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

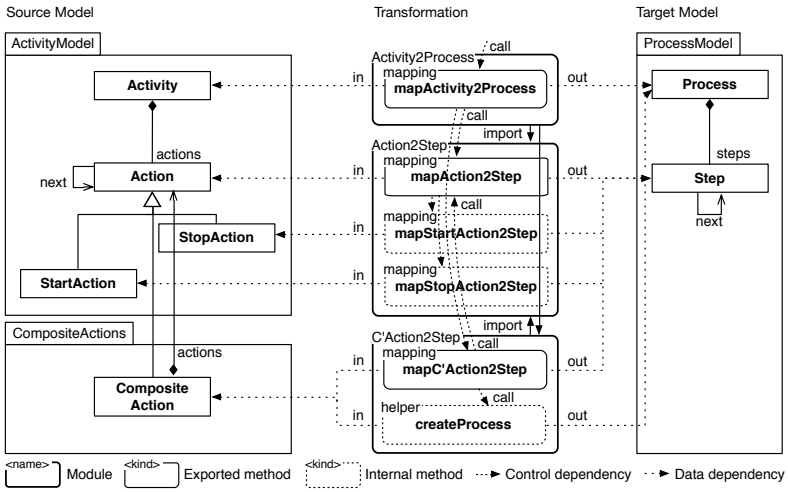

Fig. 1: Activity2Process transformation – Method and model scoping

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 de-

composition. 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:

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

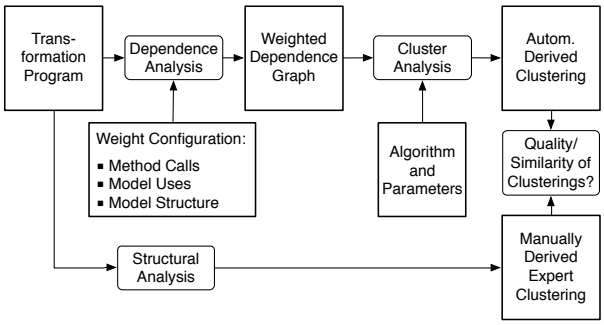

Fig. 2: Clustering approach

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.

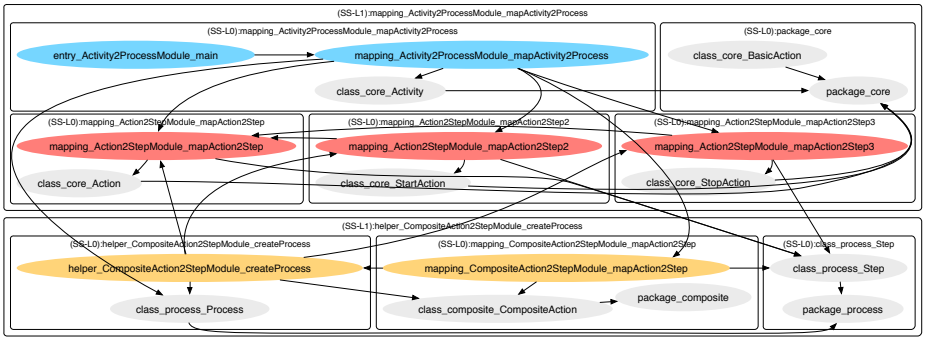

Fig. 3: Activity2Process transformation – Bunch-derived clustering based on class-level dependencies

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