Towards Generic Modularization Transformations

Martin Fleck
Business Informatics Group
Institute of Software Technology and Interactive Systems
TU Wien, Austria
fleck@big.tuwien.ac.at

Javier Troya
ISA Research Group
ETS de Ingenieria Informatica
Universidad de Sevilla, Spain
jtroya@us.es

Manuel Wimmer
Business Informatics Group
Institute of Software Technology and Interactive Systems
TU Wien, Austria
wimmer@big.tuwien.ac.at

Abstract

Modularization concepts have been introduced in several modeling languages in order to tackle the problem that real-world models quickly become large monolithic artifacts. Having these concepts at hand allows for structuring models during modeling activities. However, legacy models often lack a proper structure, and thus, still remain monolithic artifacts.

In order to tackle this problem, we present in this paper a modularization transformation which can be reused for several modeling languages by binding their concrete concepts to the generic ones offered by the modularization transformation. This binding is enough to reuse different modularization strategies provided by search-based model transformations. We demonstrate the applicability of the modularization approach for Ecore models.

Categories and Subject Descriptors D.2 Software Engineering [*D.2.2 Design Tools and Techniques*]: Modules and interfaces; D.2 Software Engineering [*D.2.7 Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and reengineering; D.3 Programming Interfaces [*D.3.3 Language Constructs and Features*]: Modules, packages

General Terms Design

Keywords Modularization, Generic Transformation, Search-based Transformation, Design Quality

1. Introduction

Modeling is considered as the technique to deal with complex and large systems (Brambilla et al. 2012). Consequently, modularization concepts have been introduced in several modeling languages in order to tackle the problem that real-world models quickly become monolithic artifacts, especially when large systems have to be modeled (Reijers and Mendling 2008). Different kinds of modularization concepts have been introduced such as modules, aspects (Wimmer et al. 2011b), concerns (Alam et al. 2013), views (Atkinson et al. 2009), or subsets (Blouin et al. 2014), to name just a few. Having these concepts at hand allows for structuring models during modeling activities. However, legacy models often lack a proper structure as these modularization concepts have not been available or have not been applied, and thus, the models are still monolithic artifacts.

In order to tackle this problem, we present in this paper a modularization transformation which can be reused for several modeling languages by binding their concrete concepts to the generic ones offered by the modularization metamodel. This binding is enough to reuse different modularization strategies, mostly based on design quality metrics, provided by search-based model transformations. In its current version, we support modules which allow for composing different entities. We demonstrate the applicability of the modularization approach for Ecore models.

The contribution of this paper is threefold: (i) we provide generic modularization support for modeling languages having at least some kind of basic modularization support; (ii) we combine query-driven model transformations and generic model transformations to provide generic means to deal with heterogeneities between the generic modularization metamodel and the specific metamodels; (iii) we provide an application of the generic modularization support for real-world Ecore models.

The outline of the paper is as follows. In Section 2 we present the background needed for introducing our generic modularization approach in Section 3. In Section 4 we apply the generic approach for a set of real-world Ecore models. Finally, we discuss related work in Section 5, before we conclude with an outlook on future work in Section 6.

2. Background

In this section, we present the foundations on which we build our modularization approach. First, we explain how modularization is in its basic form supported in modeling languages and which kind of transformations may be combined to implement a reusable generic model transformation and how to instantiate it for performing a concrete modularization.

2.1 Modularization: The Basics

The basic modularization problem, also often referred to as software clustering, considers as input a potentially huge set of elements having certain relationships among them (Praditwong et al. 2011; Mkaouer et al. 2015). The goal for the modularization task is to find meaningful modules or clusters for these elements which consider certain quality characteristics. Traditionally, modularization approaches (or re-modularization approaches if some initial module structure is already given which should be improved) considered code-based software engineering artifacts. For instance, classes are modularized into modules based on their dependencies such as method invocations or field access. The considered quality characteristics are mostly based on object-oriented design metrics.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892698

Now the question arises how we may provide automated modularization support for modeling languages. For instance, languages such as UML and Ecore provide the package concept to structure classifiers, and other languages such as SDL and BPMN provide some means for modularization of models as well. While the languages themselves seem heterogenous, i.e., generic modeling language vs. domain-specific language, structural vs. behavioural, and so on, the modularization concepts in its basic form of having modules containing entities seem common. Thus, we aim to explore how we may provide a generic modularization transformation based on the combination of three different transformation types.

2.2 Generic Model Transformation

Currently there is little support for reusing transformations in different contexts since they are tightly coupled to the metamodels they are defined upon. Hence reusing them for other metamodels becomes challenging. Inspired from generic programming, generic model-to-model transformations (Cuadrado et al. 2011) have been proposed, which are defined over so-called metamodel concepts, which are later bound to specific metamodels. Thus, with the help of generic model transformations, we are able to define a generic modularization metamodel which can be bound to different specific modeling languages. Please note that a similar approach has been proposed as role-based model transformations for in-place transformations such as model refactorings (Reimann et al. 2010).

2.3 Query Structured Transformation

One major challenge for generic model transformations is to deal with the heterogeneities (Wimmer et al. 2011a) between the generic and the specific metamodels, i.e., one concept is defined in the generic metamodel as a class, but the same concept is represented in the specific metamodel as a pattern of different classes having specific relationships. In this paper, we propose to combine the generic model transformation approach with another emerging transformation approach called query structured transformation (Gholizadeh et al. 2014). The main idea behind the latter is to enhance the source metamodel with concepts of the target metamodel in a query-driven manner. By this, the source and the target metamodel are adjusted and the mapping between the metamodels is reduced to one-to-one correspondences. As we see later, this approach allows us to easily use one-to-one correspondences for binding the generic modularization metamodel to the specific metamodels.

2.4 Search-Based Model Transformations

Having now the mechanisms to define modularization as a generic transformation and using query structured transformations to enhance the binding mechanisms, there is still the question how to implement the generic transformation actually performing the modularization. In code-based modularization problems, the usage of search-based algorithms, such as genetic algorithm, simulated annealing, and so on, has been proposed to actually compute the optimal module structure for a given problem (Praditwong et al. 2011; Mkaouer et al. 2015). The success of search-based algorithms was mostly based on the meta-heuristic search capabilities, i.e., finding a good solution without enumerating the whole search space.

In our pervious work we combined search-based algorithms and model transformations in a framework called MOMoT (Fleck et al. 2015). This framework allows for formulating the goals of a transformation (such as it is needed for specifying the quality characteristics of a modularization) and in the transformation execution the meta-heuristic search algorithms are orchestrating the rule applications performed by the transformation engine to find a good solution in the search space. Based on this capabilities, we selected MOMoT for actually implementing the transformation rules needed for the generic modularization transformation and for defining the goals of the modularization. This clear separation of the transformation rules and the transformation goals also allows more easily to customize the generic modularization transformation for specific modeling languages.

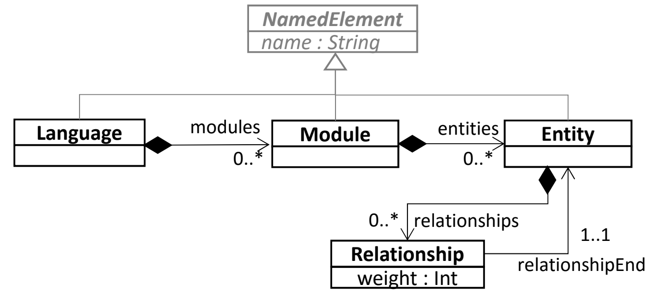

Figure 1. Generic Modularization Metamodel.

3. Generic Modularization

We now introduce our concept metamodel for modularization, our modularization chain, and outline several strategies for performing modularization based on the information provided by the concept metamodel.

3.1 Generic Modularization Metamodel

Our generic modularization metamodel is presented in Figure 1 (abstract classes and relationships are depicted in grey). Elements of type *Language* represent concrete instances of modeling languages (MLs). The concepts of a ML are therefore simplified to *Modules*, *Entities* and *Relationships*. We can see that a language is composed of modules, which represent the clusters that group entities with similarities. Such similarities can come in different ways. For instance, we can consider the similarities between the names of the entities, or the relationships among the entities. Furthermore, we have defined weights for the relationships, since some of them may be more important that others (cf. Section 4). We can see that an entity can have several relationships with other entities. Each relationship ends in a specific entity.

The idea is to express any modeling language in terms of our generic modularization metamodel. This means that the concepts appearing in the MLs are mapped to the three concepts described: modules, entities and relationships.

3.2 Modularization Transformation Chain

The overall transformation chain for the generic modularization of modeling languages is shown in Figure 2. Steps 1 and 2 are explained in this section, while step 3 is explained in Section 3.3. These three steps are exemplified with an application study in Section 4. Finally, step 4 is left for future work.

Our approach takes as input a *Modeling Language (ML)*. A ML is defined in terms of a domain-specific language (DSL). The structure of a DSL is expressed with a metamodel, which defines the concepts of the language and the relationships among them. The first step of our approach is implemented with a *Query Structured Model Transformation*, whose purpose is to make it easier and more generic the weaving of different DSLs. In our case, we want to translate a ML to our generic modularization metamodel (cf. Fig. 1).

Therefore, we seek the homomorphism between the metamodel of the ML and our generic modularization metamodel, i.e., the bindings between these two. This homomorphism, also called mapping, has to be manually identified by the software engineer, since she/he has to decide what is a module, an entity and a relation-

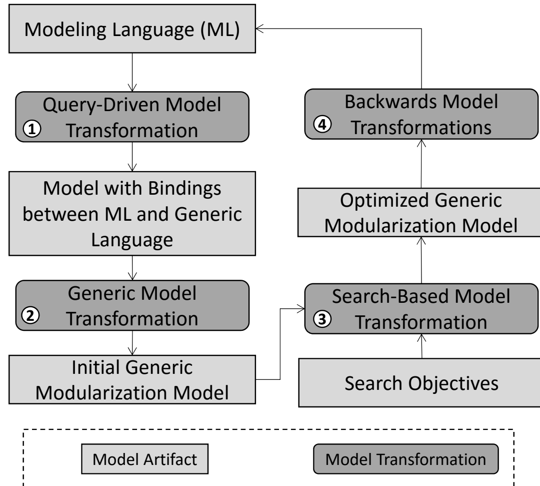

Figure 2. Generic Modularization Chain.

ship in the ML. Such mapping can be defined by simply annotating the metamodel of the ML or by adding new derived properties and classes to it that represent the mapping. The outcome of this step is a *Model with Bindings between ML and Generic Language*, "binding model" for short from here on. Thus, we obtain a model where specific concepts of the input ML are virtually connected to specific concepts of our generic modularization language.

The second step is to apply a *Generic Model Transformation* to such model. This transformation takes the binding model as input and generates the *Initial Generic Modularization Model*. In order to do so, it examines the bindings specified in the binding model. The generated model conforms to our generic modularization metamodel and is composed of only one module that contains all the entities. The entities, in turn, have relationships among them. The generic model transformation produces these relationships according to the information specified in the binding model.

In our proof-of-concept implementation (Fleck et al. 2016), we have implemented steps 1 and 2 with an ATL transformation for the Ecore case study. In future work, we plan to decouple both steps.

3.3 Modularization Strategies

The third step in our approach has to do with the optimal grouping of entities into modules. In order to do so, we apply search-based techniques using our MOMoT tool (Fleck et al. 2015). Therefore, we need to specify as input the *Search Objectives* that we want to optimize in our modularization. According to the fitness function defined by such objectives, our *Search-Based Model Transformation* decides the optimal modularization, i.e., how to optimally split the entities contained by the only module in the initial generic modularization model into different modules.

In this paper, we define four objectives: (i) coupling (COP), (ii) cohesion (COH), (iii) the difference between the maximum and minimum number of entities in a module (DIF) and (iv) number of modules (MOD). Coupling refers to the number of external relationships a specific module has, i.e., the sum of inter-relationship weights with other modules, whereas cohesion refers to the relationships within a module, i.e., the sum of intra-relationship weights in the module. Typically, low coupling is preferred as this indicates that a group covers separate functionality aspects of a system, improving the maintainability, readability and testability of the overall system (Yourdon and Constantine 1979). On the contrary, the cohesion within one module should be maximized to ensure that it does not contain parts that are not part of its functionality. Regarding the number of modules, it should be maximized in order to avoid having all entities in a single large module. At the same time, the maximum and minimum number of classes in the modules ought to be minimized to aim at equal-sized modules.

Our framework is implemented within the Eclipse Modeling Framework (EMF)[1] and builds upon Henshin[2] (Arendt et al. 2010) to define model transformations and the MOEA framework[3] for providing optimization techniques. Henshin is a graph transformation engine that provides a graphical notation for defining model transformations as graph transformation rules. The MOEA framework provides several multi-objective evolutionary algorithms, such as NSGA-II (Deb et al. 2002), NSGA-III (Deb and Jain 2014), and ε-MOEA (Deb et al. 2003), as well as tools to execute and statistically test the search performance of these algorithms.

```
fitness = {
  preprocess = {
  // use attribute for external calculation
    val root = MomotUtil.getRoot(
      solution.execute, typeof(Language))
    solution.setAttribute("metrics",
      MetricsCalculator.calculate(root))
  }
  objectives = {
    Coupling : minimize { // java-like syntax
      val metrics = solution.getAttribute(
        "metrics", typeof(LanguageMetrics))
      metrics.coupling
    }
    Cohesion : maximize {
      val metrics = solution.getAttribute(
        "metrics", typeof(LanguageMetrics))
      metrics.cohesion
    }
    NrModules : maximize {
      (root as Language).^modules
        .filter[ m | !m.entities.empty].size
    }
    MinMaxDiff : minimize {
      val sizes = (root as Language).^modules
        .filter[m | !m.entities.empty]
        .map[m | m.entities.size]
      sizes.max - sizes.min
    }
  }
}
```

Listing 1. Definition of objectives that should be optimized

The way our search approach works is the following. Given the initial generic modularization model and a set of search objectives defined in our configuration language, our approach introduces a set of empty modules in the language. The number of empty modules that is initially introduced varies between a given range to investigate different areas of the search space. In order to evolve the initial generic modularization model, we only need to define one very simple Henshin rule (cf. Figure 3). This rule moves an entity from one module to another. Our tool instantiates the input parameters of the rule with specific entities and modules names. According to the fitness function conformed by the objectives defined, the search engine searches for the optimal assignments of entities into modules. The output of the search is given as (i) the *Optimized Modularization Model*, as well as (ii) the sequence of rule applications and their input parameters.

[1] `http://www.eclipse.org/modeling`

[2] `http://www.eclipse.org/henshin`

[3] MOEA Framework, version 2.8, available from `http://www.moeaframework.org`

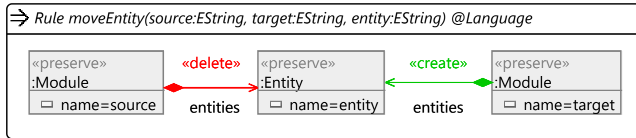

Figure 3. *assignModule* rule.

As for the objectives that compose the fitness function, they can be defined with a configuration language available in our framework, as shown in Listing 1. We can see the definition of the four objectives explained before. In this listing, we show two ways offered by our framework for the definition of objectives. First, objectives *minimize coupling* and *maximize cohesion* use an external calculator with methods implemented for the calculation of the value of the metrics. Second, objectives *maximize number of modules* and *minimize difference between number of entities in modules* are calculated directly in this configuration file. In MOMoT, the user can choose either of the two ways or specify an OCL query to calculate the objectives.

4. Application Study

In this section we apply our generic modularization chain presented in the previous section for modularizing Ecore models. In order to address this, we (i) define the research questions for our study, (ii) explain how Ecore models are translated to our modularization language (cf. Fig. 1), and finally (iii) describe the results obtained by our search approach for real-world Ecore models.

4.1 Research Questions

We are interested in answering the following research questions (RQs).

- *RQ1*. Is the binding between Ecore and the generic modularization metamodel feasible with the proposed approach?
- *RQ2*. How good are the results of the modularization task, i.e., the results of applying the generic modularization strategies?

4.2 Binding between Ecore and the Generic Modularization Metamodel

The first step is to conceptually bind the concepts of the Ecore language with those of our generic modularization language by means of a query-driven model transformation. A simplified version of the Ecore metamodel with the concepts that we take into account is presented in Figure 4. Please note that gray, and therefore unmapped, elements of Ecore are depicted with gray color. We can see that *EPackages* contain *EClasses* and *EDataTypes*. *EClasses* can inherit from other *EClasses* (*eSuperTypes* relationship). At the same time, they contain *EReferences* and *EAttributes*. The former are used to specify relationships among *EClasses*. Therefore, an *EReference* has an *eReferenceType* that points to the end *EClass*. Furthermore, an *EReference* can either be of type *containment* or not. As for *EAttributes*, they are of a specific type (*eAttributeType*), which is specified by an *EDataType* or an *EEnum*.

The homomorphism between the two languages is summarized in Table 1. Bindings for *EPackage*, *EClass*, *EDataType* and *EEnum* are quite straightforward (*EPackage* is mapped to module, all *EClassifiers* are mapped to entity). Let us explain those bindings that produce a relationship in the generic modularization model. As we mentioned in Section 3.1, relationships can have different weights, depending on how strong the relationship is. In an Ecore model, we consider that the containment relationship is the strongest one, since a contained element cannot exist without its container. Therefore, for those *EReferences* that are of type

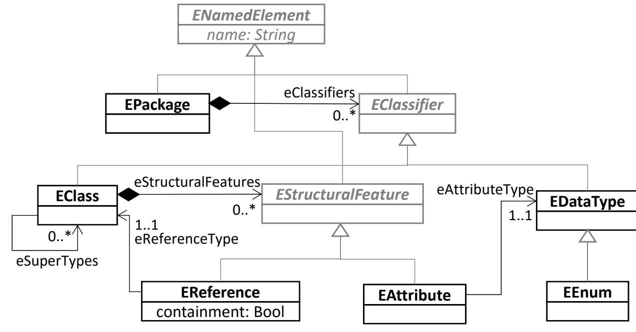

Figure 4. Simplified Ecore Metamodel.

Table 1. Correspondences between Ecore Language and Generic Modularization Language.

Ecore	Gen Module Language
EPackage	Module
EClass	Entity
EDataType	Entity
EEnum	Entity
eSuperType	Relationship (weight=2)
EAttribute	Relationship (weight=1)
EReference (non-containment)	Relationship (weight=1)
EReference (containment)	Relationship (weight=3)

containment, we create a relationship with weight 3 between the entities representing those *EClasses* that act as source and target of the *EReference*. *EReferences* that are not of type *containment* are the weakest ones in our mapping, so we give a weight of 1 to the relationships created from them. As previously mentioned, *EClasses* contain *EAttributes*. The latter, in turn, are typed with *EDataType* or *EEnum*. Therefore, also a relationship is created between an entity representing an *EClass* and those entities representing *EDataTypes* or *EEnums* that are the type of the *EAttributes* of the *EClass*. These relationships are also given weight 1. Finally, there are relationships between those entities representing *EClasses* that have inheritance relationships between them. As inheritance relationships span up type hierarchies, the weight given to such relationships is 2.

According to this mapping, we have implemented an ATL transformation[4] that takes any Ecore model and produces a model conforming to our generic modularization metamodel. The latter model contains as many *Modules* as *EPackages* the Ecore model has, and all the entities and relationships among them are correspondingly created. We implemented the transformation comprising the two steps as explained before. First, helper functions are defining the queries needed to incorporate the concepts of the modularization metamodel in the Ecore metamodel (conceptually one can think about derived properties). Second, we employ one-to-one rules to actually produce the initial modularization models from the Ecore models.

As Ecore models we have used the metamodels of HTML, JAVA, OCL and QVT available in the ATL transforation zoo[5] as they represent middle-sized as well as large metamodels (Kusel et al. 2013). We have transformed them with our approach. The number of modules, entities, and relationships (of each of the three types) obtained for these Ecore models are summarized in Table 2.

[4] All artifacts can be found on our website (Fleck et al. 2016)

[5] `http://www.eclipse.org/atl/atlTransformations/`

Table 2. Ecore models as modularization models.

Model	#Mod	#Ent	#Rel(w=1)	#Rel(w=2)	#Rel(w=3)
HTML	2	62	14	42	7
JAVA	1	132	179	145	129
OCL	2	77	47	73	37
QVT	8	151	199	152	100

At this stage we can already answer RQ1: yes, the binding between Ecore models and the generic modularization models is feasible and has been, in fact, implemented with the proposed approach.

In the following subsection, we describe the application of the search-based model transformation to modularize each of these models.

4.3 Search-Based Optimization

The values for the metrics that define the different objectives before and after we execute the search-based modularization approach are shown in Table 3, where the direction of the arrow indicates the objective direction (whether it should be maximized or minimized).

In order to perform the search to find the optimal modularizations, we make use of our MOMoT tool (Fleck et al. 2015) as explained before. The objectives that we use as input are those described in Section 3.3. The result of a search-based algorithm is the set of Pareto optimal solutions. Under Pareto optimality (Harman 2007), one solution is considered better than another if it is better according to at least one of the individual objective functions and no worse according to all the others. The algorithms used in SBSE apply the notion of Pareto optimality during the search to yield a set of non-dominated solutions. Each non-dominated solution can be viewed as an optimal trade-off between all objective functions where no solution in the set is better or worse than another solution in the set.

Some works (Branke et al. 2004) argue that the most interesting solutions of the Pareto-optimal front are solutions where a small improvement in one objective would lead to a large deterioration in at least one other objective. These solutions are sometimes also called "knees". In order to show some values for the solutions we get, we have extracted the knee point solution for each of the case studies by calculating the proper utilities for each solution (Shukla et al. 2013). The solution shown in Table 3 after the search displays the value of the different objectives for the solution considered as knee point after executing the search.

Let us respond to RQ2 by studying the numbers in the table. As we can see, the value of coupling (COP) for the first three example models before the optimization is 0. In the case of JAVA, this is obvious as there is only one module (MOD). Regarding HTML and OCL, there are two modules, where one of those only has 3 to 4 entities without any dependency with any entity from the other module. This is due to the fact that these modules contain only the primitive types such as Boolean or Integer. As for QVT, since there are 8 modules, the value of coupling is larger than 0. These modules come from the packages in the metamodel and are for example QVT Template, Imperative OCL, EMOF, or QVT Operational. As for the difference between the minimum and maximum number of entities in each module (DIF), this value is quite high for HTML, OCL, and QVT, i.e., the entities between the modules are not distributed equally. In the JAVA language, on the other hand, this value does not make sense as there is only one module. Finally, regarding cohesion (COH), all initial languages present a very high to optimal value due to the fact that most, if not all, entities are in the same module.

Table 3. Objectives values before and after optimization.

Example	Model	COH ↑	COP ↓	DIF ↓	MOD ↑
HTML	Before	119	0	56	2
	After	101	18	31	5
JAVA	Before	856	0	-	1
	After	517	339	2	7
OCL	Before	257	0	69	2
	After	262	42	45	4
QVT	Before	587	216	38	8
	After	448	355	2	8

If we investigate the values after the optimization is performed, we see that in HTML we have now 5 modules, 7 in JAVA, 4 in OCL and we keep the same number of modules in QVT. Since in most cases we have more modules than before the optimization, the values of cohesion and coupling have gotten worse, as it is obvious. However, the value of DIF is improved to a great extent. Therefore, we are sacrificing coupling and cohesion in favor of having several balanced modules. This is, in fact, the motivation for this study: to have several modules where entities are distributed equally number-wise, as better as possible regarding cohesion and coupling values.

Please note that these are the results produced by step 3 of our approach, which conform to the optimized generic modularization model. The last step would be to transform these solutions back to the original modeling language. This is left as future work and would require to inverse the transformation produced in step 1 and 2. As we are using a query structured approach and one-to-one mappings for step 1 and 2, we see here no particular challenges to inverse this transformation.

5. Related Work

Concerning related work, we shortly summarize work in software modularization in general and then discuss specific model modularization approaches.

As already mentioned in Section 2, several approaches have been proposed for software modularization considering programming artifacts (Praditwong et al. 2011; Mkaouer et al. 2015). In this paper, we follow the same search-based spirit, but aim to provide a generic representation of the modularization problem which may be reused for several modeling languages.

There are already some approaches which aim in splitting large models into more manageable chunks. First of all, EMF Splitter (Garmendia et al. 2014) provides means to split large EMF models based on metamodel annotations. Here the user has to come up with a meaningful configuration for the model splitting and the main use case is to deal with very large models. There are no design quality aspects considered as in our approach. The same is true concerning Fragmenta (Amálio et al. 2015) which is a theory on how to fragment models in order to ensure technical constraints. In (Strüber et al. 2014) an approach for extracting submodels based on textual description is presented. However, the usage of quality metrics such as done in this paper, is mentioned only as subject to future work. A graph clustering approach has been presented in (Strüber et al. 2013) to modularize large metamodels. While in our approach we are not proposing a specific transformation as it is done in (Strüber et al. 2013), an interesting line of future research is to compare the capabilities of the graph clustering algorithm with respect to the search-based meta-heuristic approaches. Finally, in (Henriksson et al. 2007), an approach is presented how modularity can be added to existing languages. While we are assuming that languages in our setting already offer some kind of modularization concept, it would be interesting to integrate the modularization concepts proposed by (Henriksson et al. 2007) in our modularization metamodel in the future.

6. Conclusions

In this paper, we have presented a first approach to deal with modularization of models in a generic and reusable way. We have achieved this goal by combining several kinds of transformation approaches: generic, query-structured, and search-based transformations. While the results seem already promising for the Ecore case study, several future lines of research are needed to deal with the specific characteristics of modeling approaches such as having megamodels, multi-viewpoint modeling, as well as having separated but highly interconnected artifacts such as models, metamodels, and transformations, to name just a few examples. Finally, we plan to study more powerful modularization approaches such as aspect-orientation and concern-oriented ones as well as how to support other related approaches such as model merging, model linking, and model splitting with our search-based transformations.

Acknowledgments

This work has been partially funded by the Christian Doppler Forschungsgesellschaft, by the BMWFW (Austria), by the European Commission (FEDER) and Spanish Government under CICYT project TAPAS (TIN2012-32273), and by the Andalusian Government projects THEOS (TIC-5906) and COPAS (P12- TIC-1867).

References

O. Alam, J. Kienzle, and G. Mussbacher. Concern-oriented software design. In *Proceedings of the 16th International Conference Model-Driven Engineering Languages and Systems (MoDELS)*, volume 8107 of *LNCS*, pages 604–621. Springer, 2013.

N. Amálio, J. de Lara, and E. Guerra. Fragmenta: A theory of fragmentation for MDE. In *Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 106–115. IEEE, 2015.

T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

C. Atkinson, D. Stoll, and P. Bostan. Supporting view-based development through orthographic software modeling. In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 71–86. INSTICC Press, 2009.

D. Blouin, Y. Eustache, and J. Diguet. Extensible global model management with meta-model subsets and model synchronization. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages (GEMOC) @ MoDELS*, volume 1236 of *CEUR Workshop Proceedings*, pages 43–52. CEUR-WS.org, 2014.

M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

J. Branke, K. Deb, H. Dierolf, and M. Osswald. Finding Knees in Multi-objective Optimization. In *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 722–731. Springer, 2004.

J. S. Cuadrado, E. Guerra, and J. de Lara. Generic model transformations: *Write Once, Reuse Everywhere*. In *Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT)*, volume 6707 of *LCS*, pages 62–77. Springer, 2011.

K. Deb and H. Jain. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation*, 2014.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

K. Deb, M. Mohan, and S. Mishra. A Fast Multi-objective Evolutionary Algorithm for Finding Well-Spread Pareto-Optimal Solutions. Technical Report 2003002, Indian Institute of Technology Kanpur, 2003.

M. Fleck, J. Troya, and M. Wimmer. Marrying Search-based Optimization and Model Transformation Technology. In *Proceedings of the 1st North American Search Based Software Engineering Symposium (NasBASE)*, pages 1–16, 2015.

M. Fleck, J. Troya, and M. Wimmer. Transformation Chain for Ecore models, 2016. Available at `http://martin-fleck.github.io/momot/casestudy/generic_modularization/`.

A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF splitter: A structured approach to EMF modularity. In *Proceedings of the 3rd Workshop on Extreme Modeling (XM)*, volume 1239 of *CEUR Workshop Proceedings*, pages 22–31. CEUR-WS.org, 2014.

H. Gholizadeh, Z. Diskin, and T. Maibaum. A query structured approach for model transformation. In *Proceedings of the Workshop on Analysis of Model Transformations (AMT) @ MoDELS*, volume 1277 of *CEUR Workshop Proceedings*, pages 54–63. CEUR-WS.org, 2014.

M. Harman. The Current State and Future of Search Based Software Engineering. In *Proceedings of the Workshop on the Future of Software Engineering (FOSE'07) @ ICSE*, pages 342–357. IEEE Computer Society, 2007.

J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware - adding modularity to your language of choice. *Journal of Object Technology*, 6(9):127–146, 2007.

A. Kusel, J. Schoenboeck, M. Wimmer, W. Retschitzegger, W. Schwinger, and G. Kappel. Reality check for model transformation reuse: The ATL transformation zoo case study. In *Proceedings of the 2nd Workshop on Analysis of Model Transformations (AMT) @ MoDELS*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-objective software remodularization using NSGA-III. *ACM Trans. Softw. Eng. Methodol.*, 24(3):17:1–17:45, 2015.

K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Trans. Software Eng.*, 37(2): 264–282, 2011.

H. A. Reijers and J. Mendling. Modularity in process models: Review and effects. In *Proceedings of the 6th International Conference on Business Process Management (BPM)*, volume 5240 of *LNCS*, pages 20–35. Springer, 2008.

J. Reimann, M. Seifert, and U. Aßmann. Role-based generic model refactoring. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 6395 of *LNCS*, pages 78–92. Springer, 2010.

P. Shukla, M. A. Braun, and H. Schmeck. Theory and Algorithms for Finding Knees. In *Evolutionary Multi-Criterion Optimization*, volume 7811 of *LNCS*, pages 156–170. Springer, 2013.

D. Strüber, M. Selter, and G. Taentzer. Tool support for clustering large meta-models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering (BigMDE) @ STAF*, pages 1–7. ACM, 2013.

D. Strüber, J. Rubin, G. Taentzer, and M. Chechik. Splitting models using information retrieval and model crawling techniques. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *LNCS*, pages 47–62. Springer, 2014.

M. Wimmer, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, J. S. Cuadrado, E. Guerra, and J. de Lara. Reusing model transformations across heterogeneous metamodels. *ECEASST*, 50:1–13, 2011a.

M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer. A survey on uml-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):1–28, 2011b.

E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1st edition, 1979.