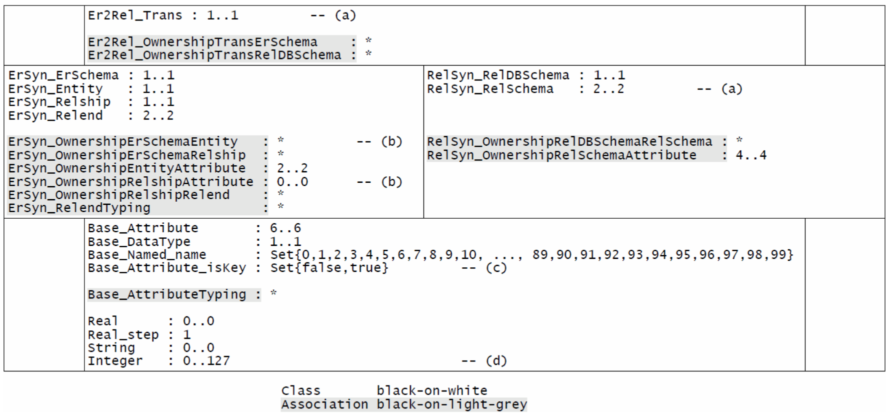
We have used the terms source and target, but transformation models are direction-neutral due to the central employment of associations. We will say that we 'transform a source ER schema into a target relational database schema', but formally the class diagram does not indicate any direction. In our view, transformation models can be looked at as a form of bidirectional transformations.

Currently our model validator does not support the computation of strings in a satisfactory way. In particular, we need string computations for relational attributes in connection with ER attribute names and relationship end names. Through this, we can establish a connection between the source and the target model. Thus, in contrast to [6], we model names (for example, of entities or attributes) as integers and have to pose certain restrictions on the use of the underlying integers and strings. We encode ten letters as digits: A↔0, B↔1, C↔2, D↔3, E↔4, F↔5, G↔6, H↔7, J↔8, K↔9. Through a derived attribute `nameS`, we are able to represent the 'integer names' formally as string values. For example, we will calculate: `20 = 2*10+0 ≅` '2.concat(0)' `≅ 'C'.concat('A') = 'CA'`. This section followed the ideas we have developed in [7].

3 Applying the USE Model Validator

We explain the application of the USE model validator by showing how the tool has to be configured in order to construct a model transformation between an example ER schema and a corresponding relational database schema. The needed configuration is shown in Fig. 2 and the resulting generated object diagram, which captures both schemata, is pictured in Fig. 3.

In a model validator configuration, the population of (a) *classes*, (b) *associations*, (c) *attributes* and (d) *datatypes* is determined. Classes, attributes and datatypes are displayed in the configuration table in black-on-white, and associations in black-on-light-grey. (a) A *class* needs an integer upper bound for the maximal number of objects in that class, and an optional lower bound may be given. (b) *Associations* may also have a lower and upper bound for the number of links or their population may be left open and be thus determined through the (up-



```
Er2Rel_Trans : 1..1          -- (a)

Er2Rel_OwnershipTransErSchema    : *
Er2Rel_OwnershipTransRelDBSchema : *
```

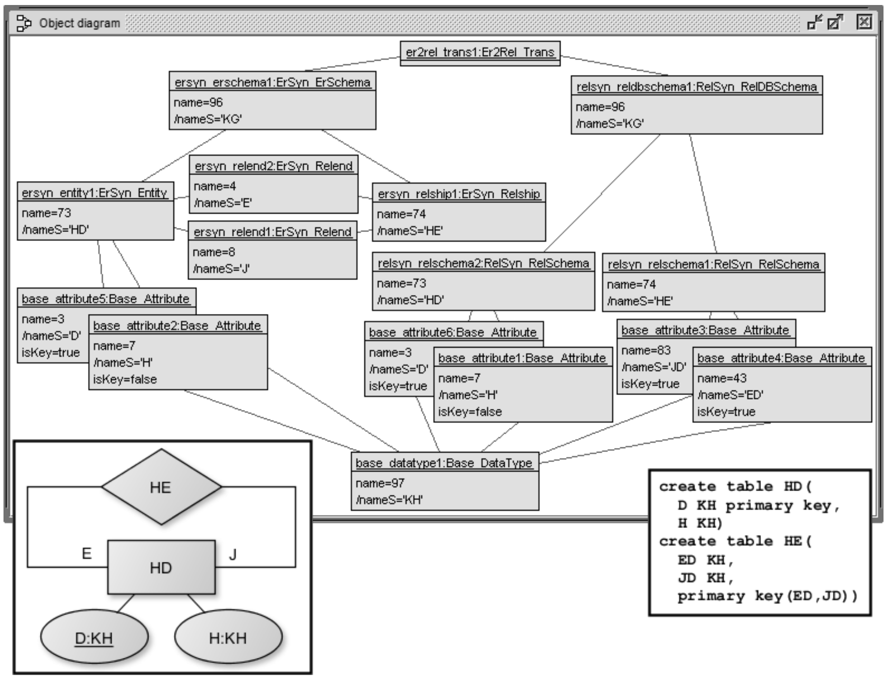
```
ErSyn_ErSchema : 1..1
ErSyn_Entity   : 1..1
ErSyn_Relship  : 1..1
ErSyn_Relend   : 2..2

ErSyn_OwnershipErSchemaEntity    : *          -- (b)
ErSyn_OwnershipErSchemaRelship   : *
ErSyn_OwnershipEntityAttribute   : 2..2
ErSyn_OwnershipRelshipAttribute  : 0..0       -- (b)
ErSyn_OwnershipRelshipRelend     : *
ErSyn_RelendTyping               : *
```

```
RelSyn_RelDBSchema : 1..1
RelSyn_RelSchema   : 2..2         -- (a)

RelSyn_OwnershipRelDBSchemaRelSchema : *
RelSyn_OwnershipRelSchemaAttribute   : 4..4
```

```
Base_Attribute       : 6..6
Base_DataType        : 1..1
Base_Named_name      : Set{0,1,2,3,4,5,6,7,8,9,10, ..., 89,90,91,92,93,94,95,96,97,98,99}
Base_Attribute_isKey : Set{false,true}          -- (c)

Base_AttributeTyping : *

Real      : 0..0
Real_step : 1
String    : 0..0
Integer   : 0..127                              -- (d)
```

```
Class       black-on-white
Association black-on-light-grey
```

Fig. 2. Configuration for ER schema with binary relationship.


Object diagram
er2rel_trans1:Er2Rel_Trans
ersyn_erschema1:ErSyn_ErSchema
name=96
/nameS='KG'
relsyn_reldbschema1:RelSyn_RelDBSchema
name=96
/nameS='KG'
ersyn_relend2:ErSyn_Relend
name=4
/nameS='E'
ersyn_entity1:ErSyn_Entity
name=73
/nameS='HD'
ersyn_relship1:ErSyn_Relship
name=74
/nameS='HE'
ersyn_relend1:ErSyn_Relend
name=8
/nameS='J'
relsyn_relschema2:RelSyn_RelSchema
name=73
/nameS='HD'
relsyn_relschema1:RelSyn_RelSchema
name=74
/nameS='HE'
base_attribute5:Base_Attribute
name=3
/nameS='D'
isKey=true
base_attribute2:Base_Attribute
name=7
/nameS='H'
isKey=false
base_attribute6:Base_Attribute
name=3
/nameS='D'
isKey=true
base_attribute1:Base_Attribute
name=7
/nameS='H'
isKey=false
base_attribute3:Base_Attribute
name=83
/nameS='JD'
isKey=true
base_attribute4:Base_Attribute
name=43
/nameS='ED'
isKey=true
base_datatype1:Base_DataType
name=97
/nameS='KH'
HE
E
J
HD
D:KH
H:KH

```
create table HD(
  D KH primary key,
  H KH)
create table HE(
  ED KH,
  JD KH,
  primary key(ED,JD))
```

Fig. 3. Generated ER and relational database schema with binary relationship.

per bounds for the) participating classes. (c) *Attributes* may be determined by specifying an enumeration of allowed values or by the set of values derived from the value set of the corresponding datatype. (d) The numerical *datatypes* Integer and Real may be configured through an enumeration (e.g., Set{42,44,46} or Set$\{3.14, 6.28, 9.42\}$) or with lower and upper bounds for the interval of allowed values with an additional step value for Real (for example, resulting in Set{-8..7} or Set$\{$-1, -0.5, 0, 0.5, 1$\}$). The datatype String may be determined by an enumeration (e.g., Set{'UML','OCL','MDE'}) or through a lower and upper bound for the number of automatically generated String literals (resulting in, for example, Set{'String1',...,'String7'}).

The example configuration requires (among other restrictions) the following: (a) there is exactly one transformation object (in class `Er2Rel_Trans`), and there are exactly two relational schemas (in class `RelSyn_RelSchema`); (b) the links in association `ErSyn_OwnershipErSchemaEntity` between `ErSyn_ErSchema` and `ErSyn_Entity` are not explicitly restricted, but only implicitly through the upper bounds of the participating classes, and there is no link in the association `ErSyn_OwnershipRelshipAttribute`, meaning that in the constructed ER schema there will be no relationship attribute; (c) the attribute isKey is allowed to take values from the enumeration Set{false,true} (recall that in UML and OCL more than two truth values are available); (d) the datatype Integer is allowed to take values from the interval [0..127].

The automatically generated transformation in Fig. 3 is displayed in form of the constructed object diagram and in form of a visual resp. textual domain-specific representation of the ER schema (in traditional ER notation) and the relational database schema (as textual SQL table declarations). In particular, the two relationship ends `E` and `J` of the relationship `HE` are represented as attributes `ED` and `JD` in the relational schema `HE`, because the attribute `D` constitutes the key in entity `HD` and in the relational schema `HD`. If there would be a composed key in the entity `HD`, say attributes `DA` and `DB`, the relational schema `HD` has to contain four attributes `EDA`, `EDB`, `JDA`, and `JDB`. Thus, the key attribute names on the relational side have to be composed from the relationship end and attribute names from the ER side. This section followed the ideas we have developed in [7].

4 Analyzing Static Transformation Model Properties by Coverage of Model Elements

Analyzing static transformation model properties means for us to explore the model transformation text in order to achieve relevant transformation properties. Static analysis is interesting because transformation models are usually structured at least into three parts: (a) the source, (b) the target, and (c) the transformation metamodel. Accompanying constraints will be found in the respective parts. For a single constraint it is thus particular interesting whether it restricts all three parts in conjunction or it treats a single part only. Such an analysis is possible with the static technique proposed here that is based on the idea of covering, i.e., to analyze and to indicate which part of the underlying class diagram is covered by a particular constraint. In our example we even have four parts in the transformation model, namely the source, the target, the transformation, and a common part that represents model features that are used in both the source and the target (here, attributes and datatypes). This is probably not an unusual situation.

In Fig. 4 we have displayed four (of the 22) invariants in the transformation model. Below we show the invariants also in detail. The coloring in the figure indicates the degree the respective model element (here classes, attributes, operations) is used in and covered by the constraint. By inspecting the invariant's color coverage profile one can analyze its effect on the respective model element, and one gets an impression about its dominance.

```
context self:ErSyn_Relend inv c_Relend_Entity_Relship_ErSchema:
  self.entity.erSchema=self.relship.erSchema
context self:ErSyn_Entity inv uniqueOsRelendNamesWithinEntity:
  self.osRelend()->forAll(re1,re2 | re1.name=re2.name implies re1=re2)
context self:Base_Attribute inv linkedToOneOfEntityRelshipRelSchema:
  (self.entity->size)+(self.relship->size)+(self.relSchema->size)=1
context self:Er2Rel_Trans inv forEntityExistsOneRelSchema:
  self.erSchema.entity->forAll(e |
    self.relDBSchema.relSchema->one(rl |
      e.name=rl.name and
      e.attribute->forAll(ea |
        rl.attribute->one(ra |
          ea.name=ra.name and ea.dataType=ra.dataType and
          ea.isKey=ra.isKey))))
```

`ErSyn_Relend::c_Relend_Entity_Relship_ErSchema` basically expresses that the path from `Relend` over `Entity` to `ErSchema` coincides with the path from `Relend` over `Relship` to `ErSchema` (`c_` stands for 'commutativity constraint'). This is reflected in the invariant's coverage profile.

`ErSyn_Entity::uniqueOsRelendNamesWithinEntity` restricts the other-side-relends (`osRelends()`) of an entity. For example, if we have `Person-employee-Job-employer-Company`, then `employer` is an `osRelend` of `Person` and `employee` is an `osRelend` of `Company`. An `osRelend` can be applied to an entity just like an attribute is applied. The collection of the `osRelends` must be unique for an entity. The coverage profile of the constraint clearly expresses that the constraint is working on the ER side only and points out the influence of the operation `osRelend()`.

`Base_Attribute::linkedToOneOfEntityRelshipRelSchema` requires that an `Attribute` either belongs to an `Entity` or to a `Relship` or to a `RelSchema`, but not to more than one model element although the multiplicities would allow this. The coverage profile points to and emphasizes the connection between the four mentioned metaclasses.

`Er2Rel_Trans::forEntityExistsOneRelSchema` is a transformation (`Trans`) constraint and thus covers a large portion of the metamodel, the source, the

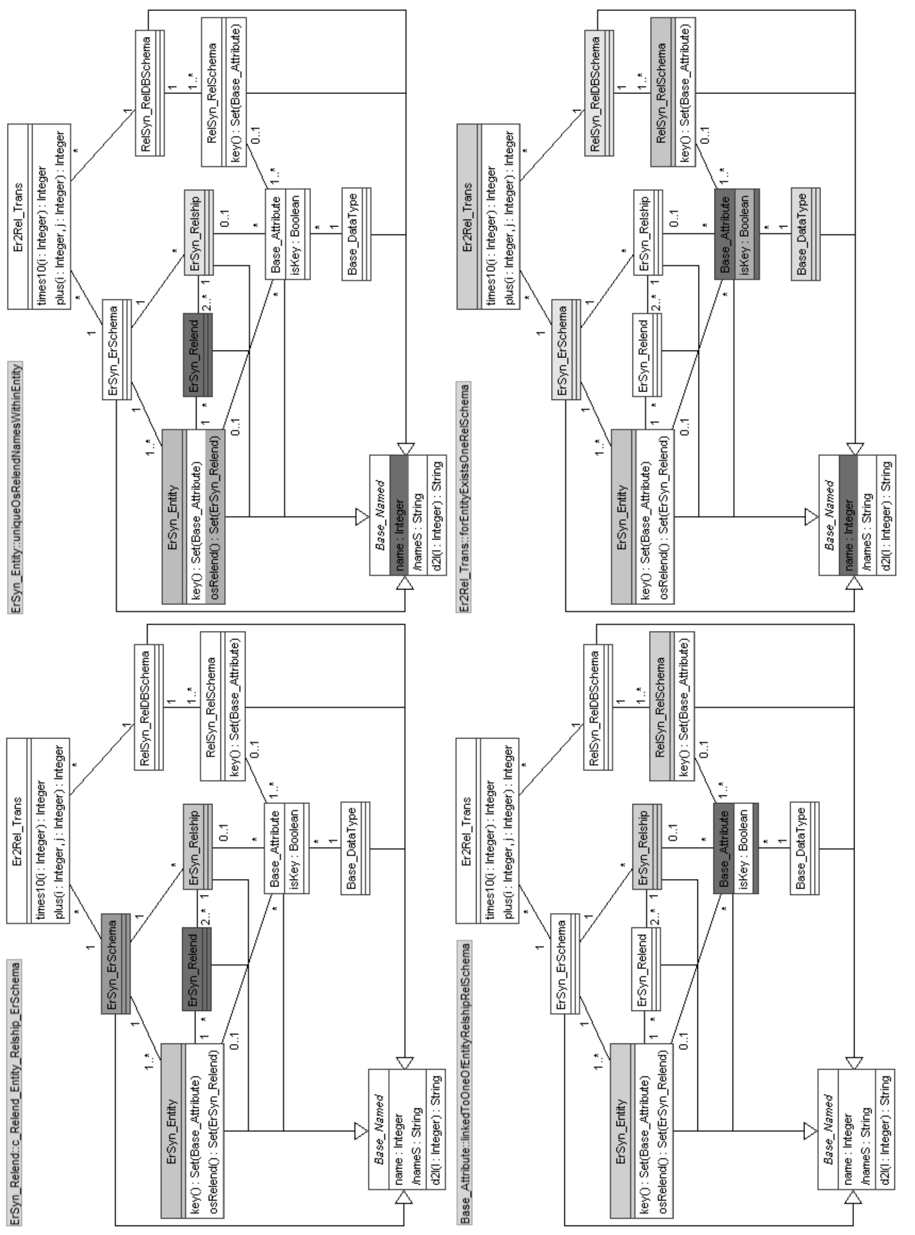

ErSyn_Relend::c_Relend_Entity_Relship_ErSchema
ErSyn_Entity::uniqueOsRelendNamesWithinEntity
Base_Attribute::linkedToOneOfEntityRelshipRelSchema
Er2Rel_Trans::forEntityExistsOneRelSchema
Er2Rel_Trans
times10(i : Integer) : Integer
plus(i : Integer, j : Integer) : Integer
RelSyn_RelDBSchema
RelSyn_RelSchema
key() : Set(Base_Attribute)
ErSyn_ErSchema
ErSyn_Relship
ErSyn_Relend
ErSyn_Entity
key() : Set(Base_Attribute)
osRelend() : Set(ErSyn_Relend)
Base_Attribute
isKey : Boolean
Base_DataType
Base_Named
name : Integer
/nameS : String
d2l(i : Integer) : String
1
*
1..*
0..1
2..*

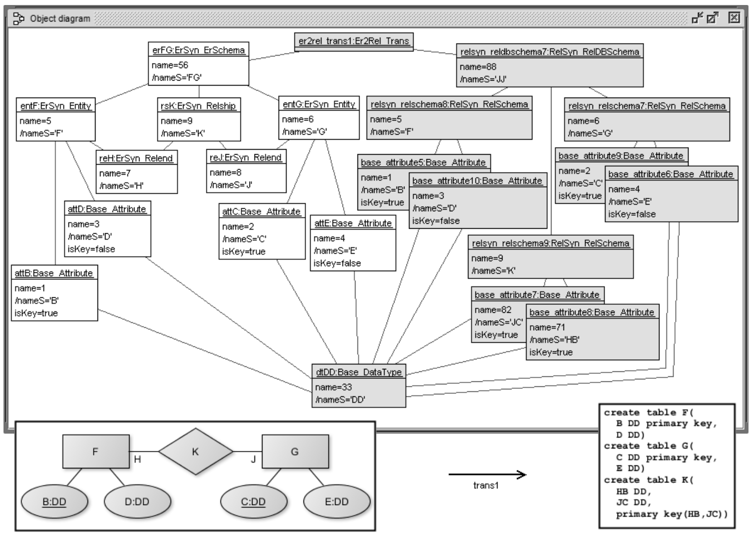
Fig. 4. Coverage of model elements for selected invariants.

target, and the transformation itself. As the constraint deals with `Entity` objects only, the coverage reveals that `Relship` or `Relend` objects are *not* touched. If one goes through all `Trans` constraints, one basically discovers that the operation `osRelend()` is not used in the `Trans` part at all. Thus the coverage indicates that this operation is *not* relevant for the transformation.

Currently we have realized the coverage analysis in the graphical user interface and with predefined metrics. We are considering to represent the analysis results in textual and table form as well and to offer apart from predefined metrics the option to let the developer define her project specific metrics, if desired.

5 Analyzing Dynamic Transformation Model Properties by Transformation Completion

Analyzing dynamic transformation model properties means for us to actually construct transformation instances in form of object diagrams. Doing so can reveal relevant transformation properties. The properties and questions that we



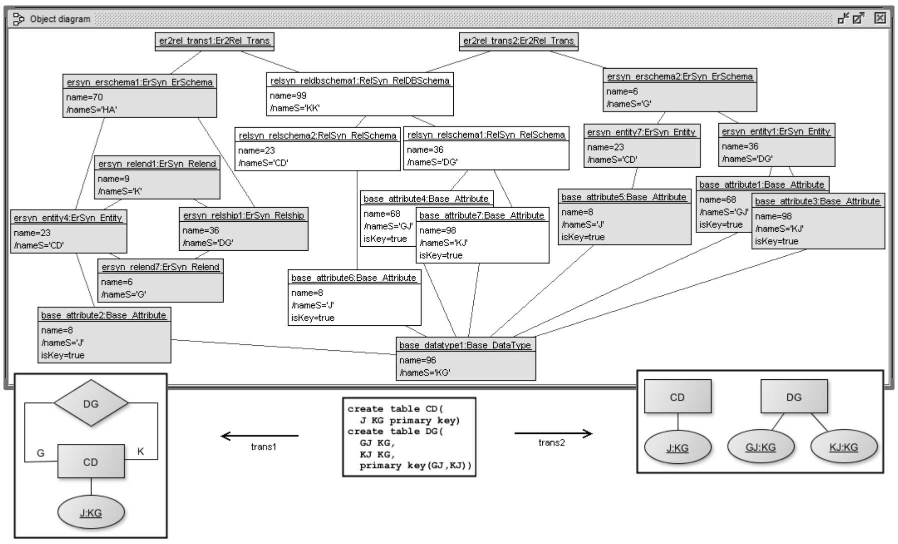
```
create table F(
  B DD primary key,
  D DD)
create table G(
  C DD primary key,
  E DD)
create table K(
  HB DD,
  JC DD,
  primary key(HB,JC))
```

Fig. 5. Transformation completion starting from an ER schema.

consider here are: (a) given a concrete ER schema that is manually constructed, is it possible to automatically complete the transformation yielding a relational database schema and to show by this the effectiveness of the transformation and inspect whether the transformation model constructs the expected result (see Fig. 5) and (b) given a manually constructed relational database schema, is it possible to complete the partially given object diagram and to show that the transformation is non-unique in the sense that the given relational database schema has two ER counterparts (see Fig. 6).

In Fig. 5 question (a) is treated. A partial object diagram representing the manually constructed ER schema (the white objects in the left part of the figure) is handed to the model validator in order to complete the object diagram. The completion is shown in the right part of the figure. The model validator configuration asks for one transformation object, one connected ER schema, and one connected relational database schema.

In Fig. 6 question (b) is handled. A partial object diagram representing the manually constructed relational database schema (the white objects in the middle of the figure) is handed to the model validator in order to complete the object diagram.


Object diagram
er2rel_trans1:Er2Rel_Trans
er2rel_trans2:Er2Rel_Trans
ersyn_erschema1:ErSyn_ErSchema
name=70
/nameS='HA'
relsyn_reldbschema1:RelSyn_RelDBSchema
name=99
/nameS='KK'
ersyn_erschema2:ErSyn_ErSchema
name=6
/nameS='G'
relsyn_relschema2:RelSyn_RelSchema
name=23
/nameS='CD'
relsyn_relschema1:RelSyn_RelSchema
name=36
/nameS='DG'
ersyn_entity7:ErSyn_Entity
name=23
/nameS='CD'
ersyn_entity1:ErSyn_Entity
name=36
/nameS='DG'
ersyn_relend1:ErSyn_Relend
name=9
/nameS='K'
base_attribute4:Base_Attribute
name=68
/nameS='GJ'
isKey=true
base_attribute7:Base_Attribute
name=98
/nameS='KJ'
isKey=true
base_attribute5:Base_Attribute
name=8
/nameS='J'
isKey=true
base_attribute1:Base_Attribute
name=68
/nameS='GJ'
isKey=true
base_attribute3:Base_Attribute
name=98
/nameS='KJ'
isKey=true
ersyn_entity4:ErSyn_Entity
name=23
/nameS='CD'
ersyn_relship1:ErSyn_Relship
name=36
/nameS='DG'
ersyn_relend7:ErSyn_Relend
name=6
/nameS='G'
base_attribute6:Base_Attribute
name=8
/nameS='J'
isKey=true
base_attribute2:Base_Attribute
name=8
/nameS='J'
isKey=true
base_datatype1:Base_DataType
name=96
/nameS='KG'
DG
G
K
CD
J:KG
trans1
trans2
CD
DG
J:KG
GJ:KG
KJ:KG

```
create table CD(
  J KG primary key)
create table DG(
  GJ KG,
  KJ KG,
  primary key(GJ,KJ))
```

Fig. 6. Non-unique transformation completion starting from a relational DB schema.

The ER completions are shown in the left and the right part of the figure. The model validator configuration in this case explicitly asks for two transformation objects connected to a single relational database schema and connected to two different ER schemas. Additionally, two invariants had to be added for the process of finding the proper object diagram. These invariants are not part of the transformation model, but are needed to drive the model validator into the proper direction.

```
context ErSyn_ErSchema inv connectedToTransformation:
  self.trans->notEmpty()
context ErSyn_ErSchema inv oneWithRelship_oneWithoutRelship:
  ErSyn_ErSchema.allInstances()->exists(with,without|
    with.relship->notEmpty() and without.relship->isEmpty())
```

Summarizing, we observe that the approach allows the developer to check the injectivity of a transformation model in either direction. We have considered a transformation model in one particular direction, from the relational database model to the ER model, and were able to show through the construction of an example, that the particular considered direction of the transformation is not injective because one relational database schema was connected with two ER schemas. As the approach is grounded on finite checks, it is not possible to prove in general that a transformation going into one direction is injective, but one can show through examples the non-injectivity.

6 Conclusion

The paper presented an approach for automatically checking transformation model features. We have analyzed transformation models statically by identifying model elements in the underlying class diagram that are covered by a transformation model invariant. We also checked transformation models dynamically through the completion of partially specified transformation pairs.

Future work could consider to study invariant independence, i.e., minimality of transformation models. The static analysis features can be improved by presenting the results in table and text form and through the introduction of project specific definition of metrics. The handling of strings must be improved. Last but not least, larger case studies must check the practicability of the approach.

References

1. Amrani, M., Lucio, L., Selim, G.M.K., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In Antoniol, G., Bertolino, A., Labiche, Y., eds.: Proc. Workshops ICST, IEEE (2012) 921–928
2. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to Systematic Model Transformation Testing. CACM **53**(6) (2010) 139–143
3. Bezivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. 9th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2006), Springer, Berlin, LNCS 4199 (2006) 440–453
4. Büttner, F., Egea, M., Guerra, E., de Lara, J.: Checking Model Transformation Refinement. In Duddy, K., Kappel, G., eds.: Proc. Inf. Conf. ICMT. LNCS 7909, Springer (2013) 158–173
5. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and Validation of Declarative Model-To-Model Transformations through Invariants. Journal of Systems and Software **83**(2) (2010) 283–302
6. Gogolla, M.: Tales of ER and RE Syntax and Semantics. In Cordy, J.R., Lämmel, R., Winter, A., eds.: Transformation Techniques in Software Engineering, IBFI, Schloss Dagstuhl, Germany (2005) Dagstuhl Seminar Proceedings 05161. 51 pages.
7. Gogolla, M., Hamann, L., Hilken, F.: Checking Transformation Model Properties with a UML and OCL Model Validator. In: Proc. 3rd Int. STAF'2014 Workshop Verification of Model Transformations (VOLT'2014). (2014) CEUR Proceedings.
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
9. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In France, R., Kazmeier, J., Breu, R., Atkinson, C., eds.: Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012), Springer, Berlin, LNCS 7590 (2012) 415–431
10. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2007). (2007) LNCS 4424, 632–647

Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation

Elie Richa[1,2], Etienne Borde[1], Laurent Pautet[1]
Matteo Bordin[2], and José F. Ruiz[2]

[1] Institut Telecom; TELECOM ParisTech; LTCI - UMR 5141
46 Rue Barrault 75013 Paris, France
`firstname.lastname@telecom-paristech.fr`
[2] AdaCore, 46 Rue d'Amsterdam 75009 Paris, France
`lastname@adacore.com`

Abstract. Complex model-based tools such as code generators are typically designed as chains of model transformations taking as input a model of a software application and transforming it through several intermediate steps and representations. The complexity of intermediate models is such that testing is more conveniently done on the integrated chain, with test models expressed in the input language. To achieve a high test coverage, existing transformation analyses automatically generate constraints guiding the generation of test models. However, these so called *test objectives* are expressed on the complex intermediate models. We propose to back-propagate test objectives along the chain into constraints and test models in the input language, relying on precondition construction in the theory of Algebraic Graph Transformation. This paper focuses on a one-step back-propagation.

Keywords: testing, model transformation chains, algebraic graph transformation, weakest precondition, ATL

1 Introduction

Tools used in the production of critical software, such as avionics applications, must be thoroughly verified: an error in a tool may introduce an error in the critical software potentially putting equipment and human lives at risk. Testing is one of the popular methods for verifying that such tools behave as specified. When testing critical applications, a primary concern is to ensure high *coverage* of the software under test (*i.e.* ensure that all features and different behaviors of the software are tested). The recommended way to achieve this is to consider each component separately, identify its functionalities, and develop dedicated tests (*unit testing*). This guideline is therefore reflected in industrial software quality standards such as DO-330 [13] for tools in the avionics domain.

However, with complex model transformation tools such as code generators, applying unit testing is very costly and impractical. In fact, such tools are often designed as a chain of several model transformations taking as input a model developed by the user in a high-level language and transforming it through several steps. Unit testing then boils down to testing each step of the chain independently. In practice, intermediate models increase in detail and complexity as transformations are applied making it difficult to produce test models in the intermediate representations: manual production is both error-prone because of the complexity of the languages and tedious since intermediate languages do not typically have model editors [1]. It is often easier for the tester to create a model in the input language of the chain, with elements that he knows will exercise a particular feature down the chain. Existing approaches [7,8,11] can automate the production of tests for model transformations, thus producing unit tests. However, when a test failure uncovers an error, analyzing the complex intermediate representations is difficult for the developer.

Given these factors, we propose an approach to the testing of model transformation chains that aims to ensure test coverage of each step while preserving the convenience of using test models in the input language. First we rely on existing analyses [7,8,11] to generate a set of so-called *test objectives* that must be satisfied by test models to ensure sufficient coverage. Then we propose to automatically propagate these test objectives *backward* along the chain into constraints over the input language. The back-propagation relies on the construction of preconditions in the theory of Algebraic Graph Transformation (AGT) [5].

Within this general approach, we focus in this paper on the translation of postconditions of one ATL transformation step to preconditions, which is a key operation in the propagation of test objectives. We thus propose a first translation of the ATL semantics into the AGT semantics where we use the theoretical construction of *weakest preconditions* [9] . We illustrate our proposal on a realistic code generation transformation using a prototype implementation based on the *Henshin*[3] and *AGG*[4] frameworks. This first prototype allowed us to back-propagate test objectives across one transformation step.

In the remainder of the paper, section 2 gives an overview of the testing approach, explaining the role of precondition construction. Section 3 recalls the main concepts of ATL and AGT. Section 4 introduces an example of ATL transformation that will serve to illustrate (i) the translation of ATL to AGT in section 5 and (ii) the construction of preconditions in section 6. Finally, we present our prototype in section 7 and conclude with our future plans in section 8.

2 General Approach

As highlighted in [1], one of the major challenges in achieving thorough testing is producing test models that are relevant, *i.e.* likely to trigger errors in the implementation. Several approaches address this challenge for standalone transformations. In [7], [8] and [11], the authors propose to consider a transformation

[3] The Henshin project, `http://www.eclipse.org/henshin`

[4] The Attributed Graph Grammar development environment, `http://user.cs.tu-berlin.de/~gragra/agg`

and analyse one or more of (i) the input metamodel, (ii) the transformation specification and (iii) the transformation implementation. This analysis results in a set of constraints, each describing a class of models that are relevant for finding errors in the transformation. We refer to such constraints as *test objectives* in the remainder of the paper. Constraint satisfaction technologies such as the Alloy Analyzer[5] and EMFtoCSP [6] are then used to produce model instances such that each test objective is satisfied by at least one test model.

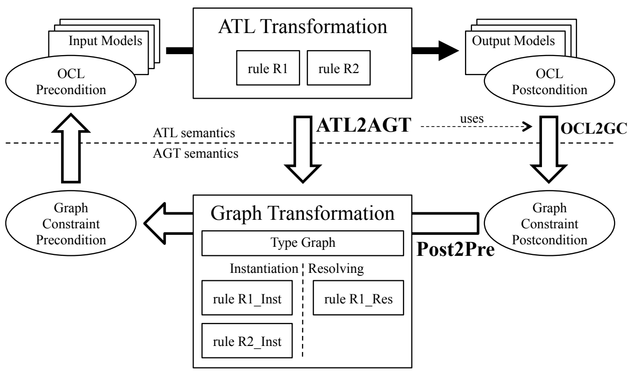


Fig. 1: Transformation of Postcondition to Precondition

Let us now consider a transformation chain $M_i \xrightarrow{T_i} M_{i+1}$ for $0 \leq i < N$ where an input model M_0 is processed by N successive transformation steps T_i into intermediate models M_i and ultimately into the final output model M_N. Focusing on an intermediate transformation T_i such that $i > 0$, we can apply the above approaches to obtain a set of test objectives $\{to_{i,j} \mid 0 \leq j\}$ ensuring the thoroughness of the testing of T_i. Each test objective $to_{i,j}$ is a constraint expressed over the input metamodel of T_i. At this point we want to produce a model M_0 at the beginning of the chain, which ultimately satisfies $to_{i,j}$ after being processed by the sequence $T_0 \ ; \cdots ; \ T_{i-1}$. We propose to automate this operation by transforming $to_{i,j}$ into a test objective $to_{i-1,j}$ at the input of T_{i-1} and thus iterate the process until we obtain $to_{0,j}$ that can serve to produce a model M_0. The key challenge of this paper is to devise an analysis that takes as input a *constraint* $to_{i,j}$ and a transformation specification T_{i-1}, and produces as output a *constraint* $to_{i-1,j}$. Such a method exists in the formal framework of *Algebraic Graph Transformation* (AGT) [5] in the context of the formal proof of correctness of graph programs. It is the transformation of postconditions into preconditions [9] that we propose to adapt and reuse in our context. Since we consider transformations specified in ATL [10], a translation to AGT is necessary.

As shown in Figure 1, we propose to translate the ATL transformation T_{i-1} into a graph transformation program (*ATL2AGT* arrow) and $to_{i,j}$ into a graph

[5] Alloy language and tool, `http://alloy.mit.edu/`

constraint (*OCL2GC* arrow). Assuming the constraint is a postcondition of T_{i-1}, we automatically compute the precondition $to_{i-1,j}$ that is sufficient to satisfy the postcondition (*Post2Pre* arrow) using the formal foundation of AGT. Since ATL embeds OCL constraints, *ATL2AGT* also uses *OCL2GC*. However this is a complex translation [14] that will not be addressed given the space limitations. We thus focus on a first proposal of *ATL2AGT* in section 5 and *Post2Pre* in section 6, both limited to the structural aspects of the semantics and constraints. First, we recall the main elements of ATL and AGT in the next section.

3 Semantics of ATL and AGT

3.1 ATL and OCL

ATL [10] is a model-to-model transformation language combining declarative and imperative approaches in a hybrid semantics. A transformation consists of a set of declarative *matched rules*, each specifying a source pattern and a target pattern. The source pattern is a set of objects of the input metamodel and an optional OCL[6] constraint acting as a guard. The target pattern is a set of objects of the output metamodel and a set of bindings that assign values to the attributes and references of the output objects. The execution of a transformation consists of two main phases. First, the matching phase searches in the input model for objects matching the source patterns of rules (*i.e.* satisfying their filtering guards). For each match of a rule's source pattern, the objects specified in the target pattern are instantiated. A tuple of source objects may only match one rule, otherwise an error is raised. For this reason the order of application of rules is irrelevant. Second, the target elements' initialization phase executes the bindings for each triggered rule. Bindings map scalar values to target attributes, target objects (instantiated by the same rule) to target references, or source objects to target references. In the latter case, a *resolve* operation is automatically performed to find the rule that matched the source objects, and the first output object created by that rule (in the first phase) is used for the assignment. If no or multiple resolve candidates are found, the execution stops with an error.

As the current proposal is limited to structural aspects, we only consider bindings of target references and not those of attributes. OCL constraints are not considered as *OCL2GC* (Figure 1) is too complex to address within this paper [14]. Instead, we will use test objectives in the form of AGT graph constraints.

3.2 AGT and Graph Constraints

Several graph transformation approaches are proposed in the theory of Algebraic Graph Transformation [5]. We will be using the approach of *Typed Attributed Graph Transformation with Inheritance* which we found suitable to our needs and which is supported in the AGG tool allowing for concrete experimentation of our proposals (see section 7). There are 3 main elements to a graph transformation:

[6] Object Constraint Language (OCL), `http://www.omg.org/spec/OCL`

a *type graph*, a set of *transformation rules*, and a *high-level program* specifying the order of execution of rules.

Graphs consist of *nodes* connected with directed *edges*. Much like models conform to metamodels, typed graphs conform to a *type graph*. As introduced in [3], *metaclasses*, *references* and *metaclass inheritance* in metamodels correspond to *node types*, *edge types*, and *node type inheritance* in type graphs which allows an easy translation between the two. Even though multiplicities and containment constraints are not addressed in type graphs, they are supported in AGG.

A graph transformation is defined as a set of *productions* or *rules* executed in a graph rewriting semantics. There are two major approaches to defining rules and their execution. Even though the theory we use is based on the Double Pushout (DPO) approach, we will use the simpler Single Pushout (SPO) approach and notation which is also the one implemented in AGG. A rule consists of a *morphism* from a *Left-Hand Side* (LHS) graph to a *Right-Hand Side* (RHS) graph. The LHS specifies a pattern to be matched in the transformed graph. Elements mapped by the morphism are preserved and elements of the RHS that are not mapped by the morphism are new elements added to the transformed graph. We do not address element deletion since our translation will not need it (see section 5). Thus the execution of a rule consists in finding a match of the LHS in the transformed graph and adding the new nodes and edges.

With the transformation rules defined above, we can construct so called *high-level programs* [9] consisting of the sequencing or the iteration of rules. A program can be (1) elementary, consisting of a rule p, (2) the sequencing of two programs P and Q denoted by $(P;Q)$, or (3) the iteration of a program P as long as possible, denoted by $P\downarrow$, which is equivalent to a sequencing $(P;(P;(P\cdots)$ until the rule no longer applies.

Graph constraints are similar to OCL constraints for models. They are defined inductively as nested conditions, but for the sake of simplicity we consider a very basic form $\exists(C)$ where C is a graph. A graph G satisfies such a constraint if G contains a subgraph isomorphic to C. This form is suitable to express test objectives which typically require particular patterns to exist in models.

Next, we present the example that will help us illustrate our proposal.

4 Example: Code Generation

We aim to apply our approach to a realistic code generator from Simulink[7] to Ada/C source code, under development in the collaborative research project *Project P*[8]. Simulink is a synchronous data flow language widely used by industrials for the design of control algorithms. The code generator consists of a chain of up to 12 model transformations (depending on configuration options), including flattening of nested structures, sequencing, code expansion and optimisation. We consider the *Code Model Generation* (CMG) transformation step of

[7] MathWorks Simulink, `http://www.mathworks.com/products/simulink/`
[8] Project P, `http://www.open-do.org/projects/p/`

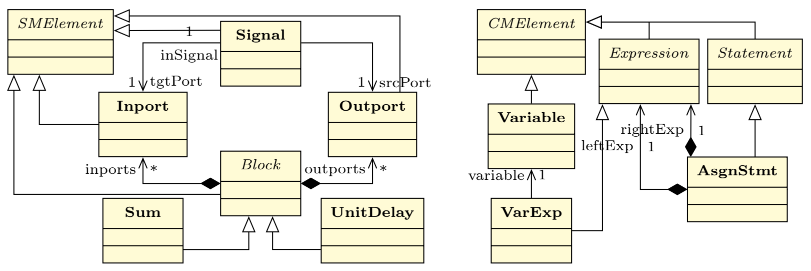


Fig. 2: Input and Output Metamodels

this chain to illustrate our translation to the AGT semantics. Then, considering a postcondition on the output of CMG, we construct a precondition on its input.

CMG transforms a Simulink model into a model of imperative code. A simplified version of the input metamodel is shown on the left side of Figure 2. Computation blocks such as *Sum* or *UnitDelay* receive data through their *Inports* and send the result of their computation through their *Outports*. *Signals* convey data from a source *Outport* to a target *Inport*. The output metamodel of CMG shown on the right side of Figure 2 features variables (*Variable*), expressions (*Expression*), references to variables (*VarExp*) and imperative code statements. In particular, an assignment statement (*AsgnStmt*) assigns its *rightExp* expression to its *leftExp* which typically is a reference to a variable.

Listing 1.1: The Code Model Generation ATL transformation

```
 1  rule O2Var { from oport : SMM!Outport
 2               to   var   : CMM!Variable }
 3
 4  rule S2VExp { from sig    : SMM!Signal
 5                to   varExp : CMM!VarExp (variable <- sig.srcPort) }       -- Resolve
 6
 7  rule UDel {
 8    from block        : SMM!UnitDelay
 9    to   assgnStmt    : CMM!AsgnStmt  (rightExp <- outVarExp,
10                                        leftExp <- memVarExp1),
11         memAssgnStmt : CMM!AsgnStmt( rightExp <- memVarExp2,
12                                  leftExp <- block.inports->at(1).inSignal), -- Resolve
13         memVar       : CMM!Variable,
14         outVarExp    : CMM!VarExp(variable <- block.outports->at(1)),      -- Resolve
15         memVarExp1   : CMM!VarExp(variable <- memVar),
16         memVarExp2   : CMM!VarExp(variable <- memVar) }
```

The ATL implementation of the CMG transformation consists of the 3 matched rules in Listing 1.1. The first rule creates a *Variable* for each *Outport* of the input model, and the second one creates a *VarExp* for each *Signal*. Note that the second rule requires resolving the *Outport* at line 5 into a *Variable* and will be used to illustrate our modeling of the resolve mechanism in AGT. The last rule creates 2 assignment statements referencing a *Variable* created by the same rule at line 13, a *VarExp* resolved at line 12, and a *Variable* resolved at line 14.

As for the test objective, we consider it directly in the graph constraint form in Figure 3. It requires that an assignment statement exists where both the source

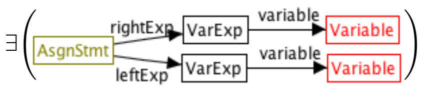


Fig. 3: Example Test Objective

and the target of the assignment are references to variables. This pattern matches the objects created by ATL rule *UDel*, and thus requires resolve operations.

5 ATL to Algebraic Graph Transformation

This section introduces our main contribution, the translation of ATL transformations to artifacts of an algebraic graph transformation: a type graph, graph transformation rules and a high-level program. Given the rewriting semantics of AGT and the exogeneous nature of the transformations we consider, we choose to model the ATL transformation as a rewriting of the input graph that adds the output elements. Consequently, the type graph includes types corresponding to both the input and the output metamodels. As explained in Section 3.2, the correspondence of metamodel elements to graph type elements is straightforward [3], and the resulting type graph is depicted in Figure 4. In addition, *tracing* node types are added to support the ATL resolve mechanism. First, an abstract *Trace* node relates source objects (*SMElement*) to target objects (*CMElement*) of ATL rules. Second, for each ATL rule, a concrete trace node (named *<atlrule-name>_ Trace*) references the actual source and target types of this rule. These trace nodes will be used by the graph transformation rules, as explained next.

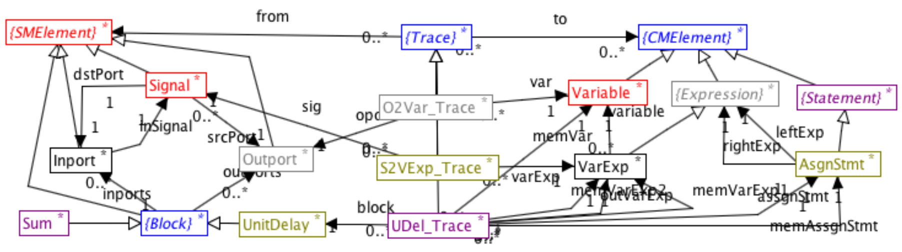
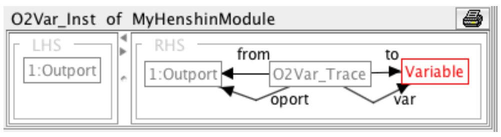


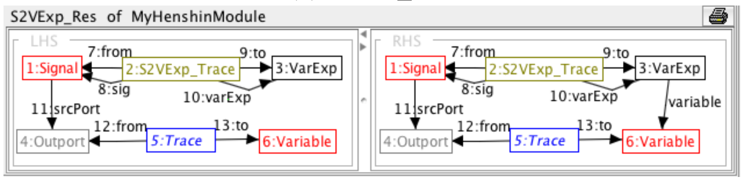
Fig. 4: Resulting Type Graph in AGG

Much like the execution semantics of ATL, the graph transformation starts with a set of *instantiation* rules that create output nodes without linking them. For example, *O2Var_ Inst* in Figure 5a matches an *Outport* and creates a *Variable* and a concrete trace *O2Var_ Trace* relating the source and target nodes (numbers indicate mapping by the rule morphism). Then, a second set of *resolving* rules rely on the trace nodes produced in the first phase to link output nodes. For example, *S2VExp_ Res* in Figure 5b matches an *Outport* and a *Trace* node to find the resulting *Variable* and create the *variable* edge. Thus the elements created in the RHS of Figure 5a (*O2Var_ Trace* and *Variable*) are matched later by the LHS in Figure 5b (*Trace* and *Variable*). Note the use of abstract *Trace* nodes in the resolving rules to allow resolving with any rule as long as the number and types of source and target elements match, as per the ATL semantics.

Finally, a high-level program implements the two phases by iterating instantiation rules first and resolving rules second, yielding the following for CMG: $P = O2Var\_Inst\downarrow; S2VExp\_Inst\downarrow; UDel\_Inst\downarrow; S2VExp\_Res\downarrow; UDel\_Res\downarrow$



(a) O2Var_Inst



(b) S2VExp_Res

Fig. 5: GTS rules translated from ATL rules

Having translated the ATL transformation to the AGT semantics, we next explain how we use precondition construction to back-propagate test objectives.

6 Transformation of Postcondition to Precondition

In [9], Habel, Pennemann and Rensink formally define a construction of *weakest precondition* for high-level programs in the interest of proving transformation correctness. Given a program and a postcondition, the weakest precondition is a constraint that characterizes all possible input graphs that lead to the termination of the program with a final graph satisfying the postcondition. A precondition construction is defined for one rule application and applied inductively to the sequence of rules defined by the program. In the case of $P\downarrow$ programs each number of iterations of P from 0 to ∞ must be considered, making the construction theoretically infinite.

However, in contrast with proof of correctness, we actually do not need to compute the *weakest* precondition. Since the final goal is to find a test model satisfying the test objective, computing one sufficient precondition would be enough. To do so, we limit iterations of rules in the program to a bounded number, making the precondition construction finite (the choice of bounds remains an open point at this stage). For example we can bound the CMG transformation to two applications of *O2Var* and one application of each of the other rules: $P = O2Var\_Inst; O2Var\_Inst; S2VExp\_Inst; UDel\_Inst; S2VExp\_Res; UDel\_Res$

As for the precondition construction of each rule, the theoretical construction requires to consider all possible overlaps of the RHS of the rule with the graph of the postcondition. Each overlap represents a way in which the rule may contribute to the postcondition. For each overlap, we perform an operation similar to a backwards execution of the rule[9] and thus construct a sufficient precondition.

7 Prototype and Results

We have prototyped our approach using the *Henshin* and *AGG* frameworks. *ATL2AGT* is implemented with the Henshin API, and an existing service is used to export the artifacts to AGG. Precondition construction is not readily available in AGG, so we have implemented *Post2Pre* using the existing services such as generating overlaps of two graphs and constructing a pushout complement. For the example test objective introduced in Figure 3, two of the preconditions we obtain are shown in Figure 6. The existence of one of these patterns in input models ensures that the *UDel* rule is able to execute and resolve the necessary elements to produce the pattern required by the test objective.

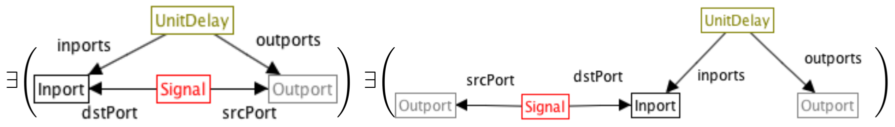


Fig. 6: Preconditions Computed for the Example Test Objective

8 Conclusion

In this paper we have approached the problem of testing model transformation chains with two main concerns: achieving high test coverage and using test models in the input language of the chain to ease the analysis of detected errors. To this end, we have proposed to extend existing approaches of test objective generation with a method to propagate intermediate test objectives back to the input language. Central to this method is the transformation of postconditions of one transformation step into preconditions, which was the focus of this paper. We have contributed a first translation from ATL semantics into the AGT semantics and adapted the theoretical precondition construction to achieve our goal.

[9] the formal construction is a *pushout complement*

In future work, we plan to investigate the *OCL2GC* step of our approach and alleviate the limitation to structural aspects by handling object attributes based on works such as [4,12,14]. Moreover, we plan to work towards test-suite minimality [2] by allowing a test model to cover several test objectives across the chain and only back-propagating non-satisfied test objectives.

References

1. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, June 2010.
2. E. Bauer, J. Küster, and G. Engels. Test suite quality for model transformation chains. In *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2011.
3. E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.
4. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Synthesis of OCL pre-conditions for graph transformation rules. In *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 45–60. Springer Berlin Heidelberg, 2010.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.
6. C. Gonzalez, F. Buttner, R. Clariso, and J. Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 44–50, June 2012.
7. C. González and J. Cabot. ATLTest: A white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 449–464. Springer Berlin Heidelberg, 2012.
8. E. Guerra. Specification-driven test generation for model transformations. In *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg, 2012.
9. A. Habel, K.-H. Pennemann, and A. Rensink. Weakest preconditions for high-level programs. In *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460. Springer Berlin Heidelberg, 2006.
10. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.
11. J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static analysis of model transformations for effective test generation. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 291–300, 2012.
12. C. M. Poskitt and D. Plump. A Hoare calculus for graph programs. In *Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2010.
13. RTCA. *DO-330 : Software Tool Qualification Considerations.* 2011.
14. J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *Electronic Notes in Theoretical Computer Science*, 211(0):159 – 170, 2008. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).

Towards Approximate Model Transformations

Javier Troya[1], Manuel Wimmer[1], Loli Burgueño[2], and Antonio Vallecillo[2]

[1] Business Informatics Group, Vienna University of Technology, Austria
{troya,wimmer}@big.tuwien.ac.at
[2] ETSI Informática, Universidad de Málaga, Spain
{loli,av}@lcc.uma.es

Abstract. As the size and complexity of models grow, there is a need to count on novel mechanisms and tools for transforming them. This is required, e.g., when model transformations need to provide target models without having access to the complete source models or in really short time—as it happens, e.g., with streaming models—or with very large models for which the transformation algorithms become too slow to be of practical use if the complete population of a model is investigated.
In this paper we introduce *Approximate Model Transformations*, which aim at producing target models that are accurate enough to provide meaningful and useful results in an efficient way, but without having to be fully correct. So to speak, this kind of transformations treats accuracy for execution performance. In particular, we redefine the traditional OCL operators used to query models (e.g., allInstances, select, collect, etc.) by adopting sampling techniques and analyse the accuracy of approximate model transformations results.

Keywords: Model Transformation, Approximation, Performance, Sampling

1 Introduction

Model transformations (MTs) are gaining acceptance as model-driven techniques are becoming commonplace. While models capture the views on systems for particular purposes and at given levels of abstraction, MTs are in charge of the manipulation, analysis, synthesis, and refinement of the models [3]. They do not only allow the generation of implementations from high-level models, but also to generate other views that can be properly analyzed or that provide users with the information they need, at the right level of abstraction, e.g., a synopsis of a larger data set.

So far the community has mainly focused on the *correct* implementation of a MT, according to its specification [4, 14–16, 30, 32], although there is an emergent need to consider other (non-functional) aspects such as performance, scalability, usability, maintainability and so forth [5]. In particular, the study of the performance of MTs is gaining interest as very large models living in the cloud have to be transformed as well [8, 28, 29]. The usual approach to improve performance has focused on the use of incremental execution [7, 22] and parallelization techniques [8, 28].

In this paper we explore a different path. Our aim is to weaken the need to produce *exact* target models but *approximate* ones. Such approximate target models should be accurate enough to provide meaningful and useful results to users, but alleviate the need for the transformation to generate fully correct models—being able to produce such target models in much shorter time. We call *Approximate Model Transformations* (AMT) those model transformations that produce *approximate* target models staying inside a given error bound. For this, we investigate the adoption of statistical sampling techniques, such as they are employed in the database area [1, 10, 19] and in the general field of approximate computing [34].

This kind of MTs are needed in various circumstances. The most obvious situation is when very large models have to be synthesized into much smaller models for decision making, the synthesis process is not trivial, the decision should be made really fast, and near-optimal solutions or just accurate results are enough (without having to be fully correct). Another situation in which AMTs come also into play is when we cannot fully guarantee the correctness of the source models. For example, when dealing with infinite [9] or streaming models [11]: only a portion of them (e.g., the ones inside the *sliding window*) is available at any given moment in time. This means that some connections between model elements may be broken because not all the elements involved in these connections are available at the same time. By eliminating the need for the source model to be correct in this case (i.e., the portion we are considering is just a fragment [6], which does not need to conform to the source metamodel), we cannot guarantee that the transformation will produce a correct target model. As an example, think, e.g., of several wireless sensor networks sending information that needs to be semantically annotated, combined, and analyzed to make decisions about traffic routes [27]. Here the transformation rules in charge of distilling the information need to be somehow simplified in order to be fast—at the cost of sacrificing correctness.

There are several issues involved in these kinds of MTs. First, we need to provide a measure of the *accuracy* of the target model produced, what we do based on the confidence level and relative error, to be able to decide how good the results given by the approximate transformation are after analyzing them, and if they are good enough for the user's purposes. Second, we need to redefine in this context some of the traditional operators used to traverse or query the models: *allInstances()*, *select()*, *collect()*, etc. Third, we need to identify adequate methods for the design of approximate model transformations that provide accurate-enough results. Finally, we need to be able to identify specific scenarios where it makes sense to apply this kind of MTs.

The structure of the paper is as follows. Section 2 sets the context by presenting a motivating example used throughout the paper to illustrate our ideas. Section 3 presents the concept of AMT and describes the ideas for our approach, which are applied in the case study as presented in Section 4. Then, in Section 5 we discuss related work, and in Section 6 we conclude the paper with an outlook on future work.

2 Motivating Example

Let us consider a real-world example of a Wireless Sensor Network (WSN) for measuring different climatological conditions. The metamodel that we show in Fig. 1 has been extracted from the data [2] obtained from a WSN deployed in the Elora George Conservation Area, Ontario, Canada, and also out of some ideas gathered from [27]. The station deployed gathers 19 different kinds of data, from which we have selected 6

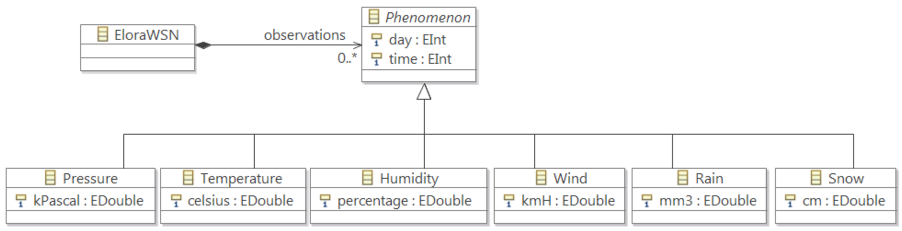


Fig. 1. Metamodel for observation phenomena in the WSN in Elora.

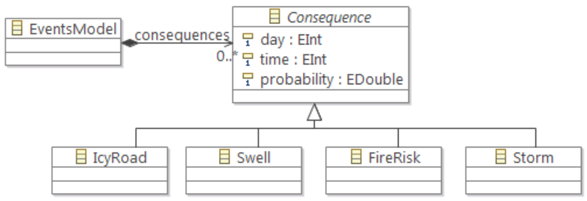


Fig. 2. Metamodel for defining consequences from observed phenomena.

of them. Thus, as we can see in the metamodel, the *EloraWSN* is composed of *observations*, of type *Phenomenon*. Each of them registers the *day* and *time*. The phenomena are specialized into different types: *Pressure*, *Temperature*, *Humidity*, *Wind*, *Rain* and *Snow*. Each of these specializations has an attribute that stores the value gathered from the environment. WSNs are normally a clear example of streaming models, since data is registered as time moves forward. Consequently, at any point in time, we cannot access the data that is to come in some future instant.

As explained in [27], to derive additional knowledge from semantically annotated sensor data, it is beneficial to use a rule-based reasoning engine. In this way, when a group of sensor nodes provides information regarding for instance temperature and precipitation, then, using such rules, we can specify possible road conditions. One of these rules could be the following: if the temperature is less than 1 degree Celsius and it is raining, then the roads are potentially icy. Also, the probability of the risk for having a fire or a storm could be obtained from the information about humidity, wind and temperature. To be able to express this kind of information in a model-based manner, we have created the metamodel shown in Fig. 2. It contains four different *Consequences* that are to be predicted according to the observed phenomena—there could be many more. Each consequence contains the *day* and *time* when it is predicted, as well as the *probability* that such consequence actually occurs.

In this context, we aim at reasoning over this kind of models not only considering the sliding window, i.e., the current information gathered from sensors that we have, but also taking into consideration that the information within such window can be potentially large. Thus, we focus on obtaining approximate predictions using AMTs.

3 Leveraging Sampling for AMTs

In statistics, a sample is a subset of individuals from within a statistical population that is known to be representative of the population as a whole. The process of using samples for addressing a specific problem is called *Sampling*. There exist several different sampling strategies. Which one to utilize depends on the context and the input data. For instance, in *Random Sampling*, each individual in the population is given the same probability to be in the sample. *Systematic Sampling*, in turn, involves a random start and then proceeds with the selection of every k^{th} individual from then onwards. There are other techniques to be used when the population embraces a number of distinct categories, such as *Stratified Sampling* or *Cluster Sampling*.

A common issue when selecting the sample of a population is to decide its size so that it can be representative. This is influenced by a number of factors, including the purpose of the study, population size, the risk of selecting a "bad" sample, and the tolerable sampling error. In [19], several strategies for determining the sample size are presented when the data in the population follow a normal distribution.

With the sampling concept in mind, our proposal aims to redefine the common operators that current model transformation languages use to manage and operate with collections, such as *allInstances()*, *collect()*, *select()*, ..., i.e., the collections operators used by OCL. When transforming very large models, these operations become very expensive, performance-wise, because they have to traverse the whole model and deal with a large number of elements. If we reduce this number of elements, the transformation will be faster.

For this reason we introduce a set of new collection operators, each one corresponding to an OCL collection operator. The new ones are suffixed by "*Approx*" and incorporate additional arguments: one indicating the confidence level (CL) and another one indicating the relative error (RE). For example, *Temperature.allInstancesApprox(95, 10)* returns a set of instances of type *Temperature* whose size is determined by the formula proposed in [19] according to the CL and RE specified.

Fig. 3 illustrates this idea. In the upper part of the figure, there are ten elements of a certain type. Realizing the *Type.allInstances()* operation would return all the elements. However, with our new operator *Type.allInstancesApprox(CL, RE)* we obtain only a subset of them. The lower part of the figure shows ten elements of a certain type that are referenced by the first element in the upper part. A normal *collect(Condition)* operation, where all the elements satisfy the condition, would select the ten elements (along with

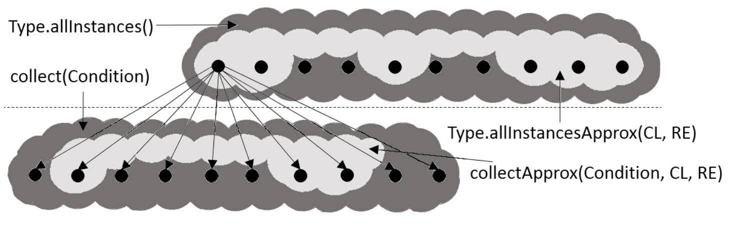

Type.allInstances()
collect(Condition)
Type.allInstancesApprox(CL, RE)
collectApprox(Condition, CL, RE)

Fig. 3. Idea for Approximate Operators

their relationship to the object in the upper part). This means that all vertices and arcs are selected. We apply now the same concept as before with the *collectApprox(Condition, CL, RE)* operation, so that only a subset of the graph is actually retrieved.

Similarly, when we have a model transformation rule (think for instance of ATL [21]), declarative rules can also have now these arguments. In fact, matched rules in ATL apply an implicit *allInstances()* operation in the matching part, since they obtain all elements that satisfy this part (in the simplest case, where the matching part specifies only one type with no filtering, i.e., all the instances of such type are transformed). Finally, we also consider our approximate operators for the imperative part of transformation languages. Indeed, when a loop like *for (p in e.observations)* is used (consider that *e* is an element of type EloraWSN), then all phenomena are traversed. For this reason, applying an approximate operator would avoid the complete traversal.

The values to be chosen in the new arguments of these operations depend on how accurate and fast we want our transformation to be. There is effectively a tradeoff between these two metrics. Thus, the smaller the CL and the higher the RE, the smaller the size of the sample population is and the faster the transformation computes, being of course less accurate, and vice versa.

4 Implementation and Evaluation

We now describe how we have implemented and evaluated our case study.

4.1 Description and Setting

For our experiment, we use data gathered from the WSN in Elora during 2013 [2]. The data we have obtained from the website consists of 8760 points in time (365 days $\times$ 24 measurements per day). For each of these points, there is data gathered for each type. We have increased the quantity of data, by extrapolating the original, in 2^6 points. Thus, we have 6 different types of data (cf. Fig. 1), and for each data we have 560768 different measurements through time, what results in 3364608 objects in our input model, whose file has a size of 306 MB.

Out of the four possible consequences that may happen in the output model (Fig. 2), we want to focus now on calculating the risk of having a fire. Noble et al. [25] present a formula for calculating it. More specifically, they describe the *MacArthur FFDI* index, which is a weather-based index derived empirically in south-eastern Australia. It indicates the probability of a fire starting, its rate of spread, intensity, and difficulty of suppression. This formula uses data from temperature, humidity, wind and rain. We will not go into what the values returned mean, since it is out of the scope of the paper, but we want to compare the values calculated by the exact model transformation (EMT) and an approximate model transformation (AMT).

$$FFDI = 2 * (0.987 * log(D) - 0.45 + 0.0338 * T + 0.0234 * V - 0.0345 * H)^e$$

In the formula, D stands for *Drought*, T for *Temperature*, V for *Wind* and H for *Humidity*. As for drought, it is calculated taking into account the data regarding rain.

To be able to create a transformation where the huge amount of data we have in our model is utilized, we want to calculate the probability for having fire risk every month. This is, for obtaining the probability in a certain month, we will use the data gathered in the previous month (by calculating the average value).

For selecting the samples, we have decided to use Systematic Sampling. In this technique, the sample is selected according to a random starting point and a fixed, periodic interval. This interval, called the sampling interval, is calculated by dividing the population size by the desired sample size. The reason for using this technique in this case study, as well as the concrete method for calculating the sample size [19], is the way data related to meteorological conditions evolve over time. For instance, if we approach summer, it is very likely that the temperature after ten days is higher than today's, and the intermediate values would normally range between these two values. The data is, consequently, normally distributed.

4.2 Implementation and Results

In both scenarios, the EMT and the AMT, we have to calculate the index 12 times – since we are considering all the months of the year. In our EMT, we have around 46730 (560768 divided by 12) points in time for each month. In our AMT, we want to make two experiments: calculate the index with a confidence level of 95% and a relative error of 3%, and with a confidence level of 99% and a relative error of 3%. According to the formula described in [19], the AMT only has to deal, each month, with 1043 and 1764 points in time in these situations, respectively.

As a proof-of-concept of our approach, we have implemented our experiment in Java/EMF. Thus, after obtaining the data from the Elora Station [2], we have converted it into a model conforming to the metamodel shown in Fig. 1 and generated the API for the metamodel in Java. Then, once we have loaded the model into memory, we have used the same implementation for obtaining the FFDI indexes for each month, where in the EMT we traverse the whole population of phenomena and in the AMT we use only the data returned by the systematic sampling, as explained above. Regarding the output metamodel, we create an element of type *FireRisk* for each month, where *day* and *time* indicate the month (first day of the year and time of the day of each month), and *probability* stores the computed FFDI index.

The results displayed in Table 1 show the differences between the three executions. As it can be read, the difference between the FFDI indexes calculated by the exact and the approximate model transformations are very similar. Furthermore, the gain in performance is huge: the first AMT is 49 times faster than the EMT, while the second AMT is 23 times faster. We have analyzed the results and calculated the relative error, obtaining a value of 0.06287% in the first AMT, what makes sense since we have used a confidence level of 95%. The second, and more accurate, AMT produces only an error of 0.0285%, at the expense of spending more time in the execution.

5 Related Work

To the best of our knowledge, this paper is the first work which deals with the development of AMTs for large models. In [3, 24], a model transformation intent catalog is

Table 1. Results from Exact MT (EMT) and Approximate MT (AMT).

		95% CL and 3% RE		**99% CL and 3% RE**	
Month	**FFDI in EMT**	**FFDI in AMT**	**Error**	**FFDI in AMT**	**Error**
January	0.36845	0.36778	0.00181	0.36445	0.01097
February	0.37482	0.40547	0.07559	0.36478	0.02751
March	0.36216	0.37434	0.03252	0.35931	0.00795
April	0.40994	0.43950	0.06727	0.38833	0.05565
May	0.59438	0.59420	0.00031	0.59081	0.00604
June	0.69891	0.76448	0.08577	0.69739	0.00217
July	0.73598	0.78531	0.06281	0.71403	0.03073
August	0.80324	0.83444	0.03739	0.80616	0.00362
September	0.76696	0.71483	0.07939	0.83423	0.08063
October	0.80842	0.73761	0.09599	0.79740	0.01381
November	0.77053	0.79436	0.02999	0.75324	0.02296
December	0.48258	0.49907	0.03304	0.47241	0.02154
Exec Time	0.25651	0.005154	–	0.011153	–

introduced which contains different transformation intents, where one of them is the *approximation* intent. According to the definition given in [24], a transformation "m_1 approximates another transformation m_2 if m_1 is equivalent to m_2 up to a certain error margin". As an example the Fast Fourier Transformation is given which is an approximation of the Fourier Transformation. This definition is in accordance with our idea of AMT. As future work, studying which other kinds of transformation intents may benefit from AMTs seems promising.

A related research area is concerned with model transformation by-example [23, 31, 33], which may be interpreted as an approximation of output models for new input models based on previously seen input/output model pairs. However, here the idea is not trading accuracy for response time, but reducing the development efforts of model transformations.

Concerning the reduction of the number of input elements to be read from input models to produce output models, two transformation strategies have been discussed in the past. First, if an output model already exists from a previous transformation run for a given input model, only the changes in the input model are propagated to the output model. Second, if only a part of the output model is needed by a consumer, only this part is produced while other elements are produced just-in-time. For the former scenario, *incremental* transformations [7, 20, 22, 26] have been introduced, while for the latter *lazy* transformations [29] have been proposed. In this paper, we proposed an orthogonal approach which does not rely on previous transformation runs and is able to produce an approximate answer independent of its consumption. However, combining AMTs with other query optimization techniques such as incremental execution, e.g., to store samples, seems a next logical step.

The approximation of computations has a long tradition in the database area in order to deal with very large data sets [10, 13]. Thus, several approximation techniques have emerged in the last decades. One technique in this respect is online query processing [17], which provides intermediate results already before the exact result is produced. Other techniques such as histograms, samples, and wavelets, are based on precomputed data synopses [13], i.e., synopses are constructed and stored for the complete data set prior to query time and used at query time to answer the queries. Most related to AMTs as presented in this paper are approximate queries based on sampling. For approximate queries the user specifies in addition to the query a certain error bound or time constraint for the query [1, 18]. Based on this meta-information, the database selects a sampling strategy or a pre-cached sample if available to answer the query.

To sum up, several approximation techniques are available for databases, but their adoption to models is challenging because of the different meta-languages (graphs vs. relations) and query languages (OCL vs. SQL) employed. Furthermore, approximate queries are still intensively studied for databases to make them useable in different scenarios as well as to support a wide range of different query types. Currently, different techniques support different query types, but supporting approximate queries for general and flexible queries is still limited [1]. In order to deal with current scalability challenges in MDE, it seems promising to further explore which kinds of approximation techniques fit best for certain MDE scenarios.

6 Conclusions and Future Work

In this paper we have introduced *Approximate Model Transformations*, which aim at producing target models that are accurate enough to provide meaningful and useful results in an efficient way, but without having to be fully correct. We have explained its basic notions and applied it successfully in a case study. AMTs provide a mechanism for being able to balance performance and accuracy in the realm of model transformations design and execution.

Of course, this paper presents an exploratory study which requires deeper investigations at all levels. In particular, we were interested in exploring the possibility of defining AMTs, and our initial results show that there are enough reasons to keep working on them. The work presented here does not pretend to be conclusive, or comprehensive, but to open the path for the model transformation community to start working on it.

The ideas presented in the paper may also be challenged, as potential threats may impact the internal and external validity [12] of the results showed here. For example, a potential threat to external validity is that the case study that has been utilized, even if it is a real-world example, does not cover all cases. For instance, our input data follows a normal distribution, so the function we have used for calculating the sample size [19], as well as the sampling strategy used—systematic sampling, are optimal for this case study. We need to study the situations in which the data come in different forms. As for internal validity, the main threat can be due to the way our approach has been implemented so far. We have used Java/EMF in our implementation. In fact, in this paper we have *simulated* the behavior of the approximate operators in Java/EMF and therefore the performance gains we have obtained may not be comparable as if the operators had been implemented in a model transformation language and the data had been distributed on different computing nodes.

There are several open issues that we plan to address next. In the first place, we would like to provide formal and precise specifications for our approximate operators, and integrate them in the ATL language. This would consist of extending the syntax of ATL so that these new operators are available, and also extending the ATL execution engine by implementing these operators. Secondly, we would like to add a new argument to the approximate operators, namely the time we want the operator to execute (such an approach is presented in [1] in the context of databases). The idea is to choose between this new operator or the two we have defined in this work (confidence interval and relative error) when calling a collection operator. Also, we need to study and develop methods for the appropriate design of AMTs. Although this will normally require a deep knowledge on the domain and the particular transformation scenarios, there is already a fair amount of work about the design of approximate and randomized algorithms [34] that could be applicable in this context. Last, but not least, in this work we have dealt with the approximation of values, i.e., numerical values stored in elements' attributes. However, we would also like to investigate how references can be approximated. For instance, if one element is referencing one million of different elements, how should we decide which ones to choose for the sample, and according to which criteria?

Acknowledgements. This work is partially funded by the European Commission under the ICT Policy Support Programme (grant no. 317859), by Research Project TIN2011-23795 and by Universidad de Málaga (Campus de Excelencía Internacional Andalucía Tech).

References

1. Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., Stoica, I.: BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In: Proc. of EuroSys. pp. 29–42 (2013)
2. Agricultural_Forest_Metereology_Group: Weather Records for the Elora Research Station, Elora, Ontario, Canada: Metereological data 2013 (2014), http://http://dataverse.scholarsportal.info/dvn/dv/ugardr
3. Amrani, M., Dingel, J., Lambers, L., Lúcio, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Towards a Model Transformation Intent Catalog. In: Proc. of AMT. pp. 3–8 (2012)
4. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proc. of VOLT. pp. 921–928 (2012)
5. van Amstel, M., Bosems, S., Kurtev, I., Pires, L.F.: Performance in Model Transformations: Experiments with ATL and QVT. In: Proc. of ICMT. pp. 198–212 (2011)
6. Azanza, M., Batory, D., Díaz, O., Trujillo, S.: Domain-specific Composition of Model Deltas. In: Proc. of ICMT. pp. 16–30 (2010)
7. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Proc. of MoDELS. pp. 76–90 (2010)
8. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: On the Concurrent Execution of Model Transformations with Linda. In: Proc. of BigMDE (2013)
9. Combemale, B., Thirioux, X., Baudry, B.: Formally Defining and Iterating Infinite Models. In: Proc. of MoDELS. pp. 119–133 (2012)
10. Cormode, G., Garofalakis, M.N., Haas, P.J., Jermaine, C.: Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Foundations and Trends in Databases 4(1-3), 1–294 (2012)

11. Cuadrado, J.S., de Lara, J.: Streaming Model Transformations: Scenarios, Challenges and Initial Solutions. In: Proc. of ICMT. pp. 1–16 (2013)
12. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting Empirical Methods for Software Engineering Research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer (2008)
13. Garofalakis, M.N., Gibbons, P.B.: Approximate Query Processing: Taming the TeraBytes. In: Proc. of VLDB (2001)
14. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: Proc. of ECMFA. pp. 221–235 (2011)
15. Guerra, E.: Specification-Driven Test Generation for Model Transformations. In: Proc. of ICMT. pp. 40–55 (2012)
16. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. 20(1), 5–46 (2013)
17. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: Proc. of SIGMOD. pp. 171–182 (1997)
18. Hu, Y., Sundara, S., Srinivasan, J.: Supporting Time-Constrained SQL Queries in Oracle. In: Proc. of VLDB. pp. 1207–1218 (2007)
19. Israel, G.D.: Determining Sample Size, University of Florida, Institute of Food and Agriculture Sciences (1992)
20. Johann, S., Egyed, A.: Instant and Incremental Transformation of Models. In: Proc. of ASE. pp. 362–365 (2004)
21. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
22. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: Proc. of ICMT. pp. 123–137 (2010)
23. Kessentini, M., Sahraoui, H.A., Boukadoum, M., Benomar, O.: Search-based model transformation by example. SoSyM 11(2), 209–226 (2012)
24. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model Transformation Intents and Their Properties. SoSyM pp. 1–35 (2014)
25. Noble, I.R., Gill, A.M., Bary, G.A.V.: Mcarthur's fire-danger meters expressed as equations. Australian Journal of Ecology 5(2), 201–203 (1980)
26. Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: Proc. of ICSE. pp. 562–572 (2012)
27. Sheth, A., Henson, C., Sahoo, S.S.: Semantic Sensor Web. IEEE Internet Computing 12(4), 78–83 (2008)
28. Tisi, M., Perez, S.M., Choura, H.: Parallel Execution of ATL Transformation Rules. In: Proc. of MoDELS 2013. pp. 656–672 (2013)
29. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: Proc. of MoDELS. pp. 32–46 (2011)
30. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Proc. of SFM. pp. 399–437 (2012)
31. Varró, D.: Model Transformation by Example. In: Proc. of MoDELS. pp. 410–424 (2006)
32. Wimmer, M., Burgueño, L.: Testing M2T/T2M Transformations. In: Proc. of MoDELS. pp. 203–219 (2013)
33. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: Proc. of HICSS (2007)
34. Zhu, Z.A., Misailovic, S., Kelner, J.A., Rinard, M.C.: Randomized accuracy-aware program transformations for efficient approximate computations. In: Proc. of POPL. pp. 441–454 (2012)

A query structured approach to model transformation

Hamid Gholizadeh, Zinovy Diskin, Tom Maibaum

McMaster University, Computing and Software Department, Canada
{ mohammh | diskinz | maibaum }@mcmaster.ca

Abstract. Model Transformation (MT) is an important operation in the domain of Model-Driven Engineering (MDE). While MDE continues to be further adopted in the design and development of systems, MT programs are applied to more and more complex configurations of models and relationships between them and grow in complexity. Structured techniques have proven to be helpful in design and development of programming languages. In this paper, using an example, we explain an approach in which MT specifications are defined in a structured manner, by distinguishing queries as their main building blocks. We call the approach *Query Structured Transformation* (QueST). We demonstrate that the contents of individual queries used in QueST to define a transformation are dispersed all over the entire corresponding MT definition in ETL or QVT-R. Our claim is that the latter two languages are less supportive of a structured approach than QueST. Finally we discuss the promising advantage of QueST in MT definition, and possible obstacles towards using it.

Keywords: Model Transformation, Query Structured Model Transformation, Formal Methods, Model Driven Engineering.

1 Introduction

As the adaption of Model-Driven Engineering (MDE) in the design and development of systems increases in industry, the complexity of Model Transformation (MT) programs —basic MDE operations which transform models to other models— also increases. Structured approaches already have proven to be successful in managing the complexity of systems; for example, the shift from programming with *goto* statements to the structured programming paradigm has proven to be a good design decision, which undoubtedly improved the quality of the software produced. Following this principle, we propose a *Query Structured Transformation* (QueST[1]) approach for defining model transformations which are translating source models to target models. QueST approach is originated from [2, 1], and its mathematical foundation has been developing for the past few years [3, 4]. In this paper, we explain QueST by introducing its structural

[1] The acronym is suggested by Sahar Kokaly, our colleague in the MDE group.

components using a well-known class-to-table example. The structural components of QueST are declarative queries that are used to define target element generation. For each individual target metamodel element, there exists one distinct query in QueST used to define the part of the source metamodel used to generate it. We exhibit the definition of a number of these queries in relation to an example, and explain their execution in QueST. Then, we demonstrate how these queries are dispersed in the body of the corresponding MT programs written in the Epsilon Transformation Language (ETL) [7], and in QVT-Relational [9]. We discuss the reasons for this phenomenon, and finally discuss the benefits of QueST for MT definitions.

The paper is organized as follows: in the next section we introduce the metamodels for the class-to-table example, and informally describe the transformation rules. In Section 3, we explain the structural components of the class-to-table example in QueST, and provide a mathematical definition for a number of these components. Then, we explain how the QueST engine would execute the MT definition. In Section 4, we briefly discuss program building blocks in QVT-R and ETL, and demonstrate the dispersal of the contents of the QueST components (i.e., queries) in the MT definitions corresponding to the same class-to-tables example in these languages. In Section 5, we briefly discuss the reasons for the query dispersals, and also promising advantages of using QueST. Section 6 concludes the paper and mentions potential future research.

2 Class diagram to database schema MT

Metamodels specify valid instances of domains, and are the focal point of MT definitions in QueST. We first briefly describe the source and target metamodels of the class-to-table example and then define the MT in QueST based on them.

2.1 Metamodels

Metamodels of the class diagram (CD) and database schema (DBS) specify the valid model spaces of the CDs and DBSs, respectively. Fig. 1(a) exhibits the CD metamodel. Each `class` contains some `attribute`s associated to it by `atts`; each attribute has a `type` and a multiplicity (`Lbound` and `Ubound`). Each `class` might inherit at most one class (`parent` arrow). `Association`s have multiplicity like `Attributes` and connect `src` classes to `trg` classes.

Fig. 1(b) shows the metamodel of the DBS. Each `Table` has at least one `Column` (see `cols` in the figure). Some columns are primary keys for the table (`pKeys`). A table might also have some foreign keys (`Fkey`). `fKeys` associate these foreign keys to tables. Each `FKey` refers to one table (`refs`) and some columns (`fCols`).

The constraints associated with the metamodels are exhibited using red labels with enclosing brackets. The multiplicities on the arrows are also constraints and therefore they are in red and are included inside brackets. Constraints can be written in any suitable language by interpreting squares as sets and arrows as binary relations. `[isAbstract]` specifies that `NamedElement` is abstract.

`[noLoop]` specifies that there is no loop in the inheritance hierarchy of classes. `[pKeysInCols]` states that the primary keys are columns of the same tables. `[FKeyColIsValid]` states that the columns to which each foreign key refers are subset of table columns of which they are a part (i.e., `fKeys;fCols` $\subset$ `cols`).

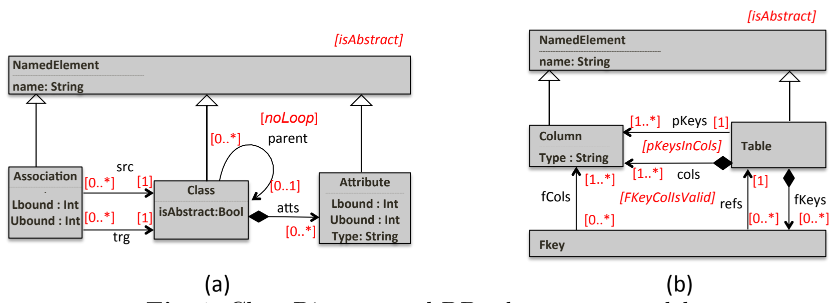


Fig. 1. Class Diagram and DB schema metamodels

2.2 Transformation rules

There are different ways to translate a CD to a corresponding DBS [5]. We propose the following rules as the transformation specification of the example in this paper. For each class we generate a table. Single-valued attributes (svAtts) —those with multiplicity of one or zero— of a class are translated to columns of the corresponding table. A table is generated for each multi-valued attribute —those with the multiplicity greater than one. These tables have two columns: one for keeping the attribute values and another for a foreign key referring to the table corresponding to the attribute containment class. Single-valued associations (svAssoci) are handled by foreign keys; for each svAssoci we create a column in the table corresponding to the source of the association. For each multi-valued association, we create a table with two foreign keys which refer to the source and target of the association, respectively. Inheritance is handled in a way similar to the single-valued associations.

Fig.2(a) shows a class diagram and Fig.2(b) shows its corresponding DB schema, following the rules specified above. In Fig.2(b), FK in parenthesis in front of a column indicates that the column is a foreign key and its outgoing arrow is referring to the table that this foreign key is referring to. As exhibited in the figure, the two tables `takes` and `teaches` are created due to the corresponding two multi-valued associations and the table `telephone` is created due to the corresponding multi-valued attribute.

3 QueST: Query Structured Transformation approach

In a typical MT language, like ETL or even QVT-R, the MT designer thinks in the following way: *how does each pattern in the source model produce a collection of elements in the target model?* But QueST assumes a different manner of thinking. The question that the MT designer is required to think about when

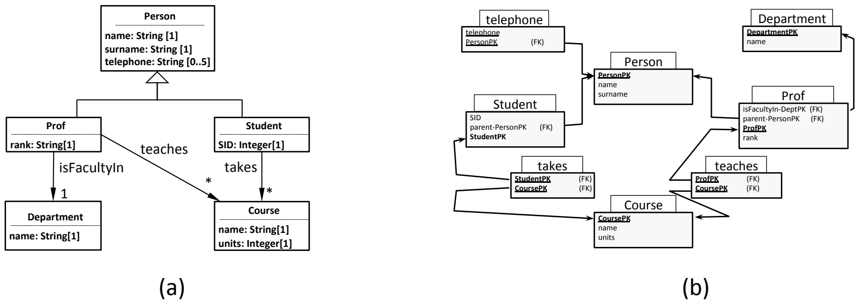


Fig. 2. A class diagram and its corresponding DB schema

starting to specify an MT in QueST is the following: *from which elements of the source model is each element of the target model generated?* This arises from the observation that the target model information is somehow hidden inside the source model and the model transformation program functionality's purpose is to reveal this information by manipulating the information inside the source model and creating the target elements out of the resulting information; for example, according to the MT rules specified in Sec. 2.2, tables in DBS are generated in three different cases: 1) for each class 2) for each multi-valued attribute and 3) for each multi-valued association. This can be specified by defining a query on the CD metamodel. The blue square with a diagonal corner named QTable in the right hand side of Fig.4 represents this query and Fig.3(a) exhibits this query definition in mathematical notation. The plus notation in the query means disjoint union. After this query definition, we associate the `Table` element in the target to this query (the green arrow from `Table` to `QTable` in Fig. 4).

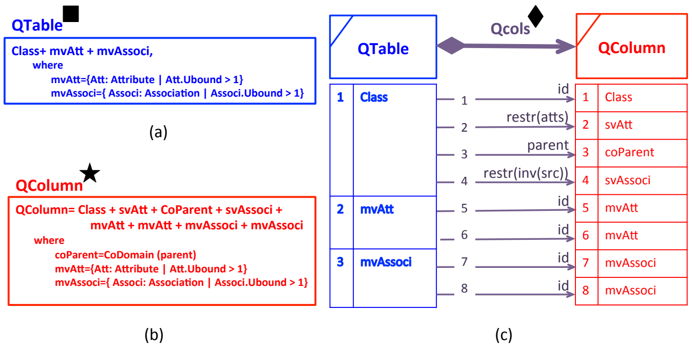

QTable
Class+ mvAtt + mvAssoci,
where
mvAtt={Att: Attribute | Att.Ubound > 1}
mvAssoci={ Associ: Association | Associ.Ubound > 1}
(a)
QColumn
QColumn= Class + svAtt + CoParent + svAssoci +
mvAtt + mvAtt + mvAssoci + mvAssoci
where
coParent=CoDomain (parent)
mvAtt={Att: Attribute | Att.Ubound > 1}
mvAssoci={ Associ: Association | Associ.Ubound > 1}
(b)
QTable
Qcols
QColumn
1 Class
2 mvAtt
3 mvAssoci
1 id
2 restr(atts)
3 parent
4 restr(inv(src))
5 id
6 id
7 id
8 id
1 Class
2 svAtt
3 coParent
4 svAssoci
5 mvAtt
6 mvAtt
7 mvAssoci
8 mvAssoci
(c)

Fig. 3. Query definitions

From the transformation specification, we need to figure out how the columns are generated, and continue to answer similar questions for all the other entities (including squares and arrows) in the target metamodel. This method of thinking is the cornerstone in writing an MT in QueST. Therefore, based on the MT specification, we write a query like the one shown in Fig. 3(b) for the `Column` entity and name it `QColumn`. This query is simply specifying all the possible ways leading to the generation of a column based on the MT specification rules. Then we associate the `Column` in the target metamodel to this query (see Fig. 4).

We apply the same method for the associations between the squares in the target metamodel. Hence we draw an arrow between the `QTable` and `QColumn` (see purple arrow called `Qcols` in Fig.4) and associate a query to this arrow. The query definition for this arrow is defining the relation between the elements of the `QTable` and `QColumn`. That is the disjoint union of all the arrows which are indexed from one to eight in Fig.3(c); `restr(atts)` is restriction of the `atts` relation over the `svAtt` co-domain; `inv(src)` is the inverse of the `src` relation, and `id` arrows are the identity relations over their domains. The ■ , ⧫ and ★ signs on the `Qtable`, `Qcol` and `QColumn` in this figure will be used for comparison purposes in Section 4.

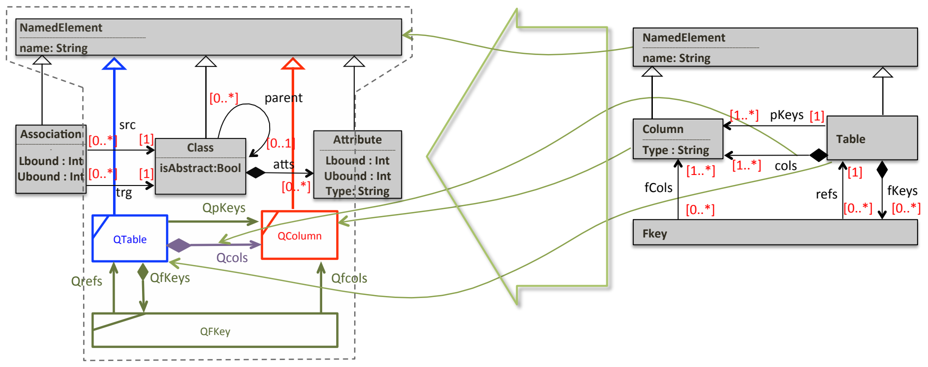


Fig. 4. CD to DBS MT definition in QueST

If we continue the above process of query definition and linking, for the other elements of the DBS metamodel, we will arrive at the structure shown in Fig.4. The entire figure shows the definition of the class-to-table MT in QueST. The new green elements called `QFKey`, `QpKeys`, `Qrefs`, `QfKeys`, and `Qfcols` are all new queries. Their definitions are not shown in the figure due to space limitations, but they are defined in a similar way to the queries defined in Fig. 3. We call the source metamodel with query annotations an *augmented* source metamodel. The augmented CD metamodel is shown in the left hand side of the Fig. 4. As it is shown by the dotted enclosing polygon in the figure, the structure of the target metamodel is somehow replicated in the source metamodel. The association links from the DBS to the CD metamodel show how the DBS metamodel is associated to the augmented CD metamodel. Not all the individual links are shown in the figure; instead a large blank green arrow is used to represent all the links. This linking from the target to the source is total.

From the MT definition in Fig. 4, it is seen that the building blocks of an MT definition in QueST are the queries on the source metamodel, and the links which associate the elements of the target metamodel to these queries (like `QTable`) or the source metamodel elements (like `NamedElement`). This provides a well-formed structure for the model transformation definition; the query definitions are encapsulated inside the squares and arrows, and are independent of each other, and tracing back (by the association links) from the target elements to the augmented metamodel shows how the transformation definition for each entity in the target metamodel is defined. In Section 4 we show how these queries are represented in ETL and QVT-R.

3.1 MT execution in QueST

For a given class diagram, like the one in Fig.2(a), the QueST engine first starts executing the queries of the augmented CD metamodel over the given class diagram. The order in which the queries are executed does not matter semantically, so the execution of the queries can be scheduled in any order by the QueST execution engine. This enables the engine implementation with the possibility of applying any appropriate optimization mechanism over the execution of the queries at the implementation level.

The collected elements from the execution of each query are then typed over that query. For example, Fig 5 shows the elements generated by execution of the `QTable` query that are typed over it (see `:QTable` square in Fig 5). The incoming dotted arrows to the `:QTable` square show from where each element of the square is coming. After all queries are executed, the next step for the QueST engine is to produce the taget elements. This is done by replicating the elements collected by the queries and changing their types according to the association links in the MT definition; for example, for the `QTable` query, the engine first duplicates all the elements collected inside the `:QTable` square and changes their types to `Table`, since the `Table` entity is associated to the `QTable` by a link in the MT definition, as in the previous section. The outgoing dotted arrows from `:QTable` connect the elements to their replicated versions which are all typed over the `Table` element (or simply are tables in DBS).

The process described for the `QTable` query in the previous paragraph continues for all other queries and its completion leads to the generation of the target model shown in Fig 2(b) .

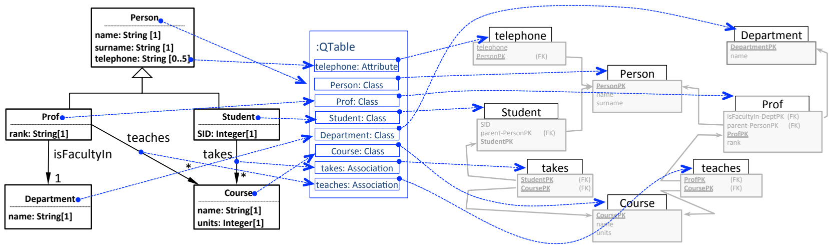


Fig. 5. QueST engine executing QTable query and generating Tables

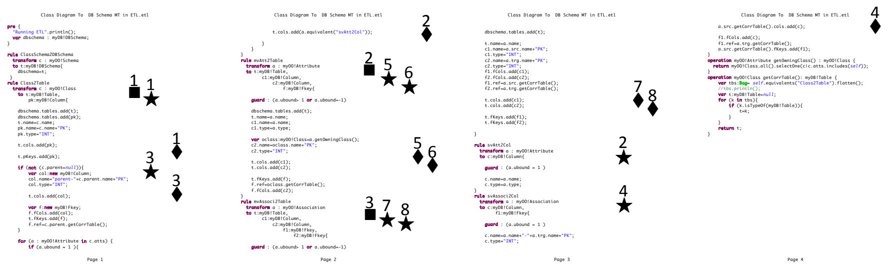
4 Dispersal of queries in ETL and QVT-R

We have defined the very same MT transformation described in Section 2.2 with ETL, and also with QVT-R. In this section, we briefly explain the structures of the MT definitions is each language, and by marking each transformation code, we demonstrate the wide dispersal of the contents of structural components (i.e., queries) of QueST in these definitions.

4.1 Query dispersal in ETL

An MT in ETL is specified by a set of rules. Each rule defines how an element of a specific type is translated to one or more elements (not all necessarily having the same type) in the target model. The rules might have guard expressions that constrain their application.

Fig. 6 shows the code for defining the class-to-table example in ETL. Intentionally, the code font size used is very small in the figure, since we will not discuss in detail each part of the definition, and instead, we only take into account the entire transformation definition to show the dispersal of the QueST queries all over it. The parts in Fig. 6 that correspond to the queries in Fig.3 are marked: 1) ■ marking the parts corresponding to the generation of tables (`QTable` query); 2) ★ marking the parts corresponding to the generation of columns (`QColumn` query) and 3)♦ marking the parts which associate the columns to the tables (`Qcols` Query). The numbers above the markings correspond to the numbers in Fig. 3(c) for each marking; for example, $\overset{2}{■}$ refers to the `mvAtt` component of the `QTable`, and $\overset{3}{♦}$ and $\overset{3}{★}$ refer to the `parent` arrow of `Qcols`, and the `coParent` component of `QColumn`, respectively.



```
Class Diagram To  DB Schema MT in ETL.etl

pre {
  "Running ETL".println();
  var dbschema : myDB!DBSchema;
}

rule ClassSchema2DBSchema
  transform c : myOO!Schema
  to t:myDB!DBSchema{
    dbschema=t;
 }
rule Class2Table
  transform c : myOO!Class
    to t:myDB!Table,
        pk:myDB!Column{

    dbschema.tables.add(t);
    dbschema.tables.add(pk);
    t.name=c.name;
    pk.name=c.name+"PK";
    pk.type="INT";

    t.cols.add(pk);

    t.pKeys.add(pk);

    if (not (c.parent=null)){
        var col:new myDB!Column;
        col.name="parent-"+c.parent.name+"PK";
        col.type="INT";

        t.cols.add(col);

        var f:new myDB!Fkey;
        f.fCols.add(col);
        t.fKeys.add(f);
        f.ref=c.parent.getCorrTable();
    }

    for (a : myOO!Attribute in c.atts) {
        if (a.ubound = 1 ){

Page 1
```

```
Class Diagram To  DB Schema MT in ETL.etl

            t.cols.add(a.equivalent("svAtt2Col"));

        }
    }
}
rule mvAtt2Table
  transform a : myOO!Attribute
  to t:myDB!Table,
        c1:myDB!Column,
            c2:myDB!Column,
                f:myDB!Fkey{

    guard : (a.ubound> 1 or a.ubound=-1)

    dbschema.tables.add(t);
    t.name=a.name;
    c1.name=a.name;
    c1.type=a.type;

    var oclass:myOO!Class=a.getOwningClass();
    c2.name=oclass.name+"PK";
    c2.type="INT";

    t.cols.add(c1);
    t.cols.add(c2);

    t.fKeys.add(f);
    f.ref=oclass.getCorrTable();
    f.fCols.add(c2);
}
rule mvAssoci2Table
  transform a : myOO!Association
  to t:myDB!Table,
        c1:myDB!Column,
            c2:myDB!Column,
                f1:myDB!Fkey,
                    f2:myDB!Fkey{

    guard : (a.ubound> 1 or a.ubound=-1)

Page 2
```

```
Class Diagram To  DB Schema MT in ETL.etl

    dbschema.tables.add(t);

    t.name=a.name;
    c1.name=a.src.name+"PK";
    c1.type="INT";
    c2.name=a.trg.name+"PK";
    c2.type="INT";
    f1.fCols.add(c1);
    f2.fCols.add(c2);
    f1.ref=a.src.getCorrTable();
    f2.ref=a.trg.getCorrTable();

    t.cols.add(c1);
    t.cols.add(c2);

    t.fKeys.add(f1);
    t.fKeys.add(f2);

}

rule svAtt2Col
  transform a : myOO!Attribute
  to c:myDB!Column{

    guard : (a.ubound = 1 )

    c.name=a.name;
    c.type=a.type;
}
rule svAssoci2Col
  transform a : myOO!Association
  to c:myDB!Column,
        f1:myDB!Fkey{

    guard : (a.ubound = 1 )

    c.name=a.name+"-"+a.trg.name+"PK";
    c.type="INT";

Page 3
```

```
Class Diagram To  DB Schema MT in ETL.etl

    a.src.getCorrTable().cols.add(c);

    f1.fCols.add(c);
    f1.ref=a.trg.getCorrTable();
    a.src.getCorrTable().fKeys.add(f1);
}
operation myOO!Attribute getOwningClass() : myOO!Class {
  return myOO!Class.all().selectOne(c|c.atts.includes(self));
}
operation myOO!Class getCorrTable(): myDB!Table {
    var tbs:Bag= self.equivalents("Class2Table").flatten();
    //tbs.println();
    var t:myDB!Table=null;
    for (k in tbs){
        if (k.isTypeOf(myDB!Table)){
            t=k;
        }
    }
    return t;
}

Page 4
```

Fig. 6. ETL code for the CD to DBS transformation

As the figure shows, each marking sign is dispersed over the entire code. This means that different components of the queries in Fig 3 are dispersed all over the code in ETL; for example the eight component of the `Qcols` which are indexed from one to eight are scattered throughout the entire definition and among the different rules.

We avoided marking all the queries that are generating the target elements in the definition of Fig. 6 to keep the figure simple. However, by repeating the marking procedure for all of the other queries, we would get a similar dispersal of their content over the entire code. One immediate consequence of these dispersals is the difficulty that occurs during the debugging process in the development of the model transformation; since the user needs to check different parts of the code, if he gets some errors regarding generation of specific element in the target.

4.2 Query dispersal in QVT-R

An MT definition in QVT-R is composed of a set of *top* and *non-top* relations. The difference is that the latter relations are invoked by the former ones. Each relation definition specifies how some elements in the source model are related to some elements in the target model. There are *when* and *where* clauses which act as pre- and post-conditions for the execution of the corresponding relation [8]. At the time QVT-R engine executes an MT definition, it enforces that all the defined top relations hold true, by creating missing elements in the target.

We wrote a transformation in QVT-R for the same class-to-table example, and perform the same analysis over the code as we did in the previous section for the ETL definition. Fig. 7 shows the definition and the marking; the semantics of the markings are identical to what we described before. It is seen from the figure that, similar to the ETL code, the queries for the generation of the target model elements are distributed all over the code.

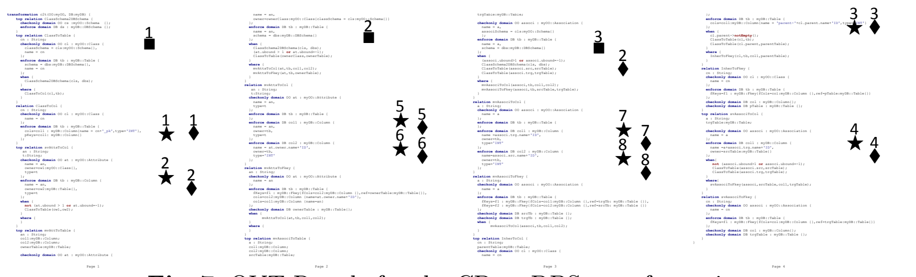


Fig. 7. QVT-R code for the CD to DBS transformation

5 Discussion

This section is divided into two parts: we first discuss the reasons that cause the scattering of each individual query in QueST over the entire MT definitions by ETL and by QVT-R; then, we discuss the promising advantages of QueST from different perspectives.

5.1 Why query dispersal happens in ETL and QVT-R

It might be argued that what are presented in Section 4, as the definitions of the class-to-table example in QVT-R, and in ETL are subjective, in the sense that there might be different implementations for the example by these languages, such that the demonstrated query dispersal are prevented. We believe that the scattering of the QueST queries would happen in any implementation, because of the three reasons briefly discussed below:

One to many and many to many associations. In ETL, each rule associates one source metamodel element to many target metamodel elements. In QVT-R, each relation associates many source metamodel elements to many target metamodel elements. Therefore one target element can be referenced in many rules/relations in a transformation definition in these languages; this means that the queries generating the elements of a specific type are spread between different rules/relations.

Arrows are secondary elements. References to arrows in MT definitions in ETL and QVT-R happen by means of nodes; queries only define nodes, and arrow definitions are implicit inside these queries. More concretely, it is not possible to define an arrow as a target of an ETL rule (i.e., as a rule header parameter), or as a domain of a relation in QVT-R. This means that if the queries generating nodes are dispersed, then the queries for generating the arrows which are referenced by these nodes will also be dispersed. The ♦ marking signs appearing

everywhere close to the ★ signs in Fig. 7 show this phenomenon.
Flattening the graphical structure. ETL and QVT-R are textual languages, while as it may be seen from the Fig. 4, MT definitions contain graphical constructs. A representation of a graphical construct in a textual format causes a scattering of references to the graphical elements, inside the representation; for example a node with several incoming edges in a graph would be inevitably referenced in different places in the graph's textual representation.

5.2 Promising advantages of QueST

MT design and development. As is shown in Section 3, QueST provides well structured definitions by encapsulating queries inside squares and arrows, i.e., the first class elements of the metamodeling language. All the target elements of a specific type (type could be an arrow or a square) are generated by one, and only one, query. We believe that this is helpful in development and debugging of MT definitions, because it follows the well-known principle of *separation of concerns*, where the concern is the production of elements of specific type. Further, each query definition is *fairly* independent in QueST, i.e., square queries are independent, and arrow queries are only dependent on their corresponding source and target queries; Hence, QueST's structural construct, the query, suitably provides a pattern to decompose complex MT definition tasks into small fairly independent definitions.
Semantic foundations. The mathematical foundation of the QueST approach is already discussed in some other work [2–4, 6]. This ensures that there is a clear and formal understanding of QueST. This would prevent some ambiguity and semantic issues like the ones investigated for the QVT-R specification [10]. Furthermore, its formal foundation provides a context to validate and formally verify MT definitions.
Flexibility in query language. Theoretically, any query language can be used for defining the queries in QueST. The expressive power of QueST depends on the expressive power of the chosen query language. We insist that the chosen query language should consider the arrows as equally as important as the nodes; the experiments of using QueST for MT definitions have shown that defining the square queries are easy in some query languages like OCL, but defining arrow queries are fairly complicated, because they necessarily include many references to the source and target of the arrow.
Declarative vs. imperative. QueST is declarative: the definitions of its structural building blocks (i.e., queries) provide specifications rather than implementations for the generation of the target elements. Further, the queries could be executed in any order in QueST. The advantages of declarative approaches in programming languages is less debatable now; even though the traditional challenges of training MT programers to think declaratively might still be an obstacle, considering the number of people who are trained to write queries declaratively, in the database community, it might be plausible enough to follow a similar pathway in the MT community as well.

6 Conclusion

From one perspective, the intent of MT programs is to collect information from the source model by executing some queries, and, then, to build up the target model elements. Formalizing this procedure suggests a structural pattern for model transformation which we call QueST. In QueST, the MT definer should think in a target-oriented manner, in the sense that he should define a query for each individual element in the target metamodel, during the MT development process. These queries define the generation of target model elements, and are encapsulated inside the fairly independent components which make up the structural building blocks of an MT definition in QueST. We used a well-known class-to-table example to explain these structural components (i.e., queries) and their definitions. We also demonstrated that the contents of these queries are necessarily dispersed all over the entire MT definitions written with ETL and QVT-R. Subsequently, we discussed how the dispersal of each query is not a specific property of the implemented examples and could be generalized to any programs written in QVT-R and ETL. Finally, we discussed the advantages of QueST from different perspectives. A deeper examination and evaluation of QueST, regarding the aspects discussed in the previous section, is the subject for our future work.

References

1. Z. Diskin and J. Dingel. A metamodel independent framework for model transformation: Towards generic model management patterns in reverse engineering. In *ATEM*, 2006.
2. Zinovy Diskin. Mathematics of generic specifications for model management. In *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.
3. Zinovy Diskin. Model synchronization: Mappings, tiles, and categories. In *Generative and Transformational Techniques in Software Engineering III*, pages 92–165. Springer, 2011.
4. Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki. Intermodeling, Queries, and Kleisli Categories. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2012.
5. David W Embley and Bernhard Thalheim. *Handbook of conceptual modeling: theory, practice, and research challenges*. Springer, 2012.
6. Hamid Gholizadeh. Towards declarative and incremental model transformation. In *DocSymp@ MoDELS*, pages 32–39, 2013.
7. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.
8. Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.
9. OMG, http://www.omg.org. *OMG. MOF2.0 query/view/transformation (QVT) version 1.1*, 2009.
10. Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

Towards Verified Java Code Generation from Concurrent State Machines

Dan Zhang*[1], Dragan Bošnački[1], Mark van den Brand[1], Luc Engelen[1], Cornelis Huizing[1], Ruurd Kuiper[1], and Anton Wijs **[2]

[1] Eindhoven University of Technology, The Netherlands
[2] RWTH Aachen University, Germany

Abstract. We present work in progress on, verified, transformation of a modeling language based on communicating concurrent state machines, SLCO, to Java. Some concurrency related challenges, related to atomicity and non-standard fairness issues, are pointed out. We discuss solutions based on Java synchronization concepts.

1 Introduction

Model-Driven Software Engineering (MDSE) is gaining popularity as a methodology for developing software in an efficient way. In many cases the models can be verified which leads to higher quality software. In MDSE, an abstract model is made increasingly more detailed through model-to-model transformations, until the model can be transformed to source code. This allows detecting defects in the early stages of the development, which can significantly reduce production costs and improve end product quality.

One of the challenges in MDSE is maintaining correctness during the development. The correctness of model-to-model transformations is one of the research topics of our group [6–8]. In this paper the correctness of model-to-code transformations is addressed. We investigate automated Java code generation from models in the domain specific language Simple Language of Communicating Objects (SLCO) [1]. SLCO was developed as a small modeling language with a clean manageable semantics. It allows modeling complex embedded concurrent systems. SLCO models are collections of concurrent objects. The dynamics of the objects is given by state machines that can communicate via shared memory (shared variables in objects) and message passing (channels between objects).

In this paper, we present the current status of our work and how we envision its continuation. We have defined a transformation from SLCO to Java code,

* This work was sponsored by the China Scholarship Council (CSC)
** This work was done with financial support from the Netherlands Organisation for Scientific Research (NWO).

and discuss in this paper the main complications as regards specifically efficient communication via shared variables and channels. We have started proving that this transformation is correct, i.e. that the semantics of SLCO models is preserved under some assumptions. This reasoning requires tackling non-standard fairness issues involving the built-in fairness assumptions in Java.

2 SLCO and its transformation to Java

SLCO In SLCO, systems consisting of concurrent, communicating objects can be described using an intuitive graphical syntax. The objects are instances of classes, and connected by channels, over which they send and receive signals. They are connected to the channels via their ports.

SLCO offers three types of channels: synchronous, asynchronous lossless, and asynchronous lossy channels. Furthermore, each channel is suited to transfer signals of a predefined type.

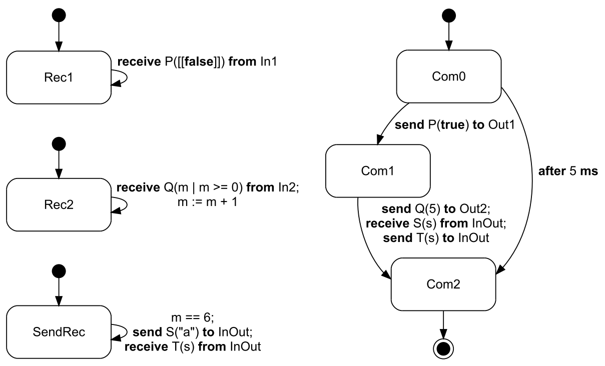


Fig. 1. Behaviour diagram of an SLCO model

The behaviour of objects is specified using state machines, such as in Figure 1. As can be seen in the figure, each transition has a source and target state, and a list of statements that are executed when the transition is fired. A transition is enabled if the first of these statements is enabled. SLCO supports a variety of statement types. For communication between objects, there are statements for sending and receiving signals. The statement **send** $T(s)$ **to** *InOut*, for instance, sends a signal named T with a single argument s via port *InOut*. Its counterpart **receive** $T(s)$ **from** *InOut* receives a signal named T from port *InOut* and stores the value of the argument in variable s. Statements such as **receive** $P([[$**false**$]])$ **from** *In1* offer a form of conditional signal reception. Only those signals whose argument is equal to **false** will be accepted. There is also a more general form of conditional signal reception. For example, statement **receive** $Q(m \mid m \geq 0)$ **from** *In2* only accepts those signals whose argument is at least equal to 0. Boolean expressions, such as $m == 6$, denote statements that block until the expression holds. Time is incorporated in SLCO by means of delay statements. For example, the statement **after** 5 **ms** blocks until 5 ms have passed. Assignment statements, such as $m := m + 1$, are used to assign values to variables. Variables either belong to an object or a state machine. The variables that belong to an object are accessible by all state machines that are part of the object.

Transformation from SLCO to Java The transformation from an SLCO model to Java is performed in two stages. First the textual SLCO model is automatically transformed into an intermediate model, which is then translated to Java. Recently we created a platform for the second stage in the Epsilon Generation Language (EGL) [2], tailored for model-to-text transformation. The execution of the generated Java code should reflect the semantics of SLCO. In the sequel we present some important parts of the end-to-end transformation from SLCO models to Java code.

State Machines Since SLCO state machines represent concurrent processes, each state machine is mapped to a different thread in the Java implementation.

We use switch statements to represent the behavior of the state machine (Listing 1.1). Each case corresponds to one state of the state machine and comprises a sequence of statements corresponding to the actions of a related outgoing state transition. If a state has more than one transition, nested switch statements are added in the generated code with a separate case for each transition.

Listing 1.1. Part of generated code for the state machine

```java
currentState = "Com0";
while(true){
  switch(currentState){
    case "Com0":
      String transitions[] = {"Com02Com1","Com02Com2"};
      ...
      int idx = new Random().nextInt(transitions.length);
      String nextTransition = transitions[idx];
      ...
      switch(nextTransition){
        case "Com02Com1":
        ...//check whether the transition Com02Com1 is enabled or not
      }
    case "Com1":
    ...
  }
}
```

Shared Variables Since shared variables can be accessed and modified by multiple state machines in one object, we need to consider synchronization constructs. In our current implementation we use one writing lock and one reading lock to control the access for all shared variables in one object. Both the reading and writing locks operate in fair mode, using an approximate arrival-order policy. A boolean expression statement using a reading lock is shown in Listing 1.2.

Listing 1.2. Part of generated code for the Boolean expression statement

```java
boolean isExecutedExpression = false;
while(!isExecutedExpression){
  r.lock();
  try{
    if((m.value==6)){
      isExecutedExpression = true;
    }
  } finally { r.unlock(); }
}
...
```

Channels Because lossy channels in SLCO are an undesired aspect of physical connections, these are not transformed to Java; hence the framework just supports the synchronous and asynchronous lossless channel.

SLCO's asynchronous channels have buffer capacity 1. The sender/receiver blocks when the channel is full/empty - this is provided by a BlockingQueue with a buffer capacity of 1 (from the package java.util.concurrent). In case of conditional reception from an SLCO channel, the element will only be consumed if the value satisfies the condition. This requires that the head of the queue is inspected without taking the element, which is provided by `BlockingQueue`.

SLCO's synchronous channels enable handshake-like synchronization between state machines. Our current implementation again uses the BlockingQueue, with additional ad hoc synchronization code.

In Listing 1.3 we give the generated Java code for sending a signal message of state machine Com via port InOut as shown in Fig. 1. Notice that we add the wait() method of Java to synchronize sending and receiving parties, if the channel between two state machines is synchronous.

Listing 1.3. Part of generated code of channels for sending signal message

```java
synchronized(port_InOut.channel.queue){
port_InOut.channel.queue.put(new SignalMessage("T",new Object[]{s}));
if(port_InOut.channel.isSynchronousChannel){
  port_InOut.channel.queue.wait();
}...
}
```

3 Challenges

The challenges in the transformation from SLCO to Java as well as in the verification of it mainly originate from the differences between the modeling-oriented primitives in SLCO and their implementation-oriented counterparts in Java.

Shared variables – atomicity In SLCO assignments where multiple shared variables are involved are atomic, therefore we use Java variables together with locks to guard the assignments. Instead of the built-in locks we use ReentrantReadWriteLock from the package java.util.concurrent.locks with concurrent read access, which improves performance. Furthermore, in SLCO conditions on a transition may involve several shared variables. The aforementioned package enables to create a condition object for each condition on a transition, that can be used to notify waiting processes, which gives better performance than checking with busy-waiting. Our current implementation uses ReentrantReadWriteLock, but with a simple busy wait to simulate the SLCO blocking. Using the condition objects whilst maintaining the SLCO semantics is the next step.

Channels – synchronization Other synchronization primitives from the package java.util.concurrent are under investigation to replace the current ad hoc synchronization code.

Choice between transitions – external/internal In case of several possibly blocking outgoing transitions of a state in SLCO, the state machine will only block if all outgoing transitions are blocked. (This applies to shared variable access as well as channel operations.) In a naive implementation, however, the disabledness of a transition can only be assessed by initiating its execution and then blocking, and waiting for the transition becoming enabled. Essentially, this means that an external choice is turned into an internal choice, which is undesired. Here we face a similar challenge as with conditions or assignments using several shared variables. This may be solved in a similar manner, namely with a condition object dedicated to the state and cooperating processes that notify the waiting process when one of the reasons for blocking might have changed.

We have solutions for specific cases and are investigating a generic solution for all transitions that involve blocking and is orthogonal to other concurrency aspects like conditional reading from a channel, synchronization of channels, etc.

Fairness – interleaving, resolving conflict The SLCO specification approach raises non-standard fairness issues. In early usage of SLCO, no fairness was assumed for SLCO specifications. If properties depended upon fairness, this meant taking this into account for the implementation. The challenge is to advance the formal rigor of the SLCO-approach by formally expressing fairness and have the generated code together with the JVM ensure this fairness.

We use an interleaving semantics for SLCO with weak fairness: if at some time point a transition becomes continuously enabled, this transition will at some later time point be taken: in linear time temporal logic $\Box(\Box enabled(t) \rightarrow \Diamond taken(t))$.

Because the granularity in Java is much finer than in SLCO, more progress is enforced by weak fairness in SLCO than in Java. Therefore we need stronger fairness in Java. We aim to achieve this through a combination of fairness in scheduling threads, obtainable by choosing the right JVM, and fair locks, obtainable from the package java.util.concurrent.locks.

Verification We aim to partially verify the transformation. Our first approach is that the generated code uses generic code that we provide as a library of implementations for, e.g., channels. We annotate and verify this generic code. To deal with pointers and concurrency in Java we will use Separation Logic [3] in combination with the tool VeriFast [5] as we have applied to a preliminary version of this research [4]. The report indicates that this is feasible. Our research aims to incorporate more advanced library code from the `java.util.concurrent` packages. A second, complementary, approach deals with specific code generated in the transformation. From the SLCO model we generate annotation along with the code, which captures what the code should satisfy to conform to the SLCO semantics – whether it does is then verified using the techniques described above.

4 Conclusions

In the context of automated Java code generation we identified several challenges involving shared memory communication and fairness. Our initial findings imply that the verification of the generic Java code implementing shared memory communication for safety properties is feasible using separation logic. As one of the anonymous reviewers observed, it might be useful to identify Java patterns for correctly capturing concurrent model semantics. Besides code generation, this can be used as a basis for developing efficient simulation, formal verification and other analysis tools.

References

1. Engelen, L.: From Napkin Sketches to Reliable Software. Ph.D. thesis, Eindhoven University of Technology (2012)
2. Kolovos, D., Rose, L., Garcia-Dominguez, A., Page, R.: The Epsilon Book (2013), checked 17/07/2014
3. Reynolds, J.C.: An Overview of Separation Logic. In: Meyer, B., Woodcock, J. (eds.) VSTTE. Lecture Notes in Computer Science, vol. 4171, pp. 460–469. Springer (2005)
4. Roede, S.: Proving Correctness of Threaded Parallel Executable Code Generated from Models Described by a Domain Specific Language. Master's thesis, Eindhoven University of Technology (2012)
5. Smans, J., Jacobs, B., Piessens, F.: Verifast for Java: A tutorial. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming, Lecture Notes in Computer Science, vol. 7850, pp. 407–442. Springer (2013)
6. Wijs, A.: Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In: FACS'13. Lecture Notes in Computer Science, vol. 8348, pp. 348–368. Springer (2013)
7. Wijs, A., Engelen, L.: Efficient Property Preservation Checking of Model Refinements. In: TACAS'13. Lecture Notes in Computer Science, vol. 7795, pp. 565–579. Springer (2013)
8. Wijs, A., Engelen, L.: REFINER: Towards Formal Verification of Model Transformations. In: NFM'14. Lecture Notes in Computer Science, vol. 8430, pp. 258–263. Springer (2014)

Towards Rigorously Faking Bidirectional Model Transformations

Christopher M. Poskitt[1]*, Mike Dodds[2], Richard F. Paige[2], and Arend Rensink[3]

[1] Department of Computer Science, ETH Zürich, Switzerland
[2] Department of Computer Science, The University of York, UK
[3] Department of Computer Science, University of Twente, The Netherlands

Abstract. Bidirectional model transformations (*bx*) are mechanisms for automatically restoring consistency between multiple concurrently modified models. They are, however, challenging to implement; many model transformation languages not supporting them at all. In this paper, we propose an approach for automatically obtaining the consistency guarantees of *bx* without the complexities of a *bx* language. First, we show how to "fake" true bidirectionality using pairs of unidirectional transformations and inter-model consistency constraints in Epsilon. Then, we propose to automatically verify that these transformations are consistency preserving—thus indistinguishable from true *bx*—by defining translations to graph rewrite rules and nested conditions, and leveraging recent proof calculi for graph transformation verification.

1 Introduction and Motivation

Model transformations are operations for automatically translating models conforming to one language (i.e. a metamodel) into models conforming to another, in a way that maintains some sense of consistency between them. At their most basic, model transformations are unidirectional: given a source model (e.g. some high-level yet user-modifiable view of a system), they generate a target model (perhaps a lower-level view, such as code) whose data is "consistent" with the source, in a sense that is either left implicit, or captured by textual constraints or an inter-model consistency relation.

Many situations arise where the source and target models may *both* be modified by users in concurrent engineering activities, e.g. when integrating parts of systems that are modelled separately but must remain consistent. Bidirectional model transformations (*bx*) [5,17] are a mechanism for automatically restoring inter-model consistency in such a scenario; in particular, *bx* simultaneously describe transformations in both directions—from source to target and target to source—with their compatibility guaranteed by construction [5].

While this advantage is certainly an attractive one, *bx* are challenging to implement on account of the inherent complexity that they must encode. Model transformation languages supporting them often do so with conditions: some require that *bx* are bijective (e.g. BOTL [3]), essentially restricting their use to models presenting identical data in

* Received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

different ways, whereas others require users to work with specific formalisms such as triple graph grammars (e.g. MOFLON [1]). The QVT-R language—part of the OMG's Queries, Views, and Transformations standard—allows *bx* to be expressed, but suffers an ambiguous semantics [18] and limited tool support (the most successful ones often departing from the original semantics [11]). Moreover, many modern transformation languages do not provide any support for *bx* (e.g. ATL [9]), meaning that users must express them as two separate unidirectional transformations. While this seems a practical workaround, it comes with the major risk that the compatibility of the transformations might not be maintained over time.

A trade-off between the benefits (but complexity) of *bx* and the practicality (but possible incoherence) of unidirectional transformations can be achieved in Epsilon, a platform of interoperable model management languages. Epsilon has languages supporting the specification of unidirectional transformations in either a rule-based (ETL), update-in-place (EWL), or operational (EOL) [12] style. Furthermore, it provides an inter-model consistency language (EVL [10]) that can be used to express and evaluate constraints between models conforming to different metamodels. With these languages together, *bx* can be "faked" in a practical way, by: (1) defining pairs of unidirectional transformations for separately updating the source and target models; and (2) defining "consistency" via inter-model constraints in EVL, the violation of which will trigger appropriate transformations to restore consistency.

Although this process gives us a means of checking consistency and automatically triggering a transformation to restore it, we lack the important guarantee that *bx* give us: the compatibility of the transformations. It might be the case that after the execution of one transformation, the other does not actually restore consistency, leading to further EVL violations. How do we check for, and maintain, compatibility?

We aim to address this shortcoming and obtain the guarantees of *bx* without the need for *bx* languages. Instead, we will use *rigorous* proof techniques to verify that faked *bx* are consistency preserving, and thus indistinguishable to users from true *bx*. To this end, we propose to apply techniques from graph transformation verification. Given a faked *bx* in Epsilon, we will model the unidirectional transformations as graph transformation rules, and EVL constraints as nested graph conditions [7]. Then, by leveraging graph transformation proof calculi [8,14,15] in a weakest precondition style, we aim to automatically prove compatibility of the unidirectional transformations with respect to the EVL constraints. Furthermore, we aim to exploit the model checker GROOVE [6] to automatically search for counterexamples when consistency preservation does not hold.

The overarching goal of our work is to achieve the ideal that Stevens [17] contemplated in her survey of *bx*: that *"if a framework existed in which it were possible to write the directions of a transformation separately and then check, easily, that they were coherent, we might be able to have the best of both worlds."*

2 Example *bx* in Epsilon: Class Diagrams to Databases

To illustrate the ideas of our proposal, we recall a common model transformation problem that concerns the consistency of class diagram and relational database models (CD2RDBM). Class diagram models conform to a simple language describing familiar object-oriented concepts (e.g. classes, attributes, relationships), whereas relational database models conform to a language describing how databases are constructed (e.g. tables, columns, primary keys). Here, consistency is defined in terms of a correspondence between the data in the models, e.g. every table n corresponds to a class n, and every column m corresponds to an attribute m. Figure 1 contains two simple models that are consistent in this sense (we omit the metamodels for lack of space).

Users of the models should be able to create new classes (or tables) whilst maintaining inter-model consistency. A *bx* would be well suited for this: upon the creation of a new class (resp. table), a table (resp. class) should be created with the same name to restore consistency. We can fake this simple *bx* in Epsilon with a pair of unidirectional transformations (one for updating the class diagram model, one for updating the relational database) and a set of EVL constraints. For the former, we can use the Epsilon Wizard Language (EWL) to define a pair of update-in-place transformations, `AddClass` and `AddTable` (for simplicity, here we assume the new class/table name `newName` to be pre-determined and unique, but Epsilon does support the capturing and sharing of such data between wizards).

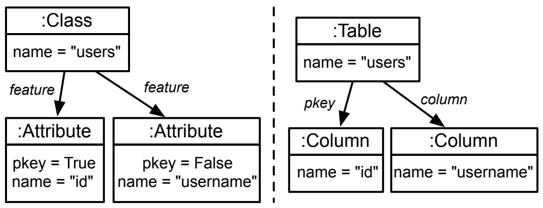


Fig. 1. Two consistent CD and RDB models

```
wizard AddClass {
 do {
   var c: new Class;
   c.name = newName;
   self.Class.all.first().contents.add(
        c);
 }}
```

```
wizard AddTable {
 do {
   var table: new Table;
   table.name = newName;
   self.Table.all.first().contents.add(
        table);
 }}
```

Using the Epsilon Validation Language (EVL), we express the relevant notion of inter-model consistency: that for every class n, there exists a table named n (and vice versa). If one of the constraints is violated, Epsilon can automatically trigger the relevant transformation to attempt to restore consistency. For example, after executing the transformation `AddClass`, the constraint `TableExists` will be violated, indicating that the transformation `AddTable` should be executed to restore consistency.

```
context OO!Class {
 constraint TableExists {
   check : DB!Table.all.select(t|t.name
        = self.name).size() > 0
}}
```

```
context DB!Table {
 constraint ClassExists {
   check : OO!Class.all.select(c|c.name
        = self.name).size() > 0
}}
```

This example of a *bx*, “faked” in Epsilon, is a deliberately simple one chosen to illustrate the concepts. Note even that the CD2RDBM problem can lead to more interesting (i.e. less symmetric) *bx*, e.g. manipulating inheritance in the class model.

3 Checking Compatibility of the Transformations

The critical difference between the “faked” *bx* in the previous section and a true *bx* is the absence of guarantees about the compatibility of the transformations: upon the violation of `TableExists`, for example, does the execution of `AddTable` actually restore consistency? For this simple example, a manual inspection will quickly confirm that the transformations are indeed compatible in this sense. But what about more intricate *bx*? And what about *bx* that evolve and change over time? For the Epsilon-based approach to be a convincing alternative to a *bx* language, it is imperative that the compatibility (or not) of the transformations can be checked, and—crucially—that this can be done in a simple and automatic way. To this end, we propose to leverage and adapt some recent developments in the verification of graph transformations.

Graph transformation is a computation abstraction: the state of a computation is represented as a graph, and the computational steps as applications of rules (i.e. akin to string rewriting in Chomsky grammars, but lifted to graphs). Modelling a problem using graph transformation brings an immediate benefit in visualisation, but also an important one in terms of semantics: the abstraction has a well-developed algebraic theory that can be used for formal reasoning. This has been exploited to facilitate the verification of graph transformation systems, i.e. calculi for systematically proving specifications about graph properties before and after any execution of some given rules. Furthermore, such calculi have been generalised to graph programs [13], which augment the abstraction with expressions over labels and familiar control constructs (e.g. sequential composition, branching) for restricting the application of rules.

Habel et al. [7,8] developed weakest precondition calculi for proving specifications of the form $\{pre\}\ P\ \{post\}$, which express that if a graph satisfies the precondition pre, then any graph resulting from the execution of graph program P will satisfy the postcondition $post$; these pre- and postconditions expressed using nested conditions, a graphical formalism for first-order (FO) structural properties over graphs. They defined constructions that, given a nested condition $post$ and program P, would return a weakest liberal precondition $\text{Wlp}(P, post)$, representing the weakest property that must hold for successful executions of P to establish $post$. The specification would then be (dis)proven by checking the validity of $pre \Rightarrow \text{Wlp}(P, post)$ in an automatic FO theorem prover. Poskitt and Plump developed proof calculi in a similar spirit, separately addressing two extensions: programs and properties involving attribute manipulation [14,15], and reasoning about non-local structural properties [16].

We aim to exploit this work to check the compatibility of transformations in the Epsilon approach to *bx*. In particular, we are developing automatic translations of EWL transformations to graph programs (denoted P_S, P_T for the source and target updates respectively), and translations from EVL constraints to nested conditions (denoted evl). Then, the task of checking compatibility of the transformations, as shown in Figure 2, reduces to proving the specifications $\{evl\}\ P_S;\ P_T\ \{evl\}$ and $\{evl\}\ P_T;\ P_S\ \{evl\}$ (here we assume the input graphs to be disjoint unions of the two models). Intuitively: if the models are consistent to start with, and executing the transformations in either order maintains consistency, then the transformations are compatible.

The technical challenges of the process fall into two main parts: computing the abstractions, and checking validity. Defining translations for the former requires care: we need to determine how much of the EWL language can be handled, we need to ensure that the graph-based semantics we abstract them to is “correct”, and we need to adapt the proof technology to our specific needs. The work in [14,15,16], for example, does not presently support type graphs (causing more effort to encode conformance to

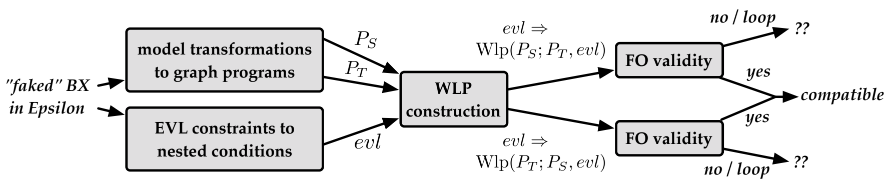

"faked" BX in Epsilon
model transformations to graph programs
EVL constraints to nested conditions
P_S
P_T
evl
WLP construction
$evl \Rightarrow \text{Wlp}(P_S; P_T, evl)$
$evl \Rightarrow \text{Wlp}(P_T; P_S, evl)$
FO validity
FO validity
no / loop
??
yes
yes
compatible
no / loop
??

Fig. 2. Overview of the process for checking compatibility of the transformations

metamodels). Similar concerns must be addressed for the translations of EVL to nested conditions (we can take inspiration from recent work on such translations for core OCL [2]). For the challenge of checking validity, we aim to leverage existing FO theorem provers (e.g. Vampire) as much as possible, adapting existing translations of nested conditions to FO logic [7,14]. Given the undecidability of FO validity, we also aim to explore the use of the GROOVE model checker [6] in finding counterexamples when the theorem provers respond with "no", or do not appear to terminate.

Our example *bx* for the CD2RDBM problem is easily translated into graph programs and nested conditions, as given in Figure 3. The programs P_S, P_T are the individual rules creating respectively a class or table node labelled `newName` (here, $\emptyset$ denotes the empty graph, indicating that the rules can be applied without first matching any structure, i.e. unconditionally). The nested condition evl, given on the right, expresses that for every class (resp. table) node, there is a table (resp. class) node with the same name (we do not define here a formal interpretation, but note that `x`, `y` are variables, and that the numbers indicate when nodes are the same down the nesting of the formula). Were the weakest liberal preconditions to be constructed, we would find:

$$\text{Wlp}(P_S; P_T, evl) \equiv \text{Wlp}(P_T; P_S, evl) \equiv evl.$$

Since $evl \Rightarrow evl$ is clearly valid, both $\{evl\}\ P_S;\ P_T\ \{evl\}$ and $\{evl\}\ P_T;\ P_S\ \{evl\}$ must hold, and—assuming correctness of the abstractions—the original EWL transformations are therefore compatible with respect to the EVL constraints.

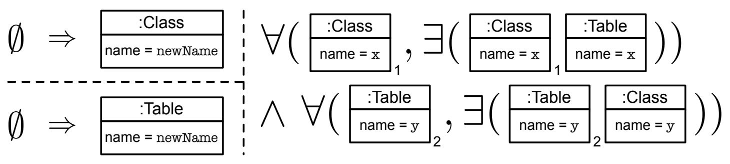


Fig. 3. Our CD2RDBM *bx* expressed as graph transformation rules and a nested condition

4 Next Steps

After further exploring the CD2RDBM example, we will identify a selection of *bx* case studies—from the community repository [4] and beyond—that exhibit a broader range of characteristics and challenges to address. We will implement these *bx* using EWL transformations and EVL constraints, then manually translate them into graph transformations and nested conditions. These will serve as a proof of concept, but also as guidance, helping us to determine how far we should adapt the proof calculi to support our goals (e.g. introducing type graphs for capturing the metamodels). After implementing the weakest precondition calculations and translations to FO logic, we will design and implement automatic translations from Epsilon *bx* to their corresponding graph-based abstractions, initially focusing on a core (but expressive) subset of the languages. Finally, we will explore the use of GROOVE in finding counterexamples when verification fails, by exploring executions of the graph transformation rules.

References

1. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A standard-compliant metamodeling framework with graph transformations. In: ECMDA-FA 2006. LNCS, vol. 4066, pp. 361–375. Springer (2006)
2. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From core OCL invariants to nested graph constraints. In: ICGT 2014. LNCS, vol. 8571, pp. 97–112. Springer (2014)
3. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. In: GT-VMT 2002. ENTCS, vol. 72, pp. 103–117. Elsevier (2003)
4. Cheney, J., McKinna, J., Stevens, P., Gibbons, J.: Towards a repository of Bx examples. In: EDBT/ICDT Workshops. vol. 1133, pp. 87–91. CEUR-WS.org (2014)
5. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer (2009)
6. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Software Tools for Technology Transfer 14(1), 15–40 (2012)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19(2), 245–296 (2009)
8. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460. Springer (2006)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
10. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218. Springer (2009)
11. Macedo, N., Cunha, A.: Implementing QVT-R bidirectional model transformations using Alloy. In: FASE 2013. LNCS, vol. 7793, pp. 297–311. Springer (2013)
12. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: ICECCS 2009. pp. 162–171. IEEE Computer Society (2009)
13. Plump, D.: The design of GP 2. In: WRS 2011. EPTCS, vol. 82, pp. 1–16 (2012)
14. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, The University of York (2013)
15. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
16. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: ICGT 2014. LNCS, vol. 8571, pp. 33–48. Springer (2014)
17. Stevens, P.: A landscape of bidirectional model transformations. In: GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer (2007)
18. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software and System Modeling 9(1), 7–20 (2010)

Towards Analysing Non-Determinism in Bidirectional Transformations [⋆]

Romina Eramo, Romeo Marinelli, Alfonso Pierantonio, and Gianni Rosa

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila, Italy
name.surname@univaq.it

Abstract. In Model-Driven Engineering, the potential advantages of using bidirectional transformations are largely recognized. Despite its crucial function, bidirectionality has somewhat limited success also because of the ambivalence concerning non-bijectivity. In fact, in certain situations more than one admissible solution is in principle possible, despite most of the current languages generate only one model at time, possibly not the desired one.
In this paper, we propose to manage non-determinism during the design process. The approach aims to analyze bidirectional transformations with the purpose to detect ambiguities and support designers in solving non-determinism in their specification.

1 Introduction

In Model-Driven Engineering (MDE) [18] bidirectionality in transformations has been always regarded as a key mechanism [20]. Its employment comprises mapping models to other models to focus on particular features of a system, simulate/validate a given application, and primarily keeping a set of interrelated models synchronized or in a consistent state. Despite its relevance, bidirectionality has rarely produced anticipated benefits as demonstrated by the lack of a language comparable to what ATL[1] represents for unidirectional transformations. Among the reasons why bidirectional techniques had limited success there is the ambivalence concerning non-bijectivity: when bidirectional transformations are non-bijective, there may be multiple *update policies* to transform two models into a consistent state, introducing uncertainty and non-determinism [8].

Most current languages are able to generate only one model. Thus, non-injective transformations involved in round-tripping can give rise to results, which are somewhat unpredictable. In these cases, the solution is normally identified according to heuristics, language implementation decisions and/or to the order the rules are written, as it happens with Medini [14] and ModelMorf [15]. Recently, few declarative approaches [6, 12, 4] to bidirectionality have been proposed; they are able to cope with the non-bijectivity by generating all the admissible solutions of a transformation at once. Among them, the Janus Transformation Language [6] (JTL) is a model transformation language specifically tailored to support bidirectionality and change propagation. Its semantics relies on Answer Set Programming (ASP) [9] in order to generate multiple

⋆ This research was supported by the EU through the Model-Based Social Learning for Public Administrations (Learn Pad) FP7 STREP project (619583).

[1] http://www.eclipse.org/atl/

models. However, the JTL transformation engine is not able to detect non-deterministic rules, i.e., rules that can give place to multiple solutions, at static-time.

There have been several works analyzing semantic issues and idiosyncrasies of bidirectional model transformations. [21] emphasizes the need of a clear semantics to grant developers full control about what the transformation does. The author goes even further by claiming that a transformation must be deterministic in order to ensure that developers will find the transformation behavior predictable. However, there exists cases in which designers are not able to disambiguate their transformations. In fact, they may lack the information needed beforehand to take appropriate design decisions. Moreover, very often the non-determinism in a transformation becomes evident only after its execution, especially if the language implementation resolves latent ambiguity by means of default strategies whose behavior is unclear to developers.

In this paper, we present an approach to statically detect non-determinism in bidirectional transformations, i.e., without executing them. To this end, Answer Set Programming (ASP) [9] is exploited to realize a logic environment for the analysis of bidirectional transformations. In particular, transformations are translated into logical rules and constraints able to analyze the behavior of the transformation with an emphasis on non-determinism. The approach aims to support designers in solving non-determinism in their specifications by detecting the rules that give place to alternative solutions at design time[2].

The paper is organized as follows. Section 2 introduces the problem by means of an example. Section 3 describes the static analysis of bidirectional transformation. Sections 4 and 5 present the proposed approach and show it in practice. Section 6 describes related work. Finally, Section 7 draws some conclusion and future work.

2 A motivating example

Non-determinism is a frequent aspect which affects model transformation design and implementation. In this section, we describe how *ambiguous* mappings in a model transformation may cause multiplicity in the generated solution, i.e., to somewhat a form of uncertainty.

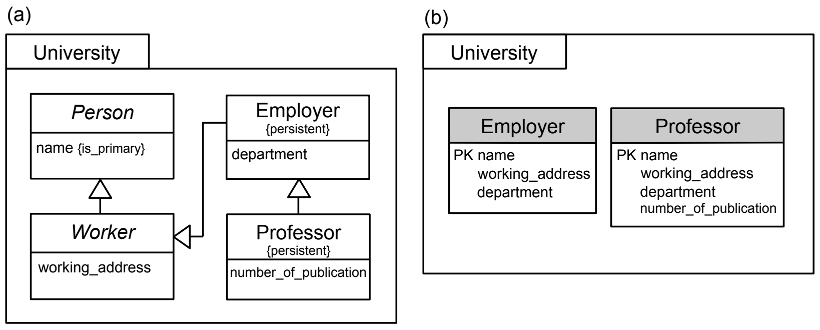


Fig. 1: The UML model (a) and the correspondent RDBMS model (b)

Scenario Let us considering a typical round-trip problem based on a non-bijective class diagram to relational data base (*UML2RDBMS*) benchmark scenario [7, 10]. Only

[2] Note that, the proposed analysis environment is general as does not depend on the JTL engine

persistent classes are mapped to correspondent tables and their attributes to columns in the tables, including inherited attributes. In order to preserve all the information of the source diagram, attributes of non-persistent classes have to be distributed over those tables stemming from persistent classes which access non-persistent ones.

The UML model in Fig. 1(a) shows the package `university` that is composed of an inheritance hierarchy of classes, whereas the correspondent schema is depicted in Fig. 1(b). Let us suppose that the generated model is manually modified (e.g., for satisfying new requirements) as depicted in Fig. 2(a): a new column `email` has been added in the table `employer`. This gives place to an interesting situation since such modifications can be reflected to the source model in Fig. 1(a) in three alternative ways: the attribute (corresponding to the manually added column) can be a member of the class `employer` or with each of parent classes `worker` and `person`, as in Fig. 2(b). Consequently, although not shown in the figure, starting from each of this alternatives, every subclass will inherit the attribute. Thus, more than one source model propagating the changes exist.

Implementation The *UML2RDBMS* bidirectional transformation, which relates class diagrams and relational database models, has been implemented by means of the Janus Transformation Language (JTL) [6].

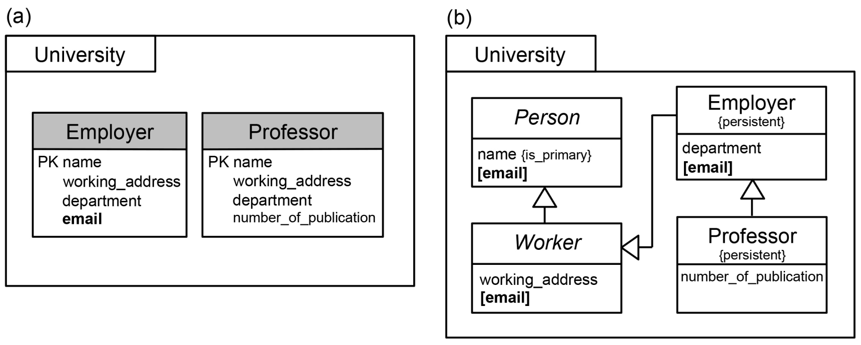


Fig. 2: The modified RDBMS model (a) and the correspondent UML model (b)

In Listing 1.1 we report a fragment of the transformation which is expressed in the textual concrete syntax of JTL and applied on models given by means of their Ecore representation within the EMF framework[3]. In particular, the following relations are defined: *a) Class2Table*, which relates classes and tables in the two different metamodels, *b) Attribute2Column*, which relates attibutes and columns in the two different metamodels and *c) SuperAttribute2Column*, which relates attributes of the parent class to columns of the corresponding child table. The *when* and *where* clause specify conditions on the relation. In particular, the when clause in Line `18` allow to navigate the parent classes of each attribute, then the where clause in Line `19` generates the correspondent columns. These relations are bidirectional, in fact both the contained domains are specified with the construct *enforce*.

The forward application of the transformation is illustrated in Fig. 1, where the UML model is mapped to the correspondent RDBMS model. As aforementioned the

[3] http://www.eclipse.org/modeling/emf/

transformation is non-injective. The back propagation of the changes showed in Fig. 2 gives place to the following situation: the transformation execution produces a set of three models each one represents a different solution where the column `email` belongs to each of the three different classes of the hierarchy. In particular, the modified RDBMS target model in Fig. 2(a) is mapped back to the UML source models in Fig. 2(b).

Such non-determinism can be resolved by specifying in the transformation that any new column of the table must be mapped to the corresponding attribute of the class from which the table is generated (for instance, the new column `email` of the table `employer` is mapped to the attribute `email` of the class `employer`).

```
transformation UML2RDBMS(uml:UML, rdbms:RDBMS)  { ...
 top relation Class2Table {
  cn, an: String;
  enforce domain uml c:Class {
   is_persistent = true,
   name = cn,
   attrs = attr: Attribute { name = an}
  };
  enforce domain rdbms t:Table {
   name = cn,
   cols = col: Column { name = an }
  };
  when { ... }
  where { Attribute2Column(c, t); }
 }
 relation Attribute2Column {
  an, at : String;
  enforce domain uml c:Class {
   attrs = attr: Attribute { name = an, owner = c, is_primary = false }
  };
  enforce domain rdbms t:Table {
   cols = col: Column { name = an, owner = t }
  };
  when { ... }
 }
 top relation SuperAttributeToColumn{
  enforce domain uml c: Class {
   parent = sc: Class { }
  };
  enforce domain rdbms t: Table { };
  when { ClassToTable(c,t) or (cc = c.parentOf and SuperAttributeToColumn(cc,t));}
  where { AttributeToColumn(sc, t); }
 } ...
```

Listing 1.1: A fragment of the UML2RDBMS transformation in JTL

Thus, the considered piece of transformation can be made deterministic by accommodating the mentioned refinement. However, when size and complexity of metamodels and transformations grow, the designer should be supported by an automated tool capable of detecting the ”guilty” rules in the transformations.

3 Static analysis of bidirectional transformations

Typically, bidirectional transformations are specified by a collection of rules which define mapping from source to target metamodel elements, and viceversa. By considering existing relational languages [16, 6], mappings among metamodel elements are expressed as relations that are executed in both the directions. In this context, transformations are models as well, therefore amenable to model operations.

Analysis of model transformations can be used for a wide range of assessments including the satisfaction of specific requirements, verifications/validation, and performance analysis. Existing approaches and tools involve testing (e.g., test case generation for model transformations), or analysis performed on executing programs (e.g., run-time monitoring) [1]. In contrast, static analysis is performed without actually executing the model transformation or generating test cases; in fact, it is performed on some (abstract) version of the transformation code with the scope to allow designer to understand and review the code at design-time.

Non-determinism is a critical aspect in bidirectionality. Thus, detecting the ambiguous fragments of a transformation at design-time may prevent unwanted behaviors like a severe increase of the solution space. At this scope, it is of paramount relevance to support the designer in understanding which rule is ambiguous and how to refactor the transformation in order to reduce non-determinism. This is intrinsically difficult as it not unusual that the backward (forward) execution of an seemingly deterministic forward (backward) transformation generates more than one alternative. The challenge consists of precisely identifying those combinations of rules which are responsible of the solution multiplicity.

The analysis is performed by considering not only the behavior of individual statements and declarations, but rather including the complete transformation. Information obtained from the analysis are used for detecting possible coding ambiguities. In particular, the approach aims to detect so-called *non-deterministic portions* of the transformation represented by collections of rules that if executed together may cause multiple alternative solutions. Please note how the degree of non-determinism of a single fragment depends on the model instance given as input.

In this paper, we propose to use the Answer Set Programming (ASP) [9] to analyze bidirectional transformation. ASP is a form of declarative programming oriented towards difficult (primarily NP-hard) search problems and based on the stable model (answer set) semantics of logic programming. Then the ASP solver[4] finds and generates, in a single execution, all the possible models which are consistent with the logic rules by a deductive process. The proposed ASP-based engine is able to analyze the model transformation specification and verify non-determinism conditions expressed by means of rules and constraints. The environment has been implemented as a set of plug-ins of the Eclipse framework and mainly exploits EMF[5] as illustrated in the next section.

4 Description of the approach

Figure 3 illustrates the approach, which is comprised of two main steps. The first step translates a model transformation specification into a notation suitable for the analysis. The second analyzes the information extracted in the previous step and produces a feedback. These two steps are explained in more details in the following.

(1) Translation of the transformation into the analysis notation
A model transformation consists of a set of rules which define mapping among element types of the involved left- and right-metamodels. Generally, a rule is constrained with pre- and post-conditions that limit its applicability and behavior. This step defines a

[4] http://www.dlvsystem.com/
[5] http://www.eclipse.org/modeling/emf/

characterization that permits to capture only the information which is relevant for performing the analysis (neglecting irrelevant details which are not pertinent to our purpose). The notation used for representing the elicited information is the *Transformation Analysis Language* (see Fig. 3). Each rule is represented by means of two sets containing the left- and right-patterns, respectively. Whereas, each pattern is represented by a class and values for any of its properties and references. As defined in OCL[6], a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. In other words, a pattern can be considered a template for objects and their properties that must be located in a candidate model to satisfy the rule. The model representing the bidirectional model transformation (*Bidirectional Transformation Model* in Fig. 3) must be translated into the corrisponding model for the analysis (*Transformation Analysis Model*) by means of a semantic anchoring [5] able to translate rules and constraints as set of patterns.

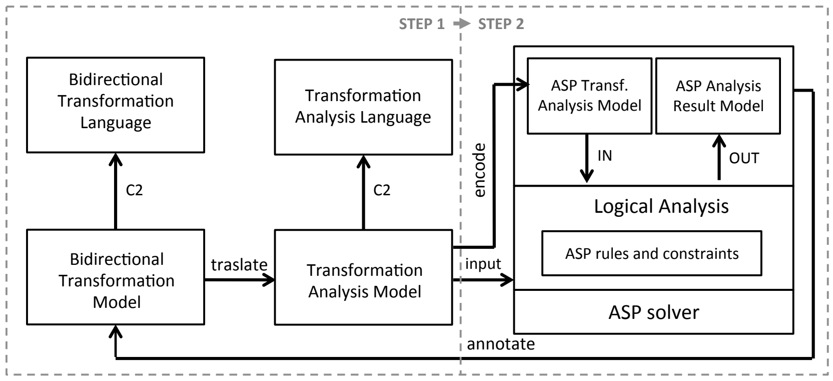

STEP 1
STEP 2
Bidirectional Transformation Language
Transformation Analysis Language
C2
C2
Bidirectional Transformation Model
traslate
Transformation Analysis Model
encode
input
ASP Transf. Analysis Model
ASP Analysis Result Model
IN
OUT
Logical Analysis
ASP rules and constraints
ASP solver
annotate

Fig. 3: Architecture overview of the approach

(2) ASP-based analysis

Once the analysis models are generated by abstracting from the concrete transformations, they need to be translated into ASP in order to be analyzed. The analysis models and the metamodels involved in the transformations are consistently consistently encoded as a knowledge base, whereas the properties which are required to be checked are translated in constraints and verified by ASP rules. As aforementioned the scope of the analysis is to provide an automated detection of non-deterministic rules. In particular, it generate sets of logically related rules whose simultaneous applications may generate multiple admissible solutions. When the ambiguous rules are marked by the analysis process, the designer can refactor the initial specification by resolving the non-determinism. In Listing 1.2 a fragment of the ASP rules for the detection of non-deterministic code is presented. In particular, the rule in Lines `1-4` deduces pairs of *equal domains* by comparing patterns and predicates of each domain. The rule in Lines `6-9` deduces *non-deterministic relations*; in particular, for each transformation direction, relations with equal domains are detected. Finally, the rule in Lines `11-13`

[6] http://http://www.omg.org/spec/OCL/

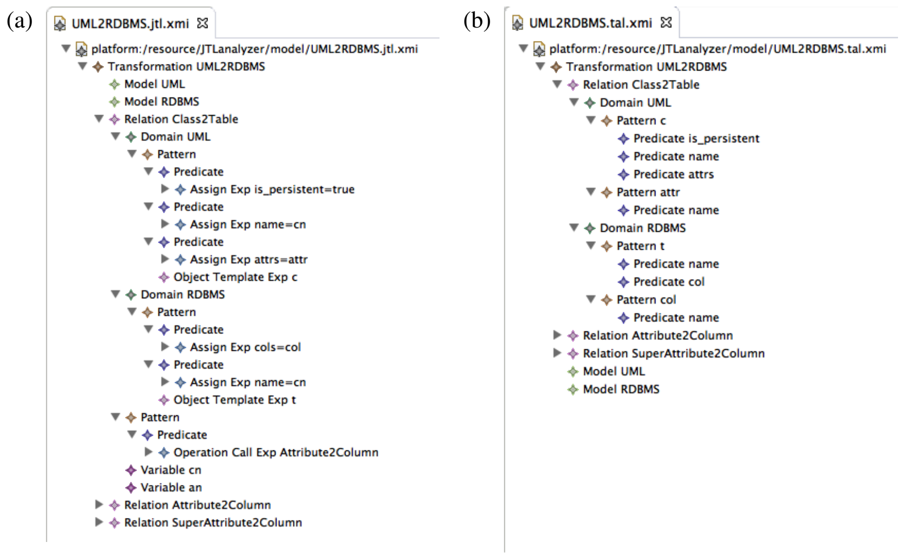
deduces pair of *ambiguous relations* which cause non-determinism if simultaneously executed.

```
are_equal_domains(ID1, ID2, Dom) :-
 have_equal_patterns(ID1, ID2, Dom, MC, MCName),
 have_equal_predicates(ID1, ID2, Dom, MC, MCName),
 not have_different_patterns(ID1, ID2, Dom).

non_deterministic_relation(ID, RelName, DomLeft) :-
 are_equal_domains(ID, ID2, DomRight),
 relation_name(ID, RelName),
 tranformation_model(DomRight), tranformation_model(DomLeft), DomRight != DomLeft.

are_ambiguous_relations(ID1, ID2, DomLeft) :-
 are_equal_domains(ID1, ID2, DomRight),
 tranformation_model(DomRight), tranformation_model(DomLeft), DomRight != DomLef
```

Listing 1.2: A fragment of the ASP rules and constraints for the analysis

5 Running the approach

In this section, we present an application of the proposed approach to the UML2RDBMS example presented in Sect. 2. The goal is to illustrate how to use of the approach in practice by exploiting the developed environment. In particular, we analyze the JTL implementation of the UML2RDBMS transformation[7]. First, the bidirectional transformation is translated into the analysis model. Then, the analysis is executed to highlight those rules which may be responsible of non-determinism.



(a)

```
UML2RDBMS.jtl.xmi
▼ platform:/resource/JTLanalyzer/model/UML2RDBMS.jtl.xmi
  ▼ Transformation UML2RDBMS
      Model UML
      Model RDBMS
    ▼ Relation Class2Table
      ▼ Domain UML
        ▼ Pattern
          ▼ Predicate
            ▶ Assign Exp is_persistent=true
          ▼ Predicate
            ▶ Assign Exp name=cn
          ▼ Predicate
            ▶ Assign Exp attrs=attr
            Object Template Exp c
      ▼ Domain RDBMS
        ▼ Pattern
          ▼ Predicate
            ▶ Assign Exp cols=col
          ▼ Predicate
            ▶ Assign Exp name=cn
            Object Template Exp t
      ▼ Pattern
        ▼ Predicate
          ▶ Operation Call Exp Attribute2Column
        Variable cn
        Variable an
    ▶ Relation Attribute2Column
    ▶ Relation SuperAttribute2Column
```

(b)

```
UML2RDBMS.tal.xmi
▼ platform:/resource/JTLanalyzer/model/UML2RDBMS.tal.xmi
  ▼ Transformation UML2RDBMS
    ▼ Relation Class2Table
      ▼ Domain UML
        ▼ Pattern c
            Predicate is_persistent
            Predicate name
            Predicate attrs
        ▼ Pattern attr
            Predicate name
      ▼ Domain RDBMS
        ▼ Pattern t
            Predicate name
            Predicate col
        ▼ Pattern col
            Predicate name
    ▶ Relation Attribute2Column
    ▶ Relation SuperAttribute2Column
      Model UML
      Model RDBMS
```

Fig. 4: The UML2RDBMS tranformation and analysis models

(1) Translating UML2RDBMS into the analysis language
Starting from the specification of the UML2RDBMS transformation by means of JTL described in Sect. 2 (see Listing 1.1), the corresponding JTL model in Ecore is represented in Fig. 4(a). As aforesaid, in JTL the mapping between candidate models are

[7] The implementation is available at http://jtl.di.univaq.it/

represented by means of a set of relations defined by two domain patterns. Each relation includes a pair of *when* and *where* predicates which specify the pre- and post-conditions that must be satisfied by the elements of the candidate models.

The JTL model in Fig. 4(a) is translated into the correspondent analysis model in Fig. 4(b) by means of an automated model-to-model transformation implemented in ATL. In particular, all the mapping rules (including pre- and post- conditions) of the source JTL model are translated in the corresponding sets of patterns of the target analysis model. For instance, let us consider the *Class2Table* relation, then every predicate belongs to the UML or RDBMS domain pattern, is represented by an OCL expression, and is evaluated as follows: *(i)* each `Object Template Expression` is transformed in a named `Pattern` of the analysis model; *(ii)* each statement is evaluated and transformed in a correspondent predicate; finally *(iii)* each condition is considered as a pattern and added to the correspondent `Domain` in the analysis model.

(2) Executing the analysis of the UML2RDMBMS transformation
The transformation analysis model, generated during the previous step is entered into the ASP-based engine. In order to perform the analysis, model transformation and the involved metamodels have to be encoded in the ASP language. For instance, in Listing 1.3 a fragment of the ASP encoding of the analysis models in Fig. 4(b) is showed. In particular, it declares the `class2table` relation with an identifier `1` (in Lines 1) and the correspondent patterns and predicates for the domain UML (in Lines 2-10) and the patterns and predicates for the domain RDBMS (in Lines 11-17), as well.

```
 1 relation_name(1, class2table).
 2 relation_domain(1, uml).
 3  relation_pattern(1, uml, c, class).
 4   relation_predicate(1, uml, c, is_persistent, true).
 5   relation_predicate(1, uml, c, name, cn).
 6   relation_predicate(1, uml, c, attrs, attr).
 7  relation_pattern(1, uml, attr, attribute).
 8   relation_predicate(1, uml, attr, name, an).
 9   relation_predicate(1, uml, attr, owner, c). % added from where
10   relation_predicate(1, uml, attr, is_primary, false). % added from where
11 relation_domain(1, rdbms).
12  relation_pattern(1, rdbms, t, table).
13   relation_predicate(1, rdbms, t, name, cn).
14   relation_predicate(1, rdbms, t, cols, col).
15  relation_pattern(1, rdbms, col, column).
16   relation_predicate(1, rdbms, col, name, an).
17   relation_predicate(1, rdbms, col, owner, t).
```

Listing 1.3: A fragment of the analysis model encoded in ASP

Starting from the ASP encoding, the analysis is now able to detect the non-deterministic rules. In particular, the analysis outcome consists of a number of rule sets; each of which includes logically related relations whose simultaneous applications may generate multiple solutions. For instance, Listing 1.4 shows a fragment of the result of the analysis executed on the UML2RDBMS specification (in Listing 1.1). In particular, when the transformation is executed in the RDBMS-to-UML direction, the set of non-deterministic relations `class2table`, `superAttribute2column` and `superAttribute2column` is detected. In particular, in the scenario described in Sect. 2, more than one alternative solutions can be obtained because of the simultaneous executions of the three ambiguous mappings.

```
%%%% Analysis Model
%%%% sets of non-deterministic relations (UML to RDBMS direction):
[]
%%%% sets of non-deterministic relations (RDBMS to UML direction):
[(1,class2table), (3,superAttribute2column), (5,superAttribute2column)]
[(..), (...)]
```

Listing 1.4: A fragment of the output of the analysis

The final goal of such analysis is to convey to the transformation implementor enough information to perform a refinement which resolves the non-determinism. In this respect, the ”guilty” rules are opportunely marked in such a way the sets of rules responsible of the non-determinism sources can be easily recognized by the designer.

6 Related work

Non-determinism is a recurrent aspect affecting model transformation design and implementation. As said, most of the current languages (e.g.,[11, 17, 3, 19, 16, 14, 15] are able to generate only one model at time; in this way even if the transformation is non-bijective the solution is normally identified according to details which are often unknown to the designer rendering such approached unpredictable and therefore unpractical. Existing declarative approaches [6, 12] cope with the non-bijectivity by generating sets of models at once, i.e., all the admissible solutions of a transformation are generated as a solution set. Among them, JTL [6] is specifically tailored to support bidirectionality and change propagation. In [12], the authors propose to use Alloy and follows the predictable principle of least change [13], by using a function that calculates the distance between instances reducing the generated models. Non-determinism is also considered in BiFlux [17], which adopts a novel ”bidirectional programming by update” paradigm, where a program succinctly and precisely describes how to update a source document with a target document, such that there is a unique inverse source query for each update program. Within the Triple Graph Grammars, PROGRES [2] considers non-deterministic cases by demanding user intervention to rule execution in order to choose the desired behavior.

Existing approaches and tools involve testing (e.g., test case generation for model transformations), or analysis performed only on executing programs (e.g., run-time monitoring) [1]. In contrast, in [1] the author propose an approach to analyze model transformation specification represented in Alloy. The simulation produces a set of random instances that conform to the well-formedness rules and eventually that satisfy certain properties. Furthermore, existing functional approaches perform model transformation analysis [11, 17], in order to evaluate the transformation validity (generally, in terms of correctness) before its execution.

7 Conclusion and Future Work

In this paper, we have proposed an approach to detect non-determinism in bidirectional transformations at static-time, i.e., without executing the transformation. The intention is to provide transformation implementors with a tool capable of analyzing transformations in order to return feedback which could resolve potential non-determinism. The approach has been demonstrated on a small yet significant case study implemented in JTL. Despite the analysis is completely independent from JTL, in order to make the approach completely language-independent additional implementation efforts are required which we intend to do in the near future. Furthermore, the logical foundation of the engine makes possible the verification of different formal properties by translating them in ASP, with the scope to provide a mean for improving the quality of model transformation specifications.

References

1. K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of model transformations via Alloy. In *ModeVVa'07*, pages 47–56, 2007.
2. S. M. Becker, S. Herold, S. Lohmann, and B. Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and System Modeling*, 6(3):287–315, 2007.
3. A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *POPL 2008*, pages 407–419, 2008.
4. G. Callow and R. Kalawsky. A Satisficing Bi-Directional Model Transformation Engine using Mixed Integer Linear Programming. *JOT*, 12(1):1: 1–43, 2013.
5. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic Anchoring with Model Transformations. In *ECMDA-FA*, 2005.
6. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: a bidirectional and change propagating transformation language. In *SLE10*, pages 183–202, 2010.
7. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective - GRACE meeting. In *Procs. of ICMT2009*.
8. R. Eramo, A. Pierantonio, and G. Rosa. Uncertainty in bidirectional transformations. In *Procs. of MiSE 2014*, 2014.
9. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Procs of ICLP*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
10. T. Hettel, M. Lawley, and K. Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *Procs. of ICMT 2008*, 2008.
11. S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *ASE 2011*, 2011.
12. N. Macedo and A. Cunha. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In *FASE*, pages 297–311, 2013.
13. N. Macedo, H. Pacheco, A. Cunha, and J. N. Oliveira. Composing least-change lenses. *ECEASST*, 57, 2013.
14. MediniQVT. http://projects.ikv.de/qvt/.
15. ModelMorf. http://www.tcs-trddc.com/modelmorf/.
16. Object Management Group (OMG). MOF 2.0 QVT Final Adopted Specification v1.1, 2011. OMG Adopted Specification formal/2011-01-01.
17. H. Pacheco and Z. Hu. Biflux: A bidirectional functional update language for XML. In *BIRS workshop: Bi-directional transformations (BX)*, 2013.
18. D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
19. A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *in Proc. of WG94*. Springer, 1995.
20. S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
21. P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *SOSYM*, 8, 2009.