Domain-Specific Metamodelling Languages for Software Language Engineering

Steffen Zschaler[1], Dimitrios S. Kolovos[2], Nikolaos Drivalos[2], Richard F. Paige[2], and Awais Rashid[1]

[1] Computing Department, Lancaster University, UK.
{zschaler,awais}@comp.lancs.ac.uk
[2] Department of Computer Science, University of York, UK.
{dkolovos,nikos,paige}@cs.york.ac.uk

Abstract. Domain-specific languages are constructed to provide modelling capabilities tailored to a specific domain. Sometimes, languages are developed many times, typically to support application in a new context. In doing so, recurring patterns and commonalities as well as variations *across* the evolving set of languages can be identified. This paper introduces the concept of a *domain-specific metamodelling language*, which codifies such commonalities and provides concepts and logic for expressing the variations. The challenges and difficulties of using domain-specific metamodelling languages are identified. We illustrate the concept with examples from different domains.

1 Introduction

Domain-specific languages (DSLs) for software engineering are custom- and purpose-built languages that target a specific domain. These languages are often smaller than general-purpose languages (GPLs), providing fewer and more focused language constructs and, ideally, a simpler and more rigorous semantics. The intent with DSLs is to provide a concise, tailored language that is easier for engineers and domain experts to learn, understand and apply for a specific class of problem.

Engineers build DSLs when they detect recurring patterns in the programs or models that they are constructing using a GPL (e.g., Java or UML). The concepts and logic in these patterns are then encoded in a new DSL that promotes these patterns to first-class language constructs. From a technical perspective, the DSL is typically implemented atop a standardised *metamodelling infrastructure*, such as EMF/Ecore [20], which provides reflective mechanisms to encode its abstract syntax (i.e. the language constructs, their properties and relationships). A number of different approaches can be used to define DSL semantics [4], as well as their graphical or textual concrete syntaxes.

With the growing adoption of DSLs in Model-Driven Engineering (MDE), the phenomenon of *families of DSLs* has started to appear. A family of DSLs is a set of languages, each of them targeting a concrete instance of a common problem domain. We have encountered this phenomenon in our previous work

on diverse domains. For example, in [18] we proposed constructing a new DSL for capturing traceability between each set of metamodels of interest, and in [48] a new DSL is implemented for every Software Product Line of interest, to enable engineers to precisely capture the mappings between features and model elements. Other authors have encountered similar phenomena too. For example in [5] a new DSL needs to be specified for each metamodel of interest to enable users and tools to capture differences of models that conform to this metamodel.

The contributions of this paper are threefold:

1. We identify the key challenges involved in developing families of DSLs. This consolidates the experience obtained from the existing work on families of DSLs mentioned above.
2. We introduce the idea of *domain-specific metamodelling languages (DSM2Ls)* and demonstrate a translational approach for automating the construction of the syntax, semantics and tool-support of families of DSLs.
3. We demonstrate the approach by showing how a number of real-world families of DSLs can benefit (and have done so in the past) from its application.

The rest of the paper is structured as follows. In Sect. 2 we identify challenges in developing families of languages. We then introduce, in Sect. 3, the concept of domain-specific metamodelling languages, and show how employing DSM2Ls makes it possible to fully generate the complete language family infrastructure, including the abstract concrete syntax, editors, and other tools. Section 4 presents examples of applying the approach. Finally, we consider related work before concluding.

2 Challenges for Developing Families of DSLs

In principle, domain-specific languages are more useful the more closely they are aligned with their target domain. However, despite the advance in tool support for defining the abstract and concrete syntaxes of DSLs, developing DSLs that are closely aligned with the problem domain still requires a significant amount of effort and expertise. In some cases, we have found ourselves developing very similar DSLs to target similar instances of a problem within marginally different domains. In particular, we have experience with two such families of languages: a family of languages for expressing traceability links between model artefacts [18], and a family of languages for variability management in product-line development [48]. In both cases, for each language of the family we needed to specify the abstract and concrete syntaxes as well as implement supporting infrastructure, such as editors, analysers, or code generators.

Having developed several DSLs of each family manually, we realized that all DSLs within a family shared a common core and demonstrated specific types of variations that propagated in a predictable manner to their syntax, semantics and tool-support. In the example of languages for tracing, there is a notion of a `TraceLink`, which needs to be specialised for each pair of language concepts for which trace links should be established. Specific types of trace links are

associated with specific types of constraints. All other language concepts are essentially the same for all tracing languages.

For all of these cases, there is an alternative solution, which is to develop one generic language rather than a whole family of languages, each customised for its specific context of use. For some cases, this may be the better solution. However, for other languages we have found a number of reasons that can make developing customised languages more appropriate. Section 4 discusses examples of language families and also discusses the specific reasons for each language. Typically, these fall into the following categories:

- *Context-specific constraints.* Customising languages for a specific context of use enables the definition of context-specific constraints. For the traceability example, each context-specific type of trace link can be associated with context-specific constraints on the number and type of elements that can be connected by such a link.
- *Context-specific terminology.* A generic language will invariably use generic terms. In certain situations, it is preferable to use terms customised for the specific context of use. For example, in the product-line languages, we customised the actions available for manipulating target models so that they would be immediately familiar to product-line engineers accustomed to using these target languages for modelling their product line rather than using generic terminology that would be immediately understandable only to language designers.
- *Enhancing existing languages with new functionality.* Especially some technical concerns, such as modularisation, need to be supported by any DSL, but are costly to develop. Hence, it would be beneficial to be able to reuse one realisation of such a concern for different languages. This requires the language concepts supporting the concern to be re-implemented in different contexts, one for each DSL for which to support the technical concern.

If, for a given domain, for one or more of the above reasons we decide to develop customised DSLs for different contexts of use, efficiency of DSL development becomes an issue. Repeatedly redeveloping the language from scratch is not only inefficient, but also error prone. To increase efficiency and reduce error-proneness, we have to address the following challenges:

1. *Systematic reuse of DSL constructs and infrastructure elements.* We require a reuse mechanism that goes beyond simple "copy&paste" and covers all aspects of a DSL's support infrastructure: metamodels, grammars, parsers, interpreters, compilers, editors, etc.
2. *Systematic support for specifying variability between the DSLs.* At the same time, we need to be able to easily specify differences between the DSLs. In particular, we should not need to make a number of co-ordinated changes throughout different aspects of the DSL's support infrastructure just to modify one feature of the language. We will need to support variability in the following forms:

(a) *Addition/Removal of concepts.* This implies changes to all aspects of the language infrastructure, but in particular to grammars, metamodels, editors, parsers, code generators, and verification tools.
(b) *Integration with different additional languages.* This requires changes to metamodels, type checkers, and evaluation strategies (e.g., compilers or interpreters).

One way to address these issues would be, in the spirit of MDE, to specify languages within each family at an even higher level of abstraction than that provided by general-purpose metamodelling languages such as MOF/ECore where their commonalities could be ignored and only their variations would need to be specified. This higher-level specification could then be used to automatically generate all the artefacts of the DSL with reduced effort and enhanced consistency. The next section discusses this approach in more detail.

3 Domain-Specific Metamodelling Languages

To raise the level of abstraction in the construction of families of languages, we propose the use of Domain Specific Metamodelling Languages (DSM2Ls). We present the following working definition for the concepts of *family of languages* and use it to define the concept of *DSM2L*:

Definition 1: A *family of languages* is a set of languages that demonstrate a common core of constructs and a well-defined set of types of variation.

Definition 2: A *Domain-Specific Metamodelling Language (DSM2L)* is a language used to define syntax, semantics, and tooling aspects[3] of languages belonging to a specific *family of languages*.

In the same sense that a tailored DSL is more concise and efficient for capturing models of a specific domain than a GPL (such as UML), a DSM2L is more concise and efficient than a general-purpose metamodelling language, such as MOF, for specifying the syntax, semantics and tooling aspects of languages of a *particular type*. Unlike general-purpose metamodelling languages, a DSM2L is not capable of capturing every metamodel—instead, it is tailored for capturing only specific types of metamodels of interest by providing first-class support for the allowed types of variation within the family.

Expressing a family of DSLs using a custom DSM2L enables the definition of generators that automatically generate editors, constraints, transformations and other supporting infrastructure by exploiting knowledge of the family of languages for which the DSM2L has been defined (see Fig. 1). For example, by using the Traceability Metamodelling Language (TML) [18], which is a DSM2L language for expressing traceability DSLs, we are able to automatically generate a set of correctness constraints for each `TraceLink`. One example of such a

[3] Here, we include parsers, editors, code generators, etc.

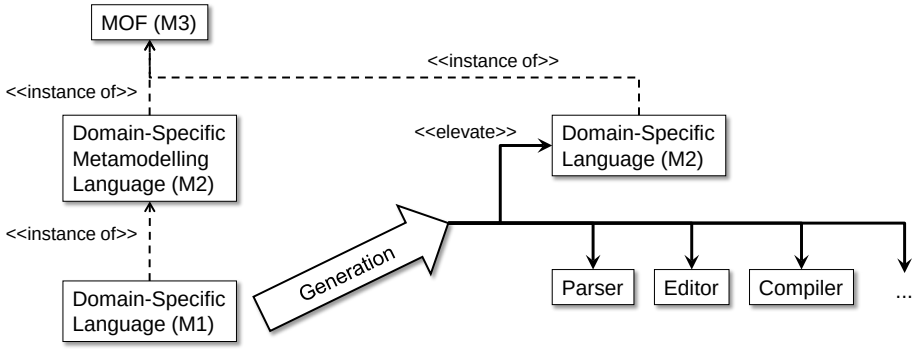

Fig. 1. Generating language support infrastructures from a DSM2L

constraint is the so called `ForAll` constraint, which guarantees that at least one traceability link of a particular kind exists for all instances of a model-element type. In general, the generated constraints in TML guarantee the semantic integrity of `TraceLink`. Further examples of DSM2Ls can be found in Sect. 4.

Implementing DSM2Ls in the context of metamodelling architectures—such as, for example, the OMG metamodelling architecture—faces the problem that these architectures typically only allow for one metamodelling language. The generative nature of the DSM2L approach can be used here as well. For every DSM2L, we can define a generator that takes an instance of the DSM2L and *elevates* it to a proper metamodel by expanding the domain-specific modelling constructs and replacing them by equivalent (if more verbose) representation in the standard metamodelling language of our choice (see Fig. 1).

The proposed approach addresses the challenges identified in Sect. 2 by automating the construction process of the DSL metamodel, as well as the respective constraints and required tooling:

1. It supports systematic reuse of DSL constructs and infrastructure elements by allowing language engineers to specify a core of reusable elements that will be used by the generators to create a language-specific infrastructure.
2. At the same time, it provides systematic support for specifying variability between the DSLs by allowing language designers to specify a well-defined set of variation points that can be then supported consistently in the different parts of the infrastructure (e.g. constraints, transformations). The DSM2L provides explicit concepts for each of these variation points and the generators encode the implementation of each variation.

4 Examples

In this section, we discuss three different families of languages and show how DSM2Ls have been used to implement these language famuilies.

4.1 TML

The Traceability Metamodelling Language is a language whose purpose is to enable the construction and maintenance of traceability metamodels as well as their accompanying correctness constraints.

Two main approaches for storing and maintaining traceability information can be found in the literature [32]. The first approach is to embed traceability information in the models it refers to, while in the second approach distinct traceability models are used to capture the traceability information. As argued in [17, 32], the second approach is to be preferred, since it does not *pollute* the models involved with information of secondary importance.

A trace metamodel can be either of a generic or of a case-specific nature. In the case of a generic trace metamodel, a set of traceability concepts is identified and it is represented in the metamodel as meta-classes. An assumption related to this approach is that the identified concepts apply to every traceability scenario. Due to the generic nature of such a metamodel, trace links can relate any number of model elements of any type in any model. Although this approach provides great flexibility and freedom, it has two main shortcomings. First, it can allow the establishment of illegitimate links. Additionally, models which conform to a generic trace metamodel cannot capture case-specific traceability information with rigorously defined semantics [18]. To overcome these shortcomings, case-specific trace metamodels are proposed.

Despite the benefits gained from the use of case-specific trace metamodels, the extra effort and time required for the development of such metamodels is an area of concern. Through experimentation with defining a number of case-specific trace metamodels, we have observed that although all of them are different, they are quite similar in several aspects. For example, all of the traceability metamodels contain the concepts of `TraceLink` and `TraceLinkEnd`. However, for each particular traceability scenario the types of those two constructs differ and this is the area of variability among the different traceability metamodels. Additionally, a set of constraints are common in different traceability metamodels. For example, a particular `TraceLink` can link elements of a particular type only or it can also link to instances of all the subtypes of the type. To reduce the time and effort required for the development of traceability metamodels, we have identified a number of recurring patterns in terms of the structure as well as in terms of the correctness constraints that guarantee the semantic integrity and completeness of the trace models, and we have promoted those patterns into first-class metamodel elements.

TML can be considered as a DSM2L for trace metamodels. The core of the TML metamodel is illustrated in Fig. 2. In addition, the semantics of TML are specified in a translational manner using two formal and executable transformations that can transform a TML model into an ECore metamodel and a set of constraints expressed using the Epsilon Validation Language (EVL) [16] which is an extension of OCL [18]. We have evolved and refined the syntax and the semantics of TML over a number of diverse case studies. In its current form,

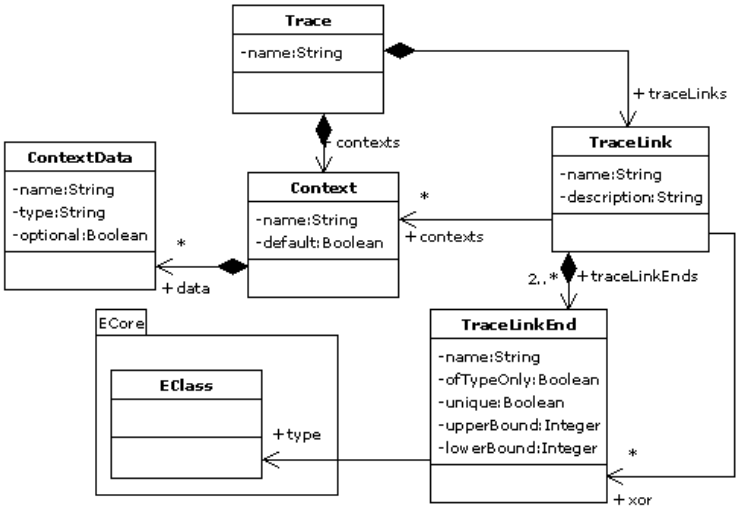

Fig. 2. Core of the TML (from [18])

TML is capable of capturing complex traceability metamodels and to our view the language is well-balanced between expressiveness and domain specificity.

With respect to the challenges identified in Sect. 2, we can critically evaluate TML as follows:

1. *Systematic reuse of DSL constructs and infrastructure elements.* We manage to achieve systematic reuse of the various traceability DSL constructs and infrastructure elements by promoting the recurring patterns among the various traceability metamodels into first-class entities in TML.
2. *Systematic support for specifying variability between the DSLs.* We manage to specify the variability among the various traceability metamodels, namely the different types of `TraceLinks` and `TraceLinkEnds`. One limitation of our approach is that we are not able to automatically generate *all* the relevant constraints for a traceability metamodel. Using TML, we can automatically generate constraints which apply to all types of `TraceLinks` and `TraceLinkEnds`. However, in our experiments with specifying various traceability metamodels, we have encountered constraints which apply only in particular cases. In our approach such constraints have to be specified manually.

4.2 VML*

A *software product line* (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core

assets in a prescribed way [13]. Apart from sharing a common set of features, every system also has features that are specific to this system and not shared with other systems in the SPL. An important part of managing the features of a product line and the individual systems (often called *products*) is to model the available features and their dependencies (e.g., if feature `A` is selected, feature `B` also must be selected) in an abstract form, called a feature model (e.g., [27]). While these models express what features there are and what products can be formed from them, they do not express how a specific feature is realised and, thus, how any specific product is realised. Feature models are strictly problem-space models. The realisation of the product line is designed in a different set of models using a different set of modelling languages.

This leads to a *mapping problem:* For each feature in a feature model we need to identify and specify the solution-space models and model elements associated with it to be able to systematically construct products given a selection of features. A number of different approaches for such mappings have been proposed [14, 21, 24, 48]. Most of the approaches propose a generic modelling language for expressing simple relations between features and model elements. In contrast, our VML* approach [48] proposes to construct customised languages for each solution-space modelling language used in an SPL. The main benefit of such an approach is the ability of providing more sophisticated mapping relations that have been custom designed for the specific solution-space modelling language. For example, if activity diagrams are used as part of the solution-space models, we can provide a mapping that merges additional steps into an activity if a certain feature has been selected. For class-diagram models we may provide a mapping relation that introduces package-merge dependencies in a model. A more detailed discussion of the respective benefits and drawbacks of these approaches can be found in [48].

All of these languages are relatively similar. Still, without support for reuse, it can be tedious and error prone to develop the support infrastructure for a new VML language. As a solution, we developed VML*, which is a DSM2L for VML languages. This is possible, because all VML languages share a common core metamodel, which can be seen in Fig. 3. The metamodel highlights in dark grey those metaclasses which must be adapted for each specific VML language. Most importantly, new subclasses of `Action` must be created defining the different specific mapping relations (called *actions* in VML*) available. The implementation of each such action is provided as a model-transformation snippet.

To enable language designers to define these variations between VML languages while reusing as much as possible of the commonalities shared by all VML languages, VML* defines a DSM2L for so-called language-instance descriptors. The metamodel for this DSM2L can be seen in Fig. 4. From instances of this metamodel, we can then generate the complete infrastructure for a VML language, including a metamodel expressed in EMF/Ecore, an editor with syntax highlighting, code completion and checking of static semantics, and evaluation engines for different evaluations of a VML specification (e.g., automatic gener-

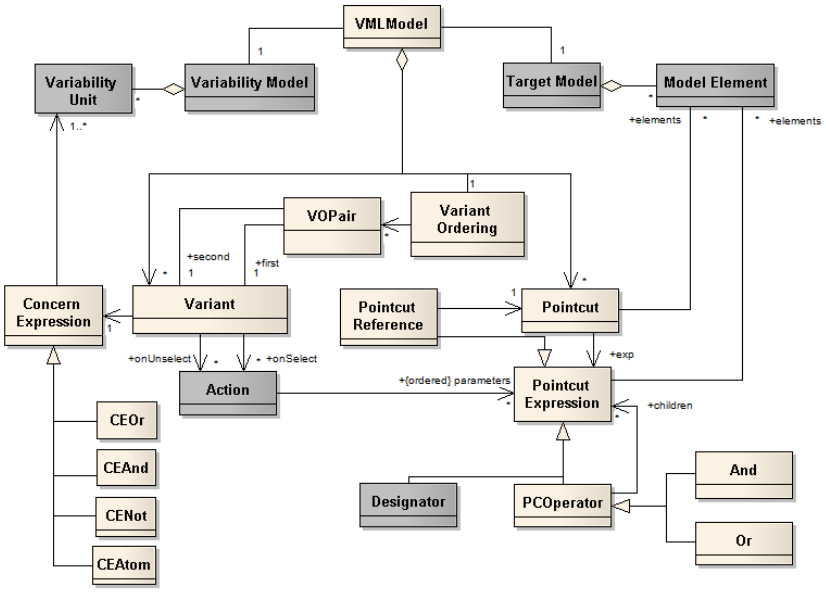

Fig. 3. Common core metamodel for VML languages (from [48])

ation of product models based on a selection of features, or generation of trace links between features and model elements).

We have used VML* to develop a number of languages already. Most importantly, we have developed VML4Arch [35]—a language for mapping from feature models to architecture models—and VML4RE [3]—a language for mapping from feature models to requirements models. Other languages for mapping to model-transformation workflows or project-specific DSLs are currently under development.

Figure 5 shows excerpts from key artefacts for VML4RE. Figure 5 a) shows an excerpt from the language-instance descriptor (i.e., an instance of the VML* DSM2L). This has a number of sections defining different parts of the language. The first two sections are concerned with the connection to feature models and target models, respectively. They define what `VariabilityUnit`s and `ModelElement`s VML4RE supports and how to extract them from a given model. The next section defines the syntax of the different `Action`s supported by VML4RE; all that is needed here is the name of each action and the types of its parameters. Finally, the `aspects` section defines the different evaluation aspects of VML4RE; that is, its different semantics for different usages of the language. In the example, we show a semantics for deriving a model transformation for product instantiation and a semantics for deriving trace links from a VML4RE specification. For the former, for each action we provide an implementation as a model-transformation function realised in xTend (see Fig. 5 b) for an example function).

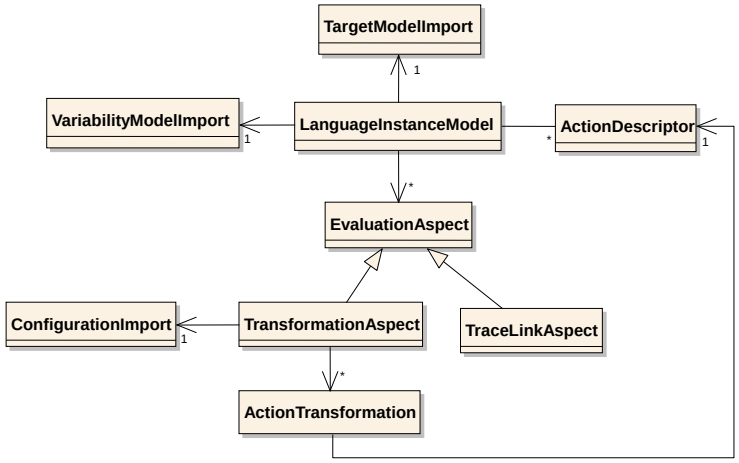

Fig. 4. Domain-specific metamodelling language for VML* (from [48])

For the tracing semantics we define which of these operations create or remove model elements. Trace links will then be created automatically to link selected features with created and removed model elements. Finally, Fig. 5 c) shows an excerpt from a VML4RE specification. The language's metamodel, concrete syntax, and evaluation infrastructure have been completely generated from the specifications exemplified in Fig. 5 a) and b). The example VML4RE specification excerpt states that when the feature `Security` has been selected for a product, two new use cases `SecureTheHouse` and `ActivateSecureMode` are created and that `SecureTheHouse` includes the existing use cases `SendSecurityNotification` and `OpenAndCloseWindowsAutomatically`.

Effectively, for VML* we have developed a DSM2L for specifying individual VML languages. With respect to the challenges from Sect. 2, we can make the following evaluation:

1. *Systematic reuse of DSL constructs and infrastructure elements.* The generator infrastructure built to support the DSM2L supports full reuse of all shared concepts and infrastructure parts for all VML languages.
2. *Systematic support for specifying variability between the DSLs.* At the same time, it enabled us to specify differences between the languages in one place; the generators then ensure that all required co-ordinated changes to the infrastructure are performed consistently. For VML*, we have not been able to fully support the specification of individual languages in the DSM2L only. Instead, semantics of actions and adapters to feature and target models must be written as rules in a separate model-transformation language[4]. However,

[4] The prototype uses openArchitectureWare's xTend language, but any other could have been used as well.

Fig. 5. Example files for VML4RE [3]: a) language instance descriptor (DSM2L instance), b) model-transformation code implementing the `createInclude` action, and c) excerpt of a VML4RE script specifying the modifications required when the `Security` feature has been selected. From [48]

as each of these rules is well encapsulated and is referenced from exactly one place in the DSM2L, the situation is sill better than having to make multiple co-ordinated changes in multiple places in the VML support infrastructure for each action.

Finally, as an interesting side effect of using a DSM2L for VML* we had to streamline some of the syntax of the original VML languages from which the language family was constructed. For example, VML4Arch used to have an action whose syntax was `connect Component1, Component2 using Interface`. In VML* all actions must follow an operation call syntax, so this had to be changed to `connect (Component1, Component2, Interface)`.

4.3 Reuseware Reuse Extension Language

Reuseware [22] is a metamodel-agnostic approach to aspect-oriented modelling (AOM). It is based on interpreting some elements of a model written in some modelling language as special—*addressable*—points. Addressable points can be used to specify either fragments of a model or points to be replaced by a fragment from some other model. Reuseware's composition engine can interpret these

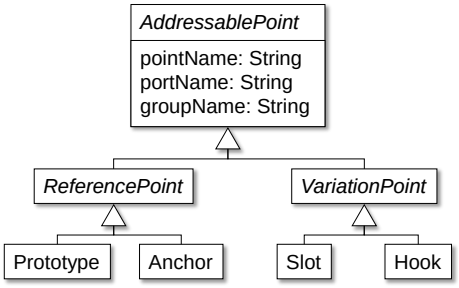

Fig. 6. Metamodel of addressable points. From [22]

addressable points and use them to perform aspect-oriented compositions of models. Figure 6 shows the metamodel of addressable points used by Reuseware.

To make Reuseware's AOM concepts available to a language `X` (called the component language in [22]), we need to generate a new language `X'` weaving the metamodel from Fig. 6 into the metamodel of `X`. Not every concept from `X` needs to be made available as an addressable point. In fact, language designers need to carefully choose, which concepts to make available as which form of addressable points, and which concepts not to expose as addressable points at all. To express these choices, Reuseware provides a Reuse-Extension Language, the metamodel of which can be seen in Fig. 7. For the moment, we focus on `SyntacticExtension` only. These effectively enumerate the model elements from `X` that are to be exposed as addressable points. Based on such a specification, [22] defines an algorithm for deriving a new language `X'` that includes these addressable points and can be successfully interpreted by Reuseware's composition engine to describe aspect-oriented model compositions. In effect, the Reuse-Extension Language is a DSM2L for AOM languages constructed with Reuseware. `SemanticExtension` is an interesting variation on our concept of DSM2Ls: Similarly to `SyntacticExtension`s, these identify model elements to be exposed as addressable points. However, instead of physically generating a new language, `SemanticExtension`s define a new *interpretation* of `X`; that is, the diverse OCL expressions of a `SemanticExtension` are interpreted by Reuseware's composition engine for every instance of `X` that is used in a composition program. The effect is similar to `SyntacticExtension`s, however, more fine-grained adjustements are possible.

With respect to the challenges from Sect. 2, we can evaluate the Reuseware Reuse-Extension Language as follows:

1. *Systematic reuse of DSL constructs and infrastructure elements.* The DSL constructs that are reused are defined in their own metamodel (cf. Fig. 6), which is woven into the component-language metamodel by a standard algorithm defined in [22]. The complete support infrastructure is implemented in the Reuseware composition engine independently of concrete component languages. Furthermore, existing editors for the component language are in-

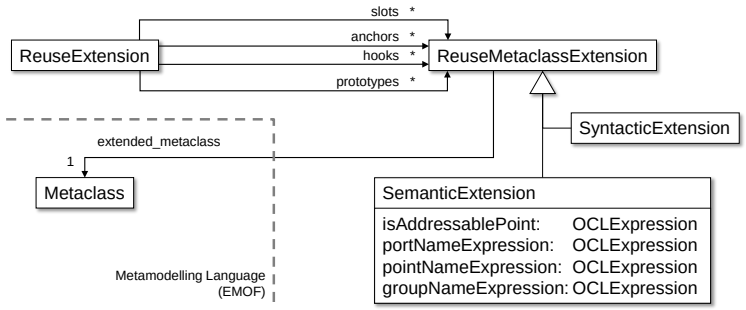

Fig. 7. Metamodel of the Reuse-Extension Language, the DSM2L defined by Reuseware [22]

tegrated into the composition infrastructure transparently. Note, however, that this is only possible for semantic extensions. Syntactic extensions modify the metamodel of the component language, meaning that model components may not be accessible to existing editors for the component language.

2. *Systematic support for specifying variability between the DSLs.* The different DSLs vary in two dimensions:
   (a) *The component language used.* This variation is supported because all reusable elements of the infrastructure are implemented without reference to any specific component language. Thus, a reuse specification can be written for any Ecore-based component language.
   (b) *The model elements that are exposed as addressable points.* The allowed variations along this dimension are captured by the Reuse-Extension Language. An interesting difference to the other examples of DSM2Ls above is that this variability can be bound in two different forms: Syntactic extensions *physically* create a new language, binding the variability at the *metamodelling* level. Semantic extensions are interpreted and *logically* create a new language, binding the variability at the *modelling* level.

5 Related Work

In this section we discuss an approach to model differences representation that shares many characteristics with the approach proposed in this paper and provide an overview of related work in the broader field of DSL development and families of DSLs.

5.1 DSM2L for Model Differences Representation

In [5], a metamodel-independent approach is presented for the representation of model differences. Given two models, M1 and M2, which conform to an arbitrary metamodel MM, their difference conforms to another metamodel MMD,

derived from the former by an automated transformation. MMD has to provide the constructs able to express the modifications that have to be performed on the initial version of a given model in order to obtain the final one. Cicchetti suggests that although the constructs in the MMD metamodel are case specific, i.e. depend on the metamodel MM under consideration, there is a common set of constructs shared across all metamodels, to which difference models conform. Hence, he defines a base metamodel for representing differences, and then all the case-specific difference metamodels are derived as an adaptation of the base metamodel to the given precise domain. This is achieved by generating the required case-specific difference meta-elements as specializations of the difference meta-elements defined in the base metamodel. This specialization of the base metamodel towards a given language is achieved by specifying a Higher Order Transformation [43], i.e. a transformation which takes as input and/or generates as output model transformations.

Similar to our notion of DSM2Ls, Cicchetti's approach is also based on generating context-specific metamodels. However, there is no explicit notion of a DSM2L. Instead, the generation can be generically applied to any metamodel expressed in a generic metamodelling language such as MOF. This is possible, because the relation between the generated metamodels (or languages) to their contexts of use is different from the cases discussed in our paper: In Cichetti's case [5], the contexts are the individual metamodels for which difference metamodels are to be derived. The specific difference metamodels are then standardised derivations of the metamodel for which they are created. Instead, for the DSM2L case, the languages to be created are customised in an open manner for each context; that is, just given the context we cannot automatically derive the customised metamodel of our DSL. Additional input from the language developer is required. This input is provided using the DSM2L. Hence, if [5] provided some means for selecting the metaclasses for which to generate difference-representing elements (similarly to the Reuseware example from Sect. 4.3), we would classify this as a DSM2L.

5.2 Development of DSLs

The development of a DSL typically involves the following steps [6]:

1. *Analysis.* The problem domain is identified, the relevant domain knowledge is gathered and an appropriate DSL is designed.
2. *Implementation.* A suitable implementation approach is chosen and used to realise the DSL and its supporting infrastructure.
3. *Use.* Programs are written in the new DSL, and if necessary feedback on design and implementation is provided.

The focus of this paper is on DSL implementation, where we discuss an interesting pattern for implementing families of DSLs that exhibit certain characteristics. Consequently, we will first discuss a number of key approaches to DSL implementation, before giving an overview of some other work on particular families of languages.

Klint et al. propose a more systematic approach to the development of languages (or *grammarware* as they call it), called Grammarware Engineering [29]. Our work fits right into this research agenda, as we have identified a pattern for systematic development of families of DSLs. Among other things Klint et al. identify *meta-grammarware* as a term to "[...] refer to any software that supports concrete grammar use cases by some means of meta-programming, generative programming or domain-specific language implementation [...]" [29]. In this sense, our work can be classified as meta-grammarware.

Language Implementation The direct approach to DSL (and GPL) implementation is the development of a *compiler* or *interpreter* for the DSL under consideration. There is a wide variety of tools and frameworks for the development of compilers and interpreters such as [1, 6, 34]. The main advantage of such an approach is that the implementation can be tailored completely towards the DSL. In addition, error detection, static analysis, and optimizations can be performed at the domain level. The main disadvantage of this approach is the high cost of implementation.

Designing DSLs as *embedded languages* is one way to avoid the complexity of building a complete compiler or interpreter. Embedded languages [41] are DSLs that are used inside larger programs written in a host language. Expressions in the DSL are translated into the host language producing a new program completely expressed in the host language that is then evaluated using the standard tools for the host language. Different implementations of this concept have been proposed in the literature—for example, [10, 19, 37]. Other proposals to simplifying language implementation involve extensible compilers (e.g., the delegating compiler objects approach [25]), grammar extension and inheritance (e.g, [36, 38]), and pre-processing (such as, macro-processing [9] or source-to-source transformations [44]).

Model-driven development can benefit greatly from the use of DSLs. Here, a DSL is typically developed using an abstract-syntax metamodel (sometimes also called domain-definition metamodel [8]), one or more concrete-syntax definitions (textual or graphical) and a specification of language semantics (often defined using model transformations). A number of frameworks to support such language development have been proposed—for example, [7, 11, 23, 28, 30, 33, 12, 38, 39, 42, 47]. We have previously proposed an approach for simplifying the development of the model transformations involved in defining such DSLs [26].

All of these approaches focus on the manual development of a single DSL. In contrast, the approach presented in this paper aims to support the development of families of languages by generating their implementation from a specification in a DSM2L. It is, thus, complementary to the approaches discussed above and can be combined with any one of them depending on the circumstances. Using an approach that already supports modular language definition may simplify the code generators to be written for interpreting the DSM2L. Apart from using these approaches in the result of the code generator, we of course also make use of them in developing the DSM2L in the first place. In particular, the model-

driven approaches with their focus on model transformation and generation as the means of providing semantics to a DSL are useful for implementing DSM2Ls.

Families of Languages Families of languages have been presented in the research literature for a range of domains: Voelter presents an approach for a family of languages for architecture design at different levels of abstraction [46], Akehurst et al. [2] present a redesign of the Object Constraint Language as a family of languages of different complexity, Visser [45] presents WebDSL, a family of interoperating languages for the design of web applications. All approaches, including ours presented in this paper, use very different kinds of technologies for their specific case: Voelter uses conditional compilation to construct an appropriate infrastructure, Akehurst et al. use a special parser technology that enables modular language specification, Visser uses rewriting of abstract syntax trees and our approach generates a monolithic infrastructure for each language.

6 Outlook and Conclusions

This paper has identified an important issue with the design and implementation of sets or families of DSLs: that there are common recurring patterns and variations across families of DSLs. This issue led us to define a set of challenges for building language families, which in turn led to the introduction of a new concept: that of domain-specific metamodelling languages. We described how to implement domain-specific metamodelling languages, via a transformational approach that eliminates difficulties with instantiation of metamodels. The approach was then evaluated based on three examples: a traceability metamodelling language, a metamodelling language for variability, to be exploited in the development of software product lines, and a language for describing aspect-oriented language extensions.

In this evaluation, we have identified limitations in our current approach to domain-specific metamodelling that we will address in the future. In particular, in the VML* example, we were not able to fully specify all language properties using the domain-specific metamodelling language alone—semantics of actions, for example, had to be encoded separately using a model transformation language. As we go on to apply the notion of DSM2Ls, it will be interesting to see if a similar pattern emerges for other language families as well. Additionally, in the TML case, although we were able to capture all structural constructs of the traceability metamodels as well as some of the accompanying corectness constraints, we were unable to generate via the use of the DSM2L the case-specific constraints, which had to be specified manually. This indicates that TML might need to provide appropriate extension mechanisms for those cases. In the AOM example, reuse of the component-language infrastructures was sometimes hampered by the fact that we had generated a new language. An alternative approach was to interpret the DSM2L information instead of generating a new language. In general, the issue of when and how to bind variability in languages of a family of DSLs deserves further research attention.

We are in the process of further evaluating our approach in a different domain and context: that of model management. Specifically, we are attempting to apply the concept of domain-specific metamodelling languages to the redesign of the Epsilon model management framework [40]. Epsilon[5] is a platform of task-specific languages for model management; it includes languages for model-to-model transformation, model-to-text transformation, model merging, model validation and inter-model consistency checking, model comparison, and model refactoring, all based on a core navigation and modification language (which combines language features of OCL and Javascript). The languages in Epsilon are inter-related, and interoperate [31]; for example, the model merging language reuses the model-to-model transformation language, and can use the output from a program expressed in the comparison language. In effect, Epsilon forms a family of integrated languages for model management; however, they have been developed and implemented without applying domain-specific metamodelling concepts (see [15] for details of how Epsilon is currently implemented based on grammar inheritance and by using virtual machines). Epsilon thus provides a rich conceptual and technical domain in which to further evaluate the domain-specific metamodelling approach, and our future efforts will focus on re-engineering Epsilon along these lines.

Acknowledgments

The work presented in this paper was funded by the European Commission through FP6 projects MODELPLEX and AMPLE.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
2. David Akehurst, Steffen Zschaler, and Gareth Howells. OCL: Modularising the language. In *Ocl4All: Workshop at MoDELS 2007, Electronic Communications of the EASST*, volume 9, 2008.
3. Mauricio Alférez, Uirá Kulesza, Nathan Weston, Joao Araujo, Vasco Amaral, Ana Moreira, Awais Rashid, and Michael C. Jaeger. A metamodel for aspectual requirements modelling and composition. AMPLE Deliverable 1.3, http://ample.holos.pt/gest_cnt_upload/editor/File/public/AMPLE_WP1_D13.pdf, 2007.
4. Anneke Kleppe. A Language Description is More than a Metamodel. In *Proc. 4th International Workshop on Software Language Engineering*, Nashville, USA, October 2007.
5. Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Dipartimento di Informatica, Universita di L'Aquila, 2008.
6. Arie Deursen and Paul Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[5] http://www.eclipse.org/gmt/epsilon

7. Atlas and LINA Project Team. AMMA Platform. http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/, 2008.
8. Jean Bézivin, Frederic Jouault, Ivan Kurtev, and Patrick Valduriez. Model-based DSL Frameworks. In *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, October 2006.
9. Claus Brabrand and Michael Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40. ACM, 2002.
10. Martin Bravenboer and Eelco Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In *4th International Workshop on Software Language Engineering (ATEM),*, 2007.
11. Carsten Amelunxen and Alexander Königs and Tobias Rötschke and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proc. Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066, pages 361–375. LNCS, Springer, 2006.
12. Tony Clark. XMF-Mosaic. http://itcentre.tvu.ac.uk/~clark/Software.html, 2009.
13. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
14. Kryzstof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proc. 4th Int'l Conf. on Generative Programming and Component Engineering (GPCE'2005)*, pages 422–437, 2005.
15. Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, Department of Computer Science, The University of York, York, United Kingdom, June 2008.
16. Dimitrios S. Kolovos and Richard F. Paige and Fiona A.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, 2007.
17. Nicholas Drivalos, Richard F. Paige, Kiran J. Fernandes, and Dimitrios S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. ECMDA Traceability Workshop*. SINTEF Technical Report, 2008.
18. Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a DSL for software traceability. In *1st Int'l Conf. on Software Language Engineering (SLE 2008), Revised Selected Papers*, pages 151–167. Springer-Verlag, 2009.
19. Erik Van Wyk and Oege de Moor and Kevin Backhouse and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *11th International Conference on Compiler Construction*. LNCS, Springer, April 2002.
20. The Eclipse Foundation. Eclipse Modelling Framework Project. http://www.eclipse.org/modeling/emf/, 2008.
21. Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Proc. 12th Int'l Conf. on Software Product Lines (SPLC'08)*, pages 139–148. IEEE, 2008.
22. Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE*, 2009. To Appear.
23. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Proc. 5th Euro-*

pean Conf. on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09), 2009.
24. Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping features to models. In *Companion 30th Int'l Conf. on Software Engineering (ICSE'2008)*. ACM, 2008.
25. Jan Bosch. Delegating compiler objects: modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1):66–92, 1997.
26. Jendrik Johannes, Steffen Zschaler, Miguel A. Fernández, Antonio Castillo, Dimitrios S. Kolovos, and Richard F. Paige. Abstracting complex languages through transformation and composition. In Andy Schuerr and Bran Selic, editors, *Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'09)*, 2009. To Appear.
27. Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-0211990, Software Engineering Institute, 1990.
28. Steve Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. Wiley, 2008.
29. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for Grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.
30. Dimitrios S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website. http://www.eclipse.org/gmt/epsilon, 2007.
31. Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. A framework for composing modular and interoperable model management tasks. In *Model-Driven Tool and Process Integration Workshop*, pages 79–90, 2008.
32. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On-demand merging of traceability links with models. In *ECMDA 2006 Traceability Workshop Bilbao*, 2006.
33. Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, 2001.
34. Lloyd Nakatani and Mark Jones. Jargons and infocentrism. In *1st Acm SIGPLAN Workshop on Domain-Specific Languages*, 1997.
35. Neil Loughran, Pablo Sánchez, Alessandro Garcia, and Lidia Fuentes. Language support for managing variability in architectural models. In *Proc. 7th Int'l Symposium on Software Composition (SC'08)*, pages 36–51, 2008.
36. Marjan Mernik and Viljem Zumer and Mitja Lenic and Enis Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices*, 34(6):68–75, jun 1999.
37. Martin Bravenboer and Rene de Groot and Eelco Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, Braga, Portugal,, 2005.
38. MontiCore. Project Website. http://www.sse-tubs.de/monticore/, 2007.
39. openArchitectureWare. openArchitectureWare Project Website. http://www.eclipse.org/gmt/oaw/, 2008.
40. Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A.C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proc. ICECCS*, pages 162–171, 2009.

41. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.
42. Triskell Team. KerMeta Platform. http://www.kermeta.org, 2008.
43. Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *ECMDA-FA*, pages 18–33, 2009.
44. Todd Veldhuizen. Blitz++ User's Guide. Version 1.2. http://www.oonumerics.org/blitz/ manual/blitz.ps., 2001.
45. Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Int'l Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer, 2008.
46. Markus Voelter. A family of languages for architecture description. In *8th OOPSLA Workshop on Domain-Specific Modeling*, 2008.
47. Jules White, Douglas C. Schmidt, and Sean Mulligan. The generic eclipse modeling system. In *Model-Driven Development Tool Implementer's Forum, TOOLS '07*, 2007.
48. Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. VML* – a family of languages for variability management in software product lines. In *Proc. 2nd Int'l Conf. on Software Language Engineering (SLE'09)*, 2009.