Preprint of: Reiner Jung, Robert Heinrich, Wilhelm Hasselbring: "GECO: A Generator Composition Approach for Aspect-Oriented DSLs". In: P. Van Gorp and G. Engels (Eds.): ICMT 2016, LNCS 9765, pp. 141–156, 2016. DOI: 10.1007/978-3-319-42064-6 10

GECO: A Generator Composition Approach for Aspect-Oriented DSLs

Reiner Jung[1], Robert Heinrich[2], and Wilhelm Hasselbring[1]

[1]Kiel University, Christian-Albrechts-Platz 4, Kiel, Germany
{reiner.jung,hasselbring}@email.uni-kiel.de,
[2]Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, Germany
robert.heinrich@kit.edu

Abstract. Code and model generators that are employed in model-driven engineering usually face challenges caused by complexity and tight coupling of generator implementations, particularly when multiple metamodels are involved. As a consequence maintenance, evolution and reuse of generators is expensive and error-prone.
We address these challenges with a two fold approach for generator composition, called GECO, which subdivides generators in fragments and modules. (1) fragments are combined utilizing megamodel patterns. These patterns are based on the relationship between base and aspect metamodel, and define that each fragment relates only to one source and target metamodel. (2) fragments are modularized along transformation aspects, such as model navigation, and metamodel semantics.
We evaluate our approach with two case studies from different domains. The obtained generators are assessed with modularity and complexity metrics, covering architecture and method level. Our results show that the generator modularity is preserved during evolution utilizing GECO.

1 Introduction

Models play a central role in Model-driven engineering (MDE). They are used to specify the different views and aspects of a software system separately in a more abstract way than programming code [35]. Models conform to metamodels, which define, supplemented by constraints, the abstract syntax and semantics of models. Domain-specific languages (DSLs) are used to create models. They provide a corresponding concrete syntax and semantics for metamodels [7].

The notion of different views and aspects is addressed in both multi-view modeling (MVM) [3,23] and aspect-oriented modeling (AOM) [25]. Both modeling approaches use separate models and metamodels to specify different parts of a software system, like data structures, architecture, behavior, and monitoring. These models are considered *source models*. They are transformed into *target models* including program code by generators [28]. In our context, a generator is an exogenous and vertical transformation [28] supplemented by model serialization and deserialization. Therefore, they are essential for MDE [27]. Especially in AOM and MVM, generators may have to process multiple source models, representing different aspects and views, integrate their information and store the

result in target models. This makes generators complex artifacts, particularly if a generator depends on multiple source and target metamodels.

Software systems evolve over time to accommodate changes in requirements, platform and environment. Each change can affect the syntax and semantics of source and target metamodels, requiring generators to be adapted and modified. While DSLs can be altered quickly and reused in other software projects [7], the complexity of generators makes changes to them cumbersome and can result in architecture degradation. This also applies to solutions where metamodel changes are handled by supplemental transformations and model adapters. The iterative addition and modification of such adapters would also lead to a complex architecture. This hinders the evolution and reuse of generators.

Present approaches address architecture degradation either with transformations chains composed of small transformations [36], or with partitioning transformations along arbitrary boundaries [8]. However, chains do not address the diversity of different source metamodels and the partitioning focuses only on single model inputs. Furthermore, these approaches do not discuss evolution.

We circumvent these limitations with our technology-independent generator composition approach (GECO) [18] by

(a) partitioning generators into generator fragments along the types of views and aspects of the application domain,
(b) modularizing the fragments along language features, e.g., typing, and
(c) providing a method to combine the output of fragments.

GECO uses our approach for metamodel evolution [22] which divides and organizes metamodels along views, aspects, and metamodel semantics. Furthermore, we supplemented GECO with tooling and libraries (see also [20]) to support its design principles and methods, which were also used in the evaluation.

We assessed GECO with two case studies. The first is based on the information system of the Common Component Modeling Example (CoCoME) [31] specified with multiple DSLs and incorporates an existing generator. The second is based on an industry project for electronic railway control centers, named MENGES [13]. In both, we evolved the DSLs and adapted the generators accordingly.

The remainder of this paper is organized as follows: Section 2 introduces AOM as foundation of GECO. Section 3 provides the running example. Section 4 explains our approach. Section 5 reports on the evaluation. Section 6 discusses the related work. Finally, Section 7 provides our conclusion and outlook.

2 Aspect-Oriented and Multi-View Modeling

The GECO approach is founded on aspect-oriented (AOM) and multi-view modeling (MVM) together with a categorization and decomposition of metamodels based on semantic properties. Therefore, we briefly introduce these four topics.

Metamodels defined with EMOF [30] use classes and references between classes to express concepts. References can express containment, association and aggregation [22]. Depending on the purpose of a metamodel, specific patterns occur to

express type structures (e.g., component types, states of workflow graphs), expressions, mappings, queries, and many more [22]. GECO uses these patterns to decompose metamodels and suggest module boundaries for generator fragments.

MVM focuses on the different views an engineer has on a software system, like architecture, component types, interaction, behavior, and data models [3,23]. Each view can have its own metamodel covering only the concepts of the specific view. Views may relate to other views [3]. For example, a behavior model expresses the interpretation and manipulation of data. Therefore, it must be able to access the data model. In this example, the behavior model depends on the data model while the data model is independent (cf. Figure 1). These properties of dependence and independence can either be seen from a project point of view for all metamodels used in a software project or be limited on two individual metamodels. From the general perspective most metamodels depend on others, e.g., architecture depends on component types, which depend on data types. For GECO, we focus on the relationship of metamodel pairs and interpret dependence and independence as two roles a metamodel can have [22]. In each relationship, we require that one metamodel is the independent and the other is the dependent one. Furthermore, we discourage the use of cyclic dependencies of metamodels representing different views, as it results in more complex generators. However, such dependencies can be addresses with additional fragments realizing partial transformations and intermediate models (cf. [8]).

AOM addresses the modeling of main and cross-cutting concerns. The main concern of a software system is its primary function, e.g., performing a purchase operation. A cross-cutting concern is a concern which must be introduced at different places in the main concern. For example, in performance monitoring logging functionality must be added to record entry and exit times of operations. In AOM, cross-cuttings concern are expressed in a separated aspect model and the main concern is defined in a base model [4,23]. The aspect can further be distinguished in a pointcut and advice. which define the points fo extension and the extension, respectively. In general, the pointcut model comprises of references to the advices and queries over the base model to identify elements which are to be extended [24]. The collected references to elements are called join points.

Similar to MVM, the distinction in advice and base metamodel describe two roles in a relationship [22]. For example, there are three metamodels for application behavior, access control, and monitoring, where access control is an aspect applied to the behavior and monitoring is applied to access control. In this case, access control has different roles depending on the context.

Weaving In aspect-oriented programming, the language of the advice represents a subset of the base language which allows to directly introduce the advice into the main function before execution. This introduction is called weaving. In AOM, a similar process can be used when the advice metamodel is a subset of the base metamodel. Weaving approaches, like the Kermeta weaver [29] and AMW [9], go

even further and do not only define additions, but also specify which elements must be replaced or removed and how references must be fixed.

3 Illustrative Example

We use as an illustrative example, an excerpt of the generator design used in our first case study. The case study implements the enterprise part of a software system for a supermarket chain, called Common Component Modeling Example (CoCoME) [14]. CoCoME comprises cash desks in stores, multiple stores with a store server, and a central enterprise server. It covers typical use cases of software systems and incorporates embedded and enterprise software.

We modeled CoCoME with the Palladio Component Model (PCM) [5]. PCM is a metamodel for architecture description and performance prediction. It covers views for component type specification and assembly, which we used in the case study. The PCM is supplemented with a DSL for Behavior, allowing to model operations declared in a PCM model. For persistence and data modeling, we created a data type language (DTL). We monitored CoCoME for a performance evaluation with the instrumentation aspect language (IAL) [21].

We describe these metamodels, their relationship and the associated transformations in Figure 1, which is also called a megamodel [11]. In Figure 1, metamodels are depicted as boxes. The edges between the boxes represent references (arrow with open tip ⟶) and transformations (arrow with filled tip ⟶).

References between metamodels are labeled to indicate their purpose. Essentially, they are the aggregate of references between classes of two metamodels with the same direction, e.g., the reference from IAL to PCM represent references to PCM-operations. Transformations are labeled with the letter T and a subscript name corresponding to the implementing generator fragment. For example, the fragment named $\mathsf{T}_{\mathsf{DTL}}$ generates Java entity classes from DTL specifications.

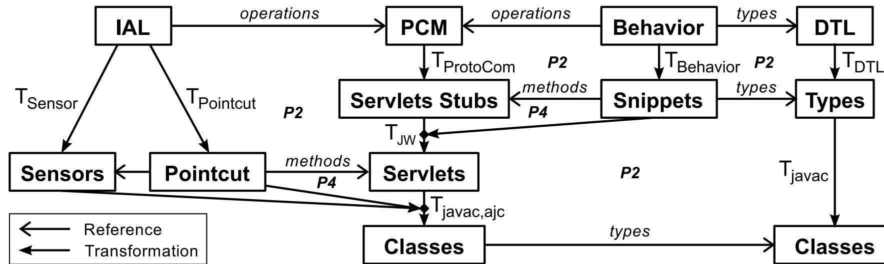

Fig. 1. Generator megamodel excerpt with the main fragments, metamodels, and their relationships. The labels P2 and P4 refer to patterns introduced in Section 4.1.

The ProtoCom generator ($\mathsf{T}_{\mathsf{ProtoCom}}$) [12] is used to generate stubs for Enterprise Java Beans and Java Servlets from PCM component declarations. These stubs are complemented by code snippets provided by $\mathsf{T}_{\mathsf{Behavior}}$. As Figure 1

illustrates, the *operations* references are mapped to *methods* references. Subsequently, the weaver T_{JW} weaves snippets and stubs. The monitoring is realized with two fragments for sensors (T_{Sensor}) and pointcuts ($T_{Pointcut}$), respectively. The sensors are Java classes and the pointcuts are stored in a file for the AspectJ weaver (T_{ajc}). As the weaver operates on byte code, classes are first compiled and then woven. For reasons of brevity we express these two steps with $T_{javac,ajc}$.

4 The GECO Approach

The GECO approach addresses generator development and mitigates issues, such as architecture and code degradation, which harm the evolution and reuse of generators. GECO is technology agnostic, as it can be applied to any modeling and generation technology and paradigm. It covers both code and model generators. However, for our evaluation, we primarily used the Xtend templating language and EMF to realize metamodels and models.

We designed GECO with AOM [23] and MWM [3] approaches in mind. In both references point from one metamodel to another (see Section 2). For reasons of brevity, we mostly refer to the term AOM in the remainder of this paper.

In GECO, generators are modularized on two levels. They are split up into smaller generators, called fragments, which are further subdivided into modules. Each fragment is defined with only one source and target metamodel, and can often be realized with one transformation. As metamodels may not be self-contained and may cover multiple views and aspects, fragments can be designed for only a partition of a source metamodel, especially for partitions that fulfill the criteria of an aspect or base metamodel [22]. This implies that it is not necessary to have de facto multiple metamodels to developed with GECO. It is sufficient to be able to partition the metamodel along the relationships of base and aspect models, and independent and dependent views, respectively.

The key challenges for GECO are the decomposition of generators along concerns reflected in metamodels and partitions of metamodels, the mapping of source to target model join points, the construction of model traces used to construct this mapping, and the modularization of fragments.

4.1 Basic Generator Megamodel Patterns

In GECO, code generation is realized by a set of fragments which are combined to provide code generation for the different models and metamodels used in a software project. The actual integration of fragments depends on the used technology and the way models are passed on from fragment to fragment. In our example, fragment execution is controlled by the Eclipse build system and models are passed via the file system as serialized models.

The modularization of a generator in GECO depends on the partitioning of metamodels into views and aspects, like Behavior and the IAL in our example (see Figure 1). Both reference the PCM as their independent view and base model,

respectively. In projects with multiple metamodels, like our example, code generation involves multiple fragments processing and combining information from different models. All these fragments can be interrelated, resulting in a web of metamodels, their relationships, and fragments which can be represented with a megamodel [11], like the one in Figure 1. In this paper, we disentangle these relationships of metamodels and fragments based on four megamodel patterns, depicted in Figure 2. We deduced these megamodel patterns from a set of minimal patterns involving at most two source and two target metamodels. The fragments are represented by transformations to abstract from technical details.

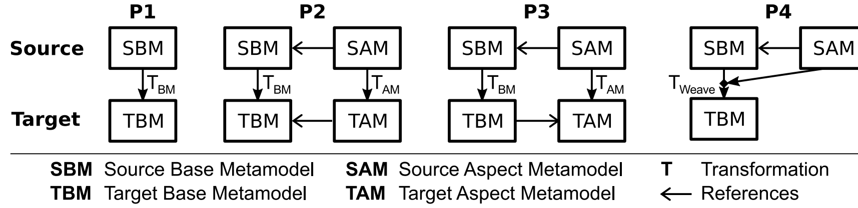

Fig. 2. Four megamodel patterns for base and aspect metamodel with their respective transformations and target metamodels (trace models omitted).

Pattern P1 is a simple transformation with one source and one target model. It is used to express independent transformations. *Pattern P2* describes that source model references are mapped to target model references preserving that information. In our example, this pattern is used four times (cf. Figure 1). *Pattern P3* reflects the situation where the direction of references is inverted from source to target level. This may happen to express aspect invocations on the target model level when the target metamodel does not support aspect weaving. *Pattern P4* covers weaving of aspect and base model. As GECO is technology agnostic, different weavers can be used, e.g., the weaver of Kermeta [29].

4.2 Combining Aspect and Base Model Fragments

In patterns P2 and P3 model traces must be exchanged between fragments to compute references on the target model level (see Figure 3). Trace models (*TRM*) are used for this exchange. Depending on the transformation language, the *TRM* generation must be explicitly implemented, or can be added automatically [17].

Pattern P2 The fragments T_{BM} and T_{AM} produce main output models conforming to a target base metamodel (*TBM*) and a target aspect metamodel (*TAM*), respectively. As the references between *SAM* and *SBM* must be mapped to the target level, T_{AM} requires a trace model relating *TBM* to *SBM* nodes. The *TRM* can be generated by T_{BM} as a second output, or can be computed by a surrogate transformation TR_{BM}. Such surrogate is necessary when adding a second output to T_{BM} is not feasible, e.g., the source code is not available.

In our example, $\mathsf{T}_{\mathsf{ProtoCom}}$ provides a trace model and $\mathsf{T}_{\mathsf{DTL}}$ uses the package structure of the source model also for the target model which makes a trace model obsolete, as T_{BL} can use the package information for the source model.

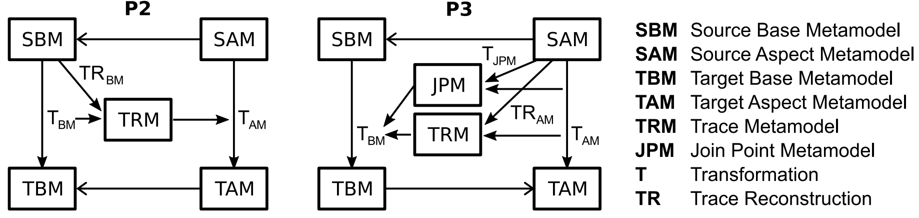

Fig. 3. Illustration of generator fragment compositions

Pattern P3 In contrast to P2, the reference direction is inverted and then mapped to the target model level. Therefore, model traces from the aspect and join point information must be generated and passed to T_{BM}. The trace model is produced by T_{AM} or a surrogate TR_{AM}. Similarly, join points are computed by T_{AM} or a surrogate T_{JPM}. The join points are required to infer the inverse reference origins which are placed in the target base model. The trace model is used to compute the reference destinations in the target aspect model nodes.

Achieving Model Traceability Model traces can be represented as relations between source and target model nodes, e.g., $TRM \subseteq SBM \times TBM$. They can be produced with constructive and recovery approaches [37]. The latter use either deterministic algorithms or heuristics [33] to find matches. Heuristics do not have predictable output and deterministic approaches use attribute value similarities to find matches, which may result in wrong and missing traces. Therefore, only constructive approaches can be used to create trace models for GECO. They are generated either by the fragment itself or by a supplement trace model transformation. The first approach can lead to a more complex fragment source code, except for transformation languages which allow to add this feature automatically [17]. The second approach circumvents this complexity issue with a separate transformation and allows to integrate legacy generators where code alterations are not feasible. However, then two transformations must be maintained.

4.3 Computing Target Join Points

In aspect-oriented metamodels, join points can be expressed as direct references [22] or they can be specified with pointcuts [21,29] which are used to compute joint points. In P2 and P3 these join points must be translated from source to target level. Due to space constraints we only describe their computation for pattern P2. However, the computation for P3 can be achieved in a similar way.

This translation is achieved in two steps where source level join points ($JP_S \subseteq SAM \times SBM$) are translated into their target counterparts ($JP_T \subseteq TAM \times TBM$). First, for each reference destination d_{s_i} in $(s_s, d_s) \in JP_S$ a set of intermediate join points is computed $JP_{I_i} = \{(s_{s_i}, n_t) | (n_s, n_t) \in TRM \wedge d_{s_i} = n_t\}$. Second, during the transformation of SAM to TAM, T_{AM} infers trace information which is used to compute target level join points from all JP_{I_i}.

As trace models may contain traces to nodes with different semantics which might not be well suited for weaving, the remaining set JP_T must be checked accordingly. For example, a component type is transformed into a class with

attributes and access methods. A join point representing an injection of a monitoring sensor should reference the methods and not the attributes. Therefore, JP_T must be filtered for target nodes conforming to method declarations.

4.4 Internal Structure of Fragments

The megamodel patterns address the combination of fragments. In contrast, the inner structure of fragments also affects reuse and evolution [32], which can be improved with modularization. We propose a twofold approach to achieve modularization along the two dimensions functionality and metamodel semantics.

a) Functionality Fragments can be modularized along common functionality (see Figure 4), like source model query, aggregation and evaluation, state, target model creation, name resolving and trace handling, and control (cf. [8,28]).

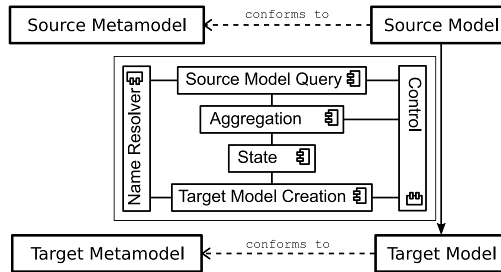

Fig. 4. General functional decomposition of a transformation (cf. [8,28])

The advantage of this decomposition is that it can be applied to any fragment regardless of the actual transformation. It also follows the decomposition of software along concerns. Furthermore, it allows to improve and test functionality separately. Its disadvantage is caused by metamodel evolution. For example, we add database queries to the Behavior DSL (cf. Section 3). This affects almost every module from model query to target model creation (see Figure 4). However, the central idea of modularization is to keep modifications local, which is not the case in the example metamodel change and similar alterations.

b) Semantics Alternatively, fragments can be decomposed along the categorization of metamodel semantics [8,22], like expressions, typing and initialization. Adding database queries to the Behavior language, like above, would affect only those modules related to expression and statement handling. The query generation itself could even be implemented in a separate module keeping the modifications in the other modules minimal (cf. [19]).

5 Evaluation

In this evaluation we focus on the generator evolution for two reasons: (a) According to a qualitative industry survey we conducted, evolvability is more important than reuse [19]. (b) Evaluation of reuse requires multiple case studies

sharing common metamodel parts, like product lines. Furthermore, reuse and evolution both depend on modularization which is addressed in our evaluation.

5.1 Evaluation Approach

We concentrate on two main goals addressing the overall feasibility and efficiency of the approach focusing on development and evolution based on two case studies. First, we evaluate the feasibility of our approach by using GECO to implement the given case studies. Second, we evaluate the efficiency of our approach from the perspective of a developer, focusing on the support GECO provides for construction and evolvability of generators.

For generator construction, as for any software architecture, modularization [16] is the key concept used to divide a larger problem into simpler modules that address only one concern of the complete generator. Therefore, modularity is important to support construction. Evolution requires modularity, extensibility, and changeability of modules [16], as new features are introduced, altered, and removed over time. Modularity supports extensibility and changeability due to the lower complexity of the modules and low coupling [16]. To show that GECO helps to keep the modularity of generators intact, we must evaluate how multiple iterations of extending and changing effect the modularity of a generator.

The modularity of a software system is determined by the cohesion, coupling and complexity [1,2]. Good modularity of a system is indicated by high inner cohesion and low inter-module coupling [16]. The greater the distance between complexity and coupling of the system, the better the modularity, as complexity refers to the complete system and coupling only to the inter-module dependencies. Extensibility and changeability are affected by modularity and the inner complexity of modules [16]. Lower complexity improves code readability, improving code comprehension, which reduces the potential for code degradation.

To determine the three properties, modularity, extensibility and changeability, we measure complexity, module cohesion and coupling. These measurements depend on many factors including size and complexity of the requirements realized in each evolution step. Therefore, it is impossible to define fixed levels to indicate a good quality. However, we can compare different generator revision and implementations, which allow us to evaluate whether the alterations affected complexity, cohesion, and coupling.

We utilize (hyper)graph based entropy metrics [1,2] and cyclomatic complexity [26] on code level. The entropy metrics allow us to focus on the information density of software which is considered to be a close approximation of the cognitive effort necessary to understand the software (cf. [2]). The entropy metrics measure only classes, which are represented as modules, methods (nodes), and method calls (hyperedges). This allows to hide complexity inside method bodies which are not represented in the hypergraph. Therefore, we monitor the method complexity with the cyclomatic complexity metric to detect changes, which indicate an complexity transfer. We are aware of the limitation of cyclomatic complexity applied to complete software systems [34]. However, we only

test whether the complexity of a method has changed (number of branches and loops). Therefore, the rationale of [34] does not apply in our case.

5.2 Setup of Case Studies

The first case study involves an information system. It evaluates the integration of existing generators with newly written fragments, and the evolution of fragments. The second case study focuses on evolution by reproducing the implementation of a generator from an industry project.

Information System Case Study This case study is based on CoCoME (see Section 3). We defined the generator's architecture for CoCoME based on the megamodel patterns, indicated by the labels P2 and P4 in Figure 1. For the evaluation, the megamodel from Figure 1 is extended by a DSL and fragment for monitoring event types [21] and different sensor technologies. The fragments used in this case study are implemented with Xtend [7].

For the evaluation, we created an initial version of the Behavior DSL and generator. Iteratively, the first version was extended to support different component types, and database access. For all revisions, we measured complexity, cohesion and coupling as explained above. In addition, we counted the number of class files, modules, nodes, and edges of the hypergraph.

Control System Case Study The control system case study is based on MENGES which comprises DSLs and a generator for the domain of railway control centers based on programmable logic controllers (PLC) [13]. The goal of MENGES was to provide developers with DSLs which fit their abstractions used in previous railway control center implementations. This includes architecture, communication protocols, conversion of external signal into discrete internal values, behavior (automata and workflows), data types, and configuration. The original DSLs were developed with Xtext and its generator with the transformation and templating language Xtend. The original generator produces code for the PLC language *Structured Text* (ST) [15] and serializes it in an XML file. For the evaluation, we reimplemented this old generator using GECO. To avoid implementing more efficient algorithms than the original developers, we reused their code adapted to the module and fragment structure of the new generator.

During the development of the old generator (G_{old}), language features were added, removed, and changed based on user feedback and tests of the DSLs and the generator. In the evaluation, we used the original documentation and code to extract features for 14 revisions of the generator. As we simulated the development of the new generator (G_{new}), it was necessary to extract only those features and changes of G_{old} which happened in the next revision. Therefore, we extracted the features of Revision 1 of the G_{old} and implemented these in G_{new}. Then we went to the next revision and repeated the process. Through this process, the developers of G_{new} gained only knowledge of features and changes the original developers had implemented in the corresponding revision.

Initially, the generator supported type structures (Revision 1-4). Later it was extended to support expressions, statements, and automata. The output is a combination of an XML-DOM and function implementations in ST. Therefore, the generator combines model-to-text and model-to-model transformations. G_{old} and G_{new} mainly differ in the modularization. Details can be found in [19].

5.3 Information System Case Study Results

This case study assesses the feasibility of GECO to model the combination of different fragments and the construction and evolution of a fragment. The first part is shown by modeling the composed generator for CoCoME with GECO (cf. Figure 1). The second part is described in this section by evolving the $T_{Behavior}$ fragment in four revisions. The columns of Table 1 show the code revision, the git revision tag, the number of classes, excluding data types, frameworks, and anonymous classes. The number of modules refer to the number of classes mapped to the hypergraph. This includes anonymous and framework classes used by the fragments. The nodes represent the fragment and the used framework methods, and the edges express method calls and access to shared data objects. The remaining columns depict values for the entropy metrics.

Table 1. Measurements of the behavior generator fragment of the CoCoME case study

$T_{Behavior}$	Revision git	# of Class	Modules	Nodes	Edges	Size	Complexity	Cohesion	Coupling
r1	be2dafbc53a	6	16	56	125	314.25	802.70	0.043709	594.93
r2	83acc26830d	6	17	57	127	321.54	813.99	0.043965	605.90
r3	0961df26eb7	6	17	58	134	328.86	873.37	0.043965	654.83
r4	0c87a9e84c4	6	17	64	156	373.23	1041.43	0.042075	781.88

For this case study, we implemented an initial version of $T_{Behavior}$ and performed three evolution steps on the language and fragment. Revision r1 supports the specification of operation bodies operating on input data and internal state. In revision r2, we added support to mark a component as stateful or stateless, which has significant effects on the scalability of components. To support this new feature one template method had to be extended. Revision r3 added support for special methods called on initialization and destruction of components. Finally, we added constructs for database access to the DSL, supporting JPA.

The measurements [19] (see Table 1) show, adding features increase size and complexity of the overall system. We can see that the intra module cohesion changes are minimal for the first three revisions, which indicates that the inner structure of the modules was not changed significantly. Only the support for database access has a significant effect. This is due to the fact that the new statements were added to the expressions module instead to a separate module.

5.4 Control System Case Study Results

In this case study we performed 14 evolution steps and measured the number of classes, modules, nodes, and edges (counting metrics), as well as, size, complexity, coupling, and cohesion (entropy metrics) for both the old and the new generator (cf. replication package [19]). Due to size constraints, we selected the most significant counting and entropy metrics. We choose module and edge count, as the module count includes classes and the edge count represents the interconnectedness of the hypergraph. Furthermore, we omitted the cohesion metric, as it shows a steady difference between both generators (G_{old} has only 60.29% of G_{new}'s cohesion in Revisions 7 to 14).

Figure 5 shows slow growth in all measures over the first 4 revisions. The growth of G_{new} is minimal, as it starts with dedicated classes for each kind of type the DSLs provide. In Revision 5, MENGES added support for expressions which is a complex endeavor, especially as the DSLs provide object-oriented constructs, but the target language is only imperative. In G_{old}, this effort resulted in many more modules (Revisions 4 to 6) and triggered a large refactoring step (Revision 6 to 8), which resulted in a minor fluctuation in the number of modules (Revisions 6, 7, 8). For G_{new}, this was not necessary at this point.

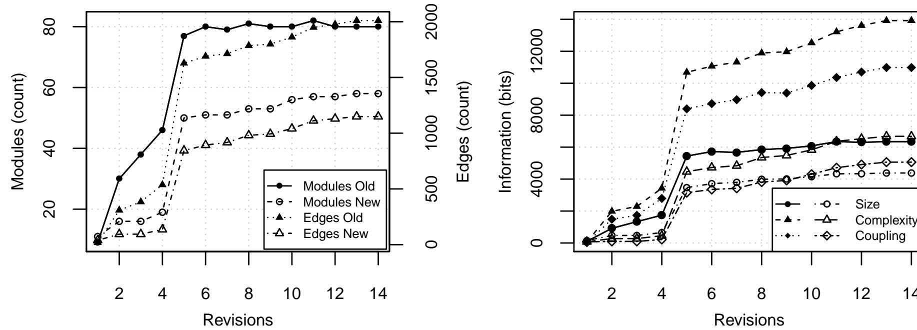

Fig. 5. Counting (left) and information (right) measurements of the control center case study; filled and empty symbols represent G_{old} and G_{new}, respectively

The remaining Revisions (8 to 14) show for both generators continuous growth. However, the increase in size, complexity, and coupling are smaller for G_{new} than for G_{old}. The only difference is the number of modules, which increase in G_{new} after Revision 10, which is caused by factoring out the generation of actions and predicates in separate fragments. G_{old} decreases due to refactoring.

Overall, G_{new} has better (lower) values for all metrics over the complete evaluation than G_{old}. As we reused method implementations from the original code in G_{new} to avoid a result bias based on a different coding style, these better values are not based on coding style. In the end G_{old} was 2.08 times more complex, 2.17 times more intensely coupled, used 1.40 times more nodes, and 1.75 times more edges. This allows the conclusion that GECO has a positive effect on generator development and evolution.

6 Related Work

AOM is an actively researched topic in MDE. Many generation approaches focus on the definition of aspects and the weaving of models, which are collected in a mapping study [27]. Most prominent are approaches based on the Formal Design Analysis Framework [6], UML, and reusable aspect models [29]. They use the UML as source language, and Java and AspectJ as target languages. They aim for the reusability of aspect models and generators. While some approaches use stereotypes or profiles to identify aspects, they neither support profiles for their base and aspect models nor address domain-specific languages. The weaving of aspects is controlled by direct references or model-subgraphs formulating pointcuts. Unlike our approach, theirs do not address the construction of generators.

In a recent survey on aspect-oriented domain specific languages (AODSL), 22 different AODSLs with generators were analyzed [10]. A key challenge of these DSLs is the integration of their aspect generator in the base language generator. Some approaches extend a base language generator in an ad-hoc manner, hindering reuse of the AODSL generator and maintainability of both generators [10]. Two AODSL frameworks use an extensible base language generator for additions by AODSL generators. However, they have multiple shortcomings compared to GECO: (a) they do not address the integration of multiple AODSLs and cascading scenarios with multiple weaving stages, which appear in our example (see Section 3) and case study. (b) they support only their own base language. (c) they do not provide a modularization approach for fragments. (d) they introduce their own frameworks making them not framework and technology agnostic.

Finally, various approaches exist which address the modularization of transformations. They primarily focus on small transformations in a chain. One exception is the approach of Etien at al. [8] which modularizes transformations along specific tasks and purposes. This approach is largely complementary to GECO for two reasons: (a) they argue that larger transformations can be composed of small localized transformations. This correlates with fragment modularization (see Section 4.4). (b) they focus on modularization, but do not discuss the impact of metamodels. And (c) they do not define *concrete methods* for modularizing large transformations. With GECO we provide such methods.

7 Conclusion

We present an approach to support the construction and evolution of generators used in the context of MDE. Key contributions of GECO are megamodel patterns to guide the combination of fragments to complex generators, and a concept for the modularization of fragments to reduce the inner complexity of generators. We evaluated GECO with two case studies representing information systems and embedded control systems. The first integrated existing and new generators, and focused on feasibility of GECO. The second re-executed the development and evolution of a generator with GECO and compared it to the original generator project. In addition, we support fragment development and composition with a library of reusable modules supplemented by a DSL and generator [20].

Our future work will explore two primary avenues of investigation. First, we will extend our evaluation based on additional evolution steps for both case studies. Second, we will compare costs (time to realize alterations) for the second case study based on logged duration information. Third, we intend to evaluate code quality and performance of GECO generators, however, this requires larger models to be transformed. And finally, it would be interesting to compare generators for profile base approaches with aspect DSL generators.

Acknowledgement This work was supported by the DFG (German Research Foundation) under the priority program SPP 1593: Design For Future – Managed Software Evolution (grants HA 2038/4-1, RE 1674/7-1) and the Helmholtz Association of German Research Centers.

References

1. Allen, E.B.: Measuring graph abstractions of software: An information-theory approach. In: Symposium on Software Metrics, 2002. pp. 182–193. IEEE (2002)
2. Allen, E.B., Gottipati, S., Govindarajan, R.: Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. Software Quality Journal 15(2), 179–212 (2007)
3. Atkinson, C., Stoll, D., Bostan, P.: Orthographic software modeling: A practical approach to view-based development. In: Evaluation of Novel Approaches to Software Engineering, Communications in Computer and Information Science, vol. 69, pp. 206–219. Springer (2010)
4. Aßmann, U.: Invasive Software Composition. Springer (2003)
5. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009)
6. Bennett, J., Cooper, K., Dai, L.: Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach. Science of Computer Programming 75(8), 689–725 (2010)
7. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing, Limited (2013)
8. Etien, A., Muller, A., Legrand, T., Paige, R.: Localized model transformations for building large-scale transformations. Journal on Software and Systems Modeling 14(3), 1189–1213 (2015)
9. Fabro, M.D.D., Bézivin, J., Valduriez, P.: Weaving models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe (2006)
10. Fabry, J., Dinkelaker, T., Noyé, J., Tanter, E.: A taxonomy of domain-specific aspect languages. ACM Computer Survey 47(3), 40:1–40:44 (2015)
11. Favre, J.M.: Foundations of model (driven) (reverse) engineering – episode i: Story of the fidus papyrus and the solarus. In: Post-Proceedings of Dagstuhl seminar on model driven reverse engineering (2004)
12. Giacinto, D., Lehrig, S.: Towards integrating Java EE into ProtoCom. In: KPDAYS. pp. 69–78 (2013)
13. Goerigk, W. et al.: Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke (in German). In: SE'12. LNI, vol. 198, pp. 119–130. GI (2012)
14. Heinrich, R., Gärtner, S., Hesse, T.M., Ruhroth, T., Reussner, R., Schneider, K., Paech, B., Jürjens, J.: A platform for empirical research on information system evolution. In: Int. Conf. SEKE (2015)

15. International Electrotechnical Commission: IEC EN 61131-3, 2.0 edn. (2003)
16. ISO: International Standard ISO/IEC 9126. Information technology: Software product evaluation: Quality characteristics and guidelines for their use (1991)
17. Jouault, F.: Loosely coupled traceability for ATL. In: Proceedings of the European Conference on MDA: Workshop on Traceability. pp. 29–37 (2005)
18. Jung, R.: GECO: generator composition for aspect-oriented generators. In: Doctoral Symposium – MODELS'14 (2014)
19. Jung, R.: Replication package for GECO: A generator composition approach for aspect-oriented dsls (2016), `10.5281/zenodo.46552`
20. Jung, R.: Software package for the GECO replication package (2016), `10.5281/zenodo.47129`
21. Jung, R., Heinrich, R., Schmieders, E.: Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In: KPDAYS. pp. 99–108 (2013)
22. Jung, R., Heinrich, R., Schmieders, E., Strittmatter, M., Hasselbring, W.: A method for aspect-oriented meta-model evolution. In: 2. VAO Workshop. pp. 19:19–19:22. ACM (2014)
23. Kienzle, J., Abed, W.A., Klein, J.: Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development. pp. 87–98. ACM (2009)
24. Klein, J., Kienzle, J.: Reusable aspect models. In: 11th Workshop on Aspect-Oriented Modeling, MODELS'07 (2007)
25. Kramer, M.E., Kienzle, J.: Mapping aspect-oriented models to aspect-oriented code. In: MODELS'10. pp. 125–139. Springer (2011)
26. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering SE-2(4), 308–320 (1976)
27. Mehmood, A., Jawawi, D.N.: Aspect-oriented model-driven code generation: A systematic mapping study. Information and Software Technology 55, 395–411 (2013)
28. Mens, T., Gorp, P.V.: A taxonomy of model transformation. In: Proc. of the Int'l WS on Graph and Model Transformation. vol. 152, pp. 125–142. Elsevier (2006)
29. Morin, B., Klein, J., Barais, O., Jézéquel, J.M.: A generic weaver for supporting product lines. In: Proceedings of the 13th International Workshop on Early Aspects. pp. 11–18. EA '08, ACM (2008)
30. OMG: Meta Object Facility (MOF) Core Specification (2006)
31. Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.): The Common Component Modelling Example (CoCoME), LNCS, vol. 5153. Springer (2011)
32. Rentschler, A., Werle, D., Noorshams, Q., Happe, L., Reussner, R.: Designing information hiding modularity for model transformation languages. In: 13th International Conference on Modularity. pp. 217–228. ACM (2014)
33. Saada, H., Huchard, M., Nebut, C., Sahraoui, H.: Recovering model transformation traces using multi-objective optimization. In: Automated Software Engineering. pp. 688–693. IEEE (2013)
34. Shepperd, M.: A critique of cyclomatic complexity as a software metric. Software Engingeering Journal 3(2), 30–36 (1988)
35. Stahl, T., Völter, M.: Model-Driven Software Development – Technology, Engineering, Management. Wiley & Sons (2006)
36. Vanhooff, B., Baelen, S., Hovsepyan, A., Joosen, W., Berbers, Y.: Towards a transformation chain modeling language. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, LNCS, vol. 4017, pp. 39–48. Springer (2006)
37. Vanhooff, B., Baelen, S.V., Joosen, W., Berbers, E.: Traceability as input for model transformations. In: 3rd ECMDA Traceability Workshop. pp. 37–46. SINTEF (2007)