International Conference on Model Transformations (ICMT) 2010

Domain-Specific Composition of Model Deltas

Maider Azanza[1], Don Batory[2], Oscar Díaz[1], and Salvador Trujillo[3]

[1] University of the Basque Country, San Sebastián, Spain
{maider.azanza, oscar.diaz}@ehu.es
[2] University of Texas at Austin, Austin, Texas, USA
batory@cs.utexas.edu
[3] IKERLAN Research Centre, Mondragon, Spain
strujillo@ikerlan.es

Abstract. We present a general approach to the incremental development of model-based applications using endogenous transformations, i.e. transformations whose input and output models conform to the same metamodel. Our work describes these transformations as model deltas that, when composed, deliver a complete model. We establish a relationship between a metamodel and its corresponding delta metamodel, show how model deltas can be defined as model changes (additions), explain how deltas can be composed using domain-specific composition algorithms, and propose metamodel annotations to specify these algorithms. We present different case studies as proofs of concept.

1 Introduction

Model Driven Engineering (MDE) conceives software development as transformation chains where models are the artifacts to be transformed. Transformations which map abstract models to executables have received the most attention [30]. An *initialModel* of an application is conceived and then is mapped, level by level, until a platform-specific model is reached. The main challenges are mapping between levels, where each level has its own distinct metamodel, and balancing trade-offs between alternative transformation strategies, e.g. addressing non-functional requirements. These transformations, that have different input and output metamodels, are *exogenous*.

The creation of *initialModel*s has itself been the subject of research. Viewpoints have been used to address distinct perspectives of a target system [20]. In UML, class diagrams, state machine diagrams and activity diagrams capture different perspectives of the same application. Each viewpoint has its own distinct metamodel. But even viewpoints can be too coarse-grained to be units of development. *Object-oriented (OO)* programming faces similar problems when modularity is arranged by classes, rather than by concerns. Individual concerns crosscut distinct classes, complicating both development and maintenance [14].

Likewise, an *initialModel* integrates many distinct concerns of an application. Ease of maintenance encourages concerns to be both explicit and specified separately as model transformations, so that the *initialModel* itself is the result of a chain of *endogenous* transformations, i.e. transformations whose input

and output models conform to the *same* metamodel. A consequence is that an $initialModel$ is constructed incrementally by composing different endogenous transformations c_i, one per concern, starting from a base: $initialModel = c_3(c_2(c_1(baseModel)))$. This is the essense of *incremental development* [35].

Incremental development is not new. Collaborations have been proposed to support role-based designs in the incremental development of OO programs where each role contributes attributes and operations to achieve the role's purpose [31]. Classes are then "incrementally" extended by the roles they endorse. Incremental development is also at the heart of feature-oriented programming [5]. Here, the notion of feature displaces that of role as the design driving force. Products are then incrementally constructed by composing the features the product will exhibit. This paper inherits from these approaches. The primary difference stems from the artifacts being incrementally defined (i.e. initialModels), and the mechanism realizing the increments (i.e. endogenous transformations).

Current model transformation languages (e.g. ATL [15], ETL [19], RubyTL [11]) can specify any transformation. They were designed to handle arbitrarily complex mappings between different metamodels. Our experience and that of many others suggests that typical endogenous transformations are much simpler [3,23,24,32].

In this paper, we explore endogenous transformations characterized by:

- *Model Deltas.* Transformations are expressed as changes (additions) to models that when composed with a base model, deliver a complete model.
- *Domain-specificity.* Domain-specific composition algorithms are essential [3,24,28,36]. To this end, we annotate the domain metamodel to specify composition algorithms. We show that this light-weight approach offers a practical and powerful way to integrate domain-specific composition algorithms seamlessly into a development environment.

We implement our ideas using the *Epsilon Merging Language (EML)* [18] and present different case studies as proofs of concept[4].

2 Big Picture

Incremental development is widely known to handle design complexity and improve design understandability [35]. Although any artifact can be developed incrementally, the greatest benefits are observed when families of related artifacts are developed. This occurs in *Software Product Lines (SPL)* [10].

In an SPL, features can be seen as endogenous transformations [5]. A *feature* is an increment in program functionality that customers use to distinguish one program from another. SPL programs are constructed from a pre-planned set of features, which map (Java) programs to (Java) programs. A particular program in an SPL is produced by applying features to an initial program.

[4] Our composition engine and examples used in this paper are available for download at `http://www.onekin.org/andromeda`.

A product line in an MDE setting is a set of models. The *baseModel* corresponds to the above mentioned initial program and expresses the greatest common denominator of all SPL members. In many product lines, *baseModel* is simply the empty model $\emptyset$ [6].

Figure 1a shows a metamodel MM and its *cone of instances* (i.e. the set of all models that conform to MM). A subset of the cone is a set of models that belongs to a product line PL (e.g. $m_2 ... m_4$ belong to PL, m_1 is not a member, and all $m_1 ... m_4$ conform to MM). Note that the empty model $\emptyset$ and model m_A are outside the cone of MM, meaning that $\emptyset$ and m_A do not conform to MM.

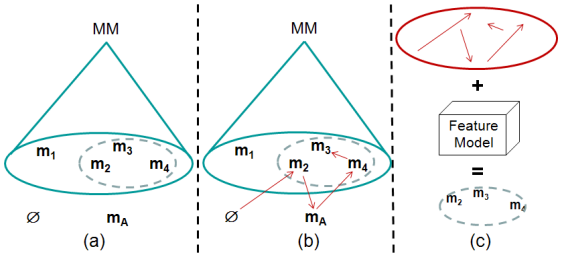

Fig. 1. Models, Endogenous Transformations and Product Lines

Figure 1b shows feature realizations (i.e. endogenous transformations) as *arrows*[5] that map one model to another. An important property about arrows is that a composition of two or more arrows is yet another arrow [6]. Further, observe that a composition of arrows may produce an intermediate model (m_A) that does not conform to MM. Only the members of PL (m_2, m_3 and m_4) are required to conform to MM, regardless of the steps needed to obtain them. The members of PL are produced by composing arrows starting from $\emptyset$ and this collection of arrows defines the features of PL.

A specification of all legal compositions of features in a product line is a *feature model* [16]. Every model in PL's product line has a derivation (i.e., composition of arrows starting from $\emptyset$) that is expressed by the PL's feature model. Figure 1c illustrates the SPL perspective: a collection of arrows (features) plus a feature model (that constrains how these arrows can be composed) yields the set of models of a product line.

The relationships among arrows described above are very general; they hold for *all* implementations. The key questions in mapping arrows to an implementation are: *(i)* how is an arrow specified?, *(ii)* if arrows are themselves models then, what is an arrow metamodel? *(iii)* how are arrows composed? and *(iv)* how can domain-specific arrow composition algorithms be defined? The next sections address these issues.

[5] The term comes from Category Theory. Arrows are maps between objects. In an MDE setting arrows can be implemented using endogenous or exogenous transformations depending on their function. In this work we are interested in arrows that build SPLs (i.e. endogenous transformations) that implement deltas. See [6] for a description of the links between MDE, SPLE and Category Theory.

3 A Questionnaire SPL and its Arrows

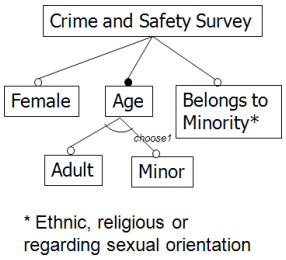

Fig. 2. Feature Model for the CSSPL

Questionnaires are a common research instrument in Social Sciences as a means to gather information. A set of questions is presented to respondents and a research hypothesis is validated against their answers. Although the Web is an invaluable resource for conducting surveys, social scientists generally lack the technical knowledge to make their questionnaires accessible online. To overcome this, abstractions are defined to liberate domain experts from the technicalities of Web surveys and are transformed into actual Web implementations[6].

Similar questionnaires that are targeted to different population profiles are needed on a regular basis. A one-size-fits-all questionnaire is inappropriate. Our example focusses on *Crime and Safety Surveys* that assess citizens' feelings on the safety of their enviroment. Figure 2 shows the feature model for the *Crime and Safety Survey Questionnaire SPL (CSSPL)*[7]. Its features define how questionnaires vary in this domain. Specifically, *(i)* if the respondent is a `female`, she is given a *Sexual Victimization Block* apart from the *Base Questionnaire*, *(ii)* if he or she `belongs to a minority`, a *Hate Crime Block* is added and *(iii)* regarding `age`, `adults` are given the *Consumer Fraud Block* while `minors` receive the *School Violence Block*. Now questionnaire creation is not just a "single shot". Rather, a questionnaire is characterized in terms of features that customizes it for a target audience.

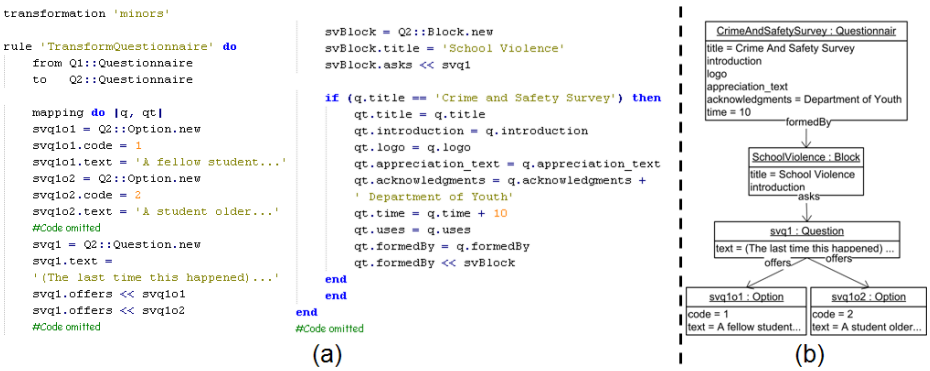

Fig. 3. Feature Implementation Options

[6] Examples of applications that cater to this need are ESurveyPro (`http://www.esurveyspro.com`) and Lime Service (`http://www.limeservice.org`).

[7] This *Questionnaire* family is inspired in the European Crime and Safety Survey (`http://www.europeansafetyobservatory.eu/euics_fiq.htm`).

There are two basic ways to implement a feature (i.e. an arrow that increments a model with certain functionality). One is to use a general-purpose transformation language. Figure 3a shows how the *Minor* feature is expressed in RubyTL. That is, how the *Department of Youth* should be added to the `acknowledgments` of the *Base Questionnaire*, how the estimated completion `time` is increased in 10 minutes, and the *School Violence Block* that should be included when the questionnaire is directed to minors. Another way is simply to create a *model delta* that defines the additions the *Minor* feature makes to a model (see Figure 3b). The latter brings the following advantages: *(i)* permits the checking of questionnaire constraints (e.g. `option codes` have to follow a certain format), *(ii)* makes features easier to understand and analyze [36] and *(iii)* separates *what* the feature adds from *how* it is added, thus making the *how* reusable for all questionnaire arrows. This advantages are revised in Section 7. Note that this definition of endogenous transformations is monotonic, features are not allowed to delete previously existing model elements.

4 Arrow Metamodels

Let M and MM be a model and its metamodel, respectively. Further, let AM and AMM stand for an arrow model and its arrow metamodel, respectively. This section presents a definition of AMM and its relationship with MM.

Figure 4 is a simplified *Questionnaire Metamodel* MM. Arrows (which to us represent fragments/additions to MM models) do not satisfy all constraints of MM. In the `Question` metaclass, for example, each question must have at least two options. An arrow can contribute just one or no option at all. This is the case for the *Minor* feature (see Figure 5), where question `atq2` has but one option. Once this feature is composed with the base, the result has four options, which is conformant with the questionnaire metamodel. Hence, an AMM can be created by eliminating constraints from MM.

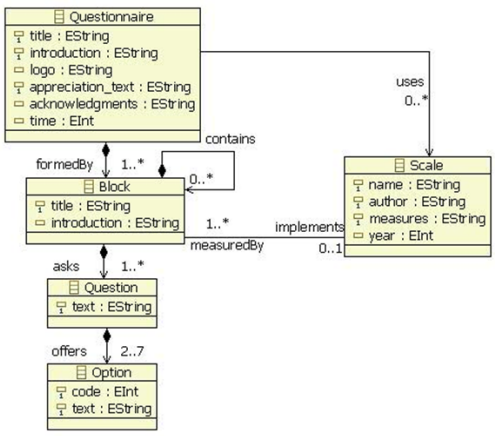

Fig. 4. Questionnaire Metamodel

However, not every constraint in the metamodel should be removed. Consider the `Question` metaclass which requires every question to have at most seven options. If any arrow adds more than seven options, every questionnaire created using that arrow will not conform

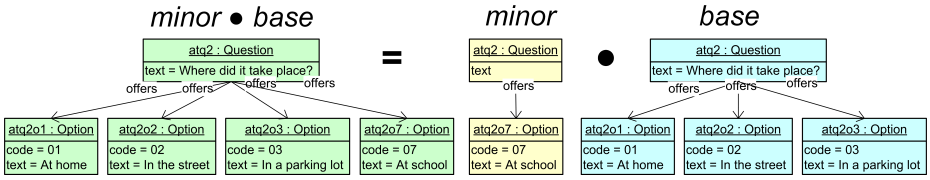

Fig. 5. Constraint Violation by the *Minor* Feature

to the metamodel. Thus, this upper-bound constraint should be fulfilled by every arrow[8].

Two types of constraints are encountered when creating arrow metamodels:

- *Arrow Constraints:* Constraints that can be validated for every arrow (e.g. upper-bounds, `option codes` have to follow a certain format, etc.). These constraints are kept in the AMM.
- *Deferred Constraints:* Constraints that can only be evaluated after *all* arrows are composed, i.e. when the entire product model has been assembled (e.g. lower-bounds)[9]. The AMM is obtained by removing these contraints from the metamodel.

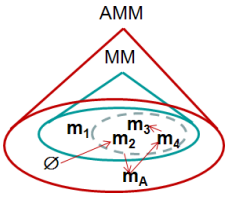

Fig. 6. AMM Cone

Domain engineers should decide to which group each constraint belongs. An AMM can also be automatically generated from the product metamodel simply by removing all the metamodel's constraints except upper-bounds. However, the first option is preferrable as it detects more inconsistencies earlier in the process (i.e. at arrow creation time). We derived the AMM for the Questionnaire MM by removing eight lower-bound constraints.

Figure 6 shows the relationship between the cones of MM and AMM. Each model m_i in MM is equivalent to an arrow $\emptyset \rightarrow m_i$ in AMM. This means that the cone of AMM not only includes every model in the cone of MM, but other models and arrows as well (e.g., models $\emptyset$ and m_A, and arrow $m_A \rightarrow m_4$ and composite arrow $\emptyset \rightarrow m_A$). The close relationship between AMM and MM means that arrows are defined using the same concepts as models and, as they belong to an SPL, they are designed with composition in mind, the topic of the next section.

[8] This holds for composed arrows as well. If a composed arrow adds more than seven options, its use will also not conform to the the `Question` metaclass.

[9] This has a database counterpart. The database administrator defines a database schema with constraints. Additionally, the administrator can declare a constraint to be validated as soon as a database update happens (immediate mode) or wait till the end of the transaction, once all updates were conducted (deferred mode). The deferred mode allows for some constraints to be violated during a transaction execution, as long as the constraints hold at the end of the transaction.

5 Arrow Composition

Model composition has been defined as the operation M_{AB} = Compose (M_A, M_B, C_{AB}) that takes two models M_A, M_B and a correspondence model C_{AB} between them as input, and combines their elements into a new output model M_{AB} [7]. Arrow composition is a special case, where both M_A and M_B conform to the *same* metamodel. Further, the correspondence model C_{AB} is implicit as objects with the same name (or rather, identifier) in models M_A and M_B are, in fact, the same object. Arrow composition is performed by pairing objects of different fragments with the same name (identifier) and composing them.

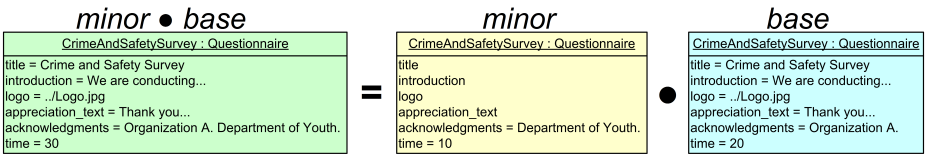

Fig. 7. Composition of Minor and Base Features

There are two distinct cases in object composition:

- *Generic:* This is the default, metamodel independent, composition [2,5]. The composition of a pair of objects equals the composition of their corresponding attribute pairs. The composition of a pair of scalar-valued attributes is this: if both have different non-null values, composition fails and an error is raised. Otherwise, the non-null value is the result. In the case of set-valued attributes (including references), their union is the result. Figure 7 presents different examples of this composition: `title`, `introduction`, `logo` and `appreciation_text` attributes.
- *Domain-Specific:* In a majority of cases, generic composition is sufficient. The remaining cases require a domain-specific composition method. Consider the `acknowledgments` and `time` attributes in the `Questionnaire` metaclass. An organization can fund all the study or only part of it. Figure 7 shows how the *base* study is funded by Organization A and the part regarding *minors* is funded by the Department of Youth. If objects were composed generically, an error would be raised as both objects have a different value in the `acknowledgments` attribute. However, the convention for questionnaires is to concatenate both acknowledgments as the result.
  The `time` attribute is another example. It indicates the estimated time needed to complete the questionnaire. The *base* feature takes 20 minutes and the *minor* feature needs 10 more. The expected behavior would add both values to the result, not to raise an error.

A practical tool should allow domain engineers to annotate any class or attribute to indicate that a customized composition algorithm, rather than the default

algorithm, is to be used. Additionally, since some algorithms are likely to be used in different domains (e.g. string concatenation), a set of keywords are provided to denote those recurrent algorithms as annotations on the metamodel, hereafter referred to as *built-in algorithms*. These annotations include: `@concat` (i.e. given two values V1 and V2, the composition delivers V1V2), `@slash_concat` (i.e. the composition delivers V1/V2), `@sum` (i.e. the composition delivers the addition of V1 and V2), `@min` (minimum value), and `@max` (maximum). Engineers can turn a domain-specific composition algorithm into a built-in algorithm by naming and including it in a composition-algorithm library.

Figure 8 depicts the annotated *Questionnaire* metamodel. Note that the `acknowledgments` attribute in *Questionnaire* and the `introduction` attribute in *Block* are annotated with `@concat`. This indicates that their values are to be composed by concatenation. Moreover, the `time` attribute in *Questionnaire* is annotated with `@sum`, meaning that composed contents should be added.

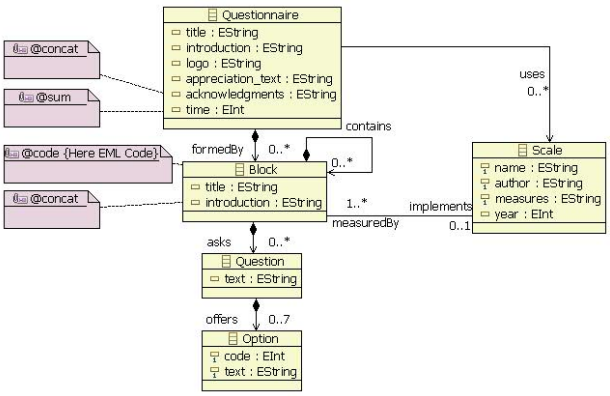

Fig. 8. Annotated Questionnaire Metamodel

When no built-in algorithm is sufficient, engineers must resort to describing domain-specific composition algorithms procedurally (hereafter called *ad-hoc algorithms*). This is the only case that requires knowledge of a transformation language, which in our case is EML. In the *Questionnaire* domain, `blocks` can be defined using a scale. When composing blocks with different scales, *Questionnaire* semantics dictate not to merge blocks but to keep them as separate sub-blocks[10]. We specify ad-hoc algorithms as a piece of EML code, and is an annotation that is attached to the *block* class[11] (see Figure 8).

[10] Note that this domain specific composition is not limited to individual attributes as in the above examples; it applies to complete *Block* objects.

[11] This option is debatable. Another possibility is to embed the ad-hoc algorithm directly into the generated code using a "protected region", i.e. a piece of code that is not overridden when the code is newly generated. We prefer to have all composition algorithms specified in a single place: the metamodel. In our opinion, this facilitates understandability and maintainability.

5.1 Implementation

Given an annotated metamodel MM, our prototype automatically generates *Epsilon Merging Language (EML)* rules for composing arrows. EML is a rule-based language for merging models of diverse metamodels and technologies [18].

The `Object` metaclass defines the core metamodel of our implementaion of *ANDROMEDA (ANnotation-DRiven Oid-based coMposEr for moDel Arrows)* (see Figure 9). `Object` instances are singled out by an explicit identifier that is used for object matching. `Object` holds the generic methods for `match`, `merge`, and `copy` methods. *Match* compares pairs objects and decides if they match. *Merge* merges two objects that match and *copy* copies objects for which no match has been found to the result.

This generic behavior can be specialized to cater for the composition peculiarities of the domain at hand. To this end, `Object` specializations are created for each domain metaclass (see Figure 9). Note how `Questionnaire` and `Block` override the generic merge method to address their own specific composition semantics.

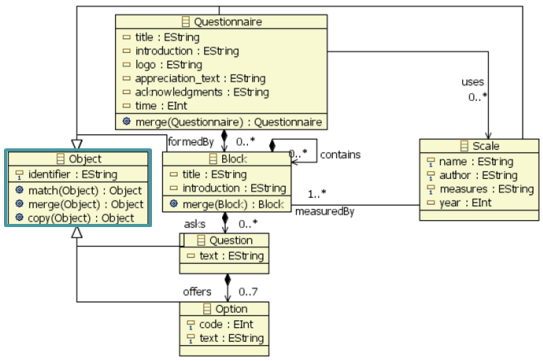

Fig. 9. Extended Questionnaire Metamodel

Generic behavior. This behavior is provided by ANDROMEDA. The `match` method is realized through an *Epsilon Comparison Language (ECL)* rule (see Figure 10a) [17]. This rule indicates that two `Objects` match if they are instances of the same class and have the same identifier. The `merge` method is supported through an EML rule (see Figure 10b). The rules applies to all Object instances. We also provide methods that define default merge behavior for attributes and references.

Domain-specific behavior. Domain-specific EML rules that extend the generic EML rules (rule extension is the counterpart of method inheritance) are **generated**. Metamodel annotations indicate that generic merging needs to be substituted by domain-specific merging. Three situations arise:

- *Generic Algorithm* (e.g. *Question* metaclass). See Figure 10c for an example. This rule extends the generic rule `MergeObject` for the *Question* metaclass by just reusing the default methods.
- *Built-in Algorithm* (e.g. *Questionnaire* metaclass). The annotations in the metaclass the metamodel (see Figure 8), the `acknowledgments` attribute is annotated with `@concat` and the `time` attribute is annotated with `@sum`. These annotations lead to the merge rule in Figure 10d where the generic

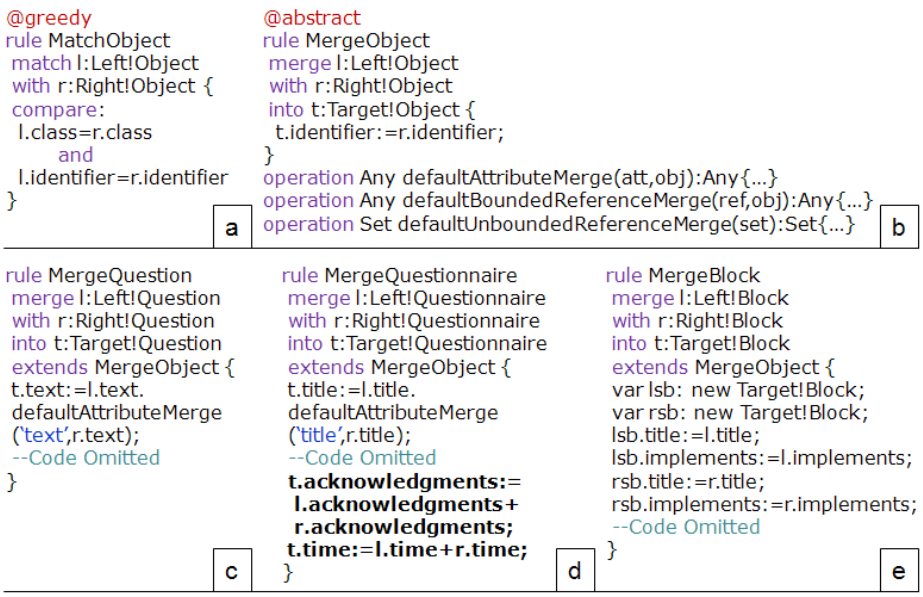

Fig. 10. Implementing composition through EML and ECL rules.

merge of `acknowledgments` and `time` attributes is overridden by string concatenation and integer addition respectively.

– *Ad-hoc Algorithm* (e.g. *Block* class). The approach is similar to the built-in algorithm, but now the code embedded in the annotation is directly copied into the EML rule (see Figure 10e).

6 Case Studies

Among the motivations for connecting arrows and product lines in Section 2 is to underscore the generality of their relationships. We believe our *CSS Questionnaire SPL (CSSPL)* is typical of a basic MDE product line, and we have explained how our tools handle it. But there are examples from other technical spaces [8] for which these same relationships can be demonstrated to hold.

The largest examples of feature-based compositions come from *Feature Oriented Programming (FOP)*, i.e. from the EBNF technical space [8]. SPLs in FOP are families of Java-based programs. *AHEAD Tool Suite (ATS)* is a set of tools to support FOP [5]. ATS was refactored into the *ATS Product Line (APL)*, i.e., ATS bootstraps itself by composing features of APL. Over time, ATS has grown to 24 different tools comprising over 200K LOC Java. In our case study, we focus only on model representations of the code that corresponds to features[12].

[12] Along with APL, we also created models for the *Graph Product-Line (GPL)*, a family of classical graph applications [21]. The results of GPL are not much different

Features in APL are written in Jak, a superset of Java that supports features and metaprogramming [5]. We defined a bridge that translates an APL feature written in Jak into a model fragment that conforms to the Jak Arrow Metamodel and the other way around. By composing model fragments, we can produce a model of the ATS product line.

Interestingly, no constraint had to be removed from the Jak metamodel to produce its arrow metamodel: all constraints defined in the Jak metamodel where applicable to arrow models. Next, a composition algorithm was chosen for each element in the Jak metamodel.

Table 1 lists statistics about the *Jak Metamodel*, with statistics from the *Questionnaire metamodel (Qst)* for comparison purposes. The number of classes, attributes and references of each metamodel and the distribution of their composition types are listed. Note that Jak uses a large number of domain-specific composition algorithms. While this is not unexpected, what is interesting is that several of these algorithms worked on objects and not attributes.

MM DATA	Jak	Qst
Classes	12	5
Attributes	18	15
References	25	7
Removed Constraints	0	8
Generic Comp.	37	23
D.S. Comp.	18	4

Table 1. Jak and Questionnaire Metamodels

Table 2 compares the average size of CSSPL and APL features with their size in objects (a.k.a *Objs*) and references (a.k.a *Refs*). It also presents the average size for particular products we composed. Note that APL model arrows are on average four times larger than the arrows of CSSPL, and APL products are over fifty times larger than CSSPL products.

Again, one of the advantages of the Jak case studies was to demonstrate our composition principles hold across different technical spaces. Another advantage was that we could verify the correctness of our compositions. Model arrows were composed to yield a certain product and then transformed into code. The same product was obtained by directly composing its code features and both results were compared using source equivalence to test for equality[13].

	Objs	Refs
CSSPL Features	**22**	21
APL Features	109	332
CSSPL Products	63	62
APL Products	3035	9394

Table 2. CSSPL and APL Statistics

7 Perspective and Related Work

7.1 Perspective

We conceive model construction as a gradual composition of model arrows. Each arrow realizes an increment in application functionality (i.e. a feature), and the

than what we report for APL and CSSPL. The GPL case study is available at `http://www.onekin.org/andromeda`.

[13] Source equivalence is syntactic equivalence with two relaxations: it allows permutations of members when member ordering is not significant and it allows white space to differ when white space is unimportant.

functionality of the final application is the added functionality of each of its arrows. Therefore, the characterization of our work is twofold. First, increments in functionality are embodied as deltas that are expressed in the same language as the final model. The benefits include:

- Certain domain constraints can be checked at arrow building time, allowing earlier error detection. It paves the way to safe composition, the assurance that all products of the SPL conform to the domain metamodel.
- It separates what a feature adds from how it is added, thus making the composition algorithms reusable for all features that conform to the same metamodel.
- Declarativeness. Model arrows are easier to read and write than their transformation counterpart. Figure 3a shows the same *Minor* feature but now realized using a general-purpose transformation language, RubyTL [11]. Even to someone accustomed to reading transformation rules, it takes some time to grasp information that the transformation adds to the base model. When handling SPLs with hundreds of features, scalability is a main issue and declarativeness a major enabler.
- The Arrow Metamodel can be derived from the domain metamodel (e.g. *Questionnaire*) by removing constraints.

Second, our work also uses model superimposition as the default approach to model composition. Additionally, domain-specific composition semantics are captured through annotations in the domain metamodel. This improves the declaractiveness of how the composition algorithms are captured. The benefits include:

- *Automatization*. The composition algorithm can be automatically generated. As a proof of concept, we showed the implementation that generates EML rules.
- *Understandability*: Anchoring composition annotations on the metamodel, permits designers to focus on the *what* rather than on *how* the composition is achieved.
- *Maintenance*: Additional composition algorithms can be added or removed with minimal effort. This could be of interest in the context of metamodel evolution where new metaclasses/attributes can be added that require customized composition algorithms [33].

Our work also raises research questions that are subject of future work. Generalizing our arrow definition to allow deletions is a case in point. We want to study in which cases is that deletion necessary and what its drawbacks are. Moreover, in an MDE setting endogenous transformations define only half of the picture. We are interested in transformations that map an arrow that is an instance of an arrow metamodel to an arrow that is an instance of another [6]. Another topic deals with safe composition: once all arrows are composed, the resulting model should fulfill all the constraints of the metamodel. We want to guarantee that all legal feature compositions yield models that satisfy all constraints of the metamodel, *but without enumeration*. Existing work suggests a direction in which to proceed [12,22,29].

7.2 Related Work

Implementation Approaches. Two main trends in implementing endogenous transformations can be identified: transformation languages vs. model deltas. Examples of the former include C-SAW [4], MATA [34] and VML* [36]. A transformation language is more versatile and it performs analyses that are presently outside our work. For example, MATA comes with support to automatically detect interactions between arrows. Since arrows in MATA are graph rules, the technique of critical pair analysis can be used to detect dependencies and conflicts between rules[14]. This versatility, however, comes at a cost: *(i)* it requires developers to be familiar with a graph transformation language and *(ii)* being defined in a different language, it reduces checking with respect to the domain metamodel that can be performed on the transformations. To ease the developers burden, MATA defines transformations between UML and the underlying graph rules. Nevertheless, these transformations must be defined for every metamodel at hand.

Traditionally SPL arrows are defined as model deltas — a set of additions — that are superimposed on existing models. Recently, several researchers have followed this trend, particularly using aspects to implement them [3,23,24,32]. To the best of our knowledge, none of these approaches defines a mechanism to check the conformance of arrows to their arrow metamodels. Interestingly, SmartAdapters [24] defines pointcuts as model snippets that conform to a metamodel that is obtained by eliminating all constraints from the domain metamodel, in a similar fashion to the way we define arrow metamodels [26]. However, no mention is made about an advice metamodel.

Along with aspects, collaborations are another candidate paradigm for realizing model deltas. Collaborations are monotonic extensions of models that encode role-based designs and are composed by superimposition [31]; collaborations are a centerpiece in recent programming languages (e.g. Scala [25]) and prior work on feature-based program development [5]. In contrast, aspects may offer a more general approach to express model deltas, where pointcuts identify one or more targets for rewriting (a.k.a. advising). However, the generality of aspects is by no means free: it comes at a cost of increased complexity in specifying, maintaining, and understanding concerns [1].

Domain-Specificity. There are a number of model delta composition tools, some using *Aspect Oriented Modeling (AOM)*, that are now available. XWeave [32] and Kompose [13] support only generic composition, although the latter has some limited support for domain specificities in the form of pre-merge and post-merge directives. Other approaches, e.g. SmartAdapters [24], VML* [36] and FeatureHouse [3], provide support for domain-specific composition. The first two require engineers to explicitly indicate how arrows are to be composed, while we strive to promote reuse by automatically generating as much as possible from the metamodel by leveraging metamodel annotations. FeatureHouse also accounts for domain specificities. However, it is restricted to tree composition, and domain-specific composition is limited to tree leaves. In contrast, our

[14] It may be possible to define a corresponding analysis on model fragments.

approach considers graphs rather than trees and composition can be simultaneously specified at different points within models.

Model Differences. Model arrows are closely related to the idea of model model differences [9,27]. The main distinction stems from their purpose, the former implement a feature in an SPL and are normally built separately while the latter are calculated as the difference between two models that conform to the same metamodel.

8 Conclusions

MDE conceives software development as transformation chains where models are the artifacts to be transformed. We presented an approach to the incremental development of *initialModel*s in an SPL setting. Models are created incrementally by progressively applying endogenous transformations that add increments in functionality. We depicted these transformations as arrows and implemented them as model deltas. We explained how arrows realize features in SPLs, how arrows conform to arrow metamodels, and how arrow metamodels are derivable from domain metamodels. The focus of our work stressed the need for domain-specific composition algorithms, and how they could be added to metamodels in a practical way. Further, our case studies illustrated the technical-space independence of our approach. We believe our light-weight approach offers a practical and powerful way to integrate domain-specific composition algorithms seamlessly into an MDE development environment.

Acknowledgments. We gratefully thank L. Vozmediano for her help with the questionnaires domain and J. De Sosa, M. Kuhlemann, R. Lopez-Herrejon, G. Puente and J. Saraiva for fruitful discussions during the preparation of this draft. This work was co-supported by the Spanish Ministry of Education, and the European Social Fund under contract MODELINE, TIN2008-06507-C02-01 and TIN2008-06507-C02-02. Batory's work was supported by NSF's Science of Design Project CCF-0724979.

References

1. S. Apel et al. Aspectual Feature Modules. *IEEE TSE*, 34(2), 2008.
2. S. Apel et al. FeatureHouse: Language-Independent, Automated Software Composition. In *ICSE*, 2009.
3. S. Apel et al. Model Superimposition in Software Product Lines. In *ICMT*, 2009.
4. K. Balasubramanian et al. Weaving Deployment Aspects into Domain-Specific Models. *IJSEKE*, 16(3), 2006.
5. D. Batory et al. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
6. D. Batory et al. The Objects and Arrows of Computational Design. In *MoDELS*, 2008.
7. J. Bézivin et al. A Canonical Scheme for Model Composition. In *ECMDA-FA*, 2006.
8. J. Bézivin and I. Kurtev. Model-based Technology Integration with the Technical Space Concept. In *MIS*, 2005.

9. A. Cicchetti et al. A Metamodel Independent Approach to Difference Representation. *JOT*, 2007.
10. P. Clements and L. M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.
11. J. S. Cuadrado et al. RubyTL: A Practical, Extensible Transformation Language. In *ECMDA-FA*, 2006.
12. K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE*, 2006.
13. F. Fleurey et al. A generic approach for automatic model composition. In *MoDELS Workshops*, 2007.
14. G. Gottlob et al. Extending Object-Oriented Systems with Roles. *ACM TOIS*, 14(3), 1996.
15. F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*, 2005.
16. K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
17. D. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *ECMDA-FA*, 2009.
18. D. Kolovos et al. Merging Models with the Epsilon Merging Language (EML). In *MoDELS*, 2006.
19. D. Kolovos et al. The Epsilon Transformation Language. In *ICMT*, 2008.
20. V. Kulkarni and S. Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5), 2003.
21. R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*, 2001.
22. R. Lopez-Herrejon et al. Using incremental consistency management for conformance checking in feature-oriented model-driven engineering. In *VAMOS*, 2010.
23. B. Morin et al. A Generic Weaver for Supporting Product Lines. In *EA*, 2008.
24. B. Morin et al. Weaving Aspect Configurations for Managing System Variability. In *VAMOS*, 2008.
25. M. Odersky et al. An Overview of the Scala Programming Language. Technical report, EPFL, 2004.
26. R. Ramos et al. Matching Model-Snippets. In *MODELS*, 2007.
27. J. E. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *TOOLS*, 2008.
28. Ina Schaefer. Variability modelling for model-driven development of software product lines. In *VAMOS*, 2010.
29. S. Thaker et al. Safe Composition of Product Lines. In *GPCE*, 2007.
30. A. Vallecillo, J. Gray, and A. Pierantonio, editors. *ICMT*, 2008.
31. M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA*, 1996.
32. M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, 2007.
33. Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *ECOOP*, 2007.
34. J. Whittle and P. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In *MODELS Workshops*, 2007.
35. N. Wirth. Program Development by Stepwise Refinement. *Commununications of the ACM*, 14(4), 1971.
36. S. Zschaler et al. VML* - A Family of Languages for Variability Management in Software Product Lines. In *SLE*, 2009.