Weaving of Domain-Specific modelling Languages
A literature review

Rafael Ugaz
rafael.ugazriofrio@student.uantwerpen.be

January 30, 2014

Abstract

As the use of Domain Specific modelling Languages (DSLs) in model-driven engineering (MDE) increases, so does the need for their combination. The combination of DSLs allows for the re-use of existing DSLs so that designers do not have to start from scratch. It also enables the use of smaller DSLs to specify a complete system, where each DSL provides a different *view* of it. These features, already present in the programming languages world, provide reusability and modularity, which in turn can increase the productivity and overall quality of the final product. The term weaving is inspired by aspect oriented modelling (AOM) where aspect models are combined or weaved into a base model. In this paper, a literature review of the current techniques for DSL weaving is provided, along with a classification of approaches in order to discover areas for future research.

1 Introduction

As the use of Domain Specific modelling Languages (DSLs) [14] in model-driven engineering (MDE) [34] increases, so does the need for their combination. The combination of DSLs allows for the re-use of existing DSLs so that designers do not have to start from scratch. It also enables the use of smaller DSLs to specify a complete system, where each DSL provides a different *view* of it. These features, already present for programming languages, provide reusability and modularity, which can increase the productivity and overall quality of the final product. We define DSL weaving as the combination or composition of two or more DSLs together. The term weaving is inspired by aspect oriented modelling (AOM) where aspect models are combined or woven into a base model. In this paper, a literature review of the current techniques for DSL weaving is provided, along with a classification of approaches in order to discover areas for future research.

A DSL is formed by three main elements: abstract syntax, concrete syntax and semantics. The main idea of DSL weaving in this paper is that, since the three elements can be represented as models, the strategy for DSL combination decomposes into three different model combinations. DSL weaving has many applications in the area of DSL engineering since it is required for any approach that follows the separation of concerns principle. This principle states that a system can be modelled by multiple models where each one represents an important view or aspect of it. Modelling a system in this manner provides modularity and reusability, two important aspects of MDE. The main contribution of this paper is a comparison of the current model combination techniques according to several relevant criteria in order to lay the groundwork for future work focused on DSL weaving.

The paper is organized as follows: section 2 gives some background information and terminology in the context of the combination of modelling and DSLs, section 3 lists the model and language combination techniques obtained from the literature review, section 4 lists the criteria used for the classification of these techniques, section 5 analyses the results of the classification and finally, section 6 concludes the paper with some final considerations and future work.

1.1 Domain Specific modelling

Domain-Specific modelling (DSM) aims to raise the level of abstraction beyond current programming languages by specifying the solution in a language that directly uses concepts specific to the problem domain [14]. This allows domain-experts (possibly with no programming expertise) to play active roles in development efforts by modelling the solution using only familiar domain constructs [22]. The final products of a DSM solution are complete artifacts equivalent to those that developers used to write by hand (e.g. executable programs, documentation, test cases or other models). These artifacts are automatically generated from their high-level specifications via domain-specific transformations. These transformations can have as result another model (of an intermediate modelling language) or they can generate code (of a textual general purpose language), at which level no further transformations are usually performed.

This automation avoids the need to perform error-prone and time-consuming mappings from domain to design and to programming language concepts. Since this translation is automated, it can be repeated effortlessly,
it effectively raises the level of abstraction to the domain-specific modelling level and it hides all the other lower levels from the user.

DSM can also provide translation in the other direction, i.e. from artifacts (e.g. code) to domain specific model, that can be useful for inspecting the system at runtime, debugging, formal analysis results, etc. This is possible by means of traceability links between the different levels of abstraction that allow obtaining the equivalent of a certain concept at a higher or lower level of abstraction of the DSM solution process.

1.2 Domain Specific modelling Language

The raise in abstraction is achieved by the specification of a *Domain Specific Language* (DSL). A DSL is a modelling language that contains concepts and rules that represent the application domain. They provide the abstraction for development and are therefore the most visible part for developers of a DSM solution.

Just like languages in general, a modelling language is formally defined in the following way: it consists of syntax and semantics. The syntax, which defines the form of a language, is divided into abstract and concrete syntax. The former defines the concepts and rules of a language and specifies what a valid instance of a language looks like. The latter takes care of the notation used to represent these concepts, be it textually or visually. Finally, the semantics define the meaning of the modelling concepts of the language. A formal definition of a DSL can be found in section 2.

The next subsections explain the three components of a DSL and its importance in more detail.

1.2.1 Abstract Syntax

The abstract syntax describes the concepts of a language and their properties, the legal connections between them, the model hierarchy structures, and also grammatical rules that enforce model correctness. These rules can significantly reduce the possible design space which helps in designing correct applications.

The abstract syntax of a modelling language is normally specified with a meta-model [19]. The prefix ”meta” is used to denote that an operation, in this case modelling, is applied twice. In other words, a meta-model is a conceptual model of a modelling language. A meta-model provides a formal specification of the language which, supported by tools (e.g. AToMPM, MetaEdit+), is used to create and modify models as well as generating code from them.

1.2.2 Concrete Syntax

The concrete syntax provides a representation of the abstract syntax of a meta-model as a mapping (syntactic mapping) between the meta-model concepts and their textual or graphical representation. A language can have several concrete syntaxes.

1.2.3 Semantics

The semantics of a modelling language define the meaning of the syntactically correct concepts of a modelling language. Semantics can be further divided by a semantic domain,

The two main approaches for defining semantics formally for a modelling language are:

- *Operational* semantics often encode behavior and give meaning to a language by describing the transformation of the system from one state to the next (possibly along some time model).

  They are usually better suited for giving meaning to behavioral languages (e.g. finite state automata (FSA), Petri nets or activity diagrams). Since structural languages (e.g. class diagrams or entity relationship diagrams) have a more static and non-behavioral nature, they cannot be very well represented by this type of semantics.

- *Denotational* semantics define the meaning of a language in terms of another language or formalism for which well defined semantics (operational or denotational) exist, for example code, mathematics or Petri Nets. With this kind of semantics, any type of language (behavioral or structural) can be specified, as long as the language it is being defined in can properly represent its concepts.

The specification of the semantics is not as straightforward as the specification of the syntax, which can be specified through the definition of a metamodel. This is because models may have different interpretations or meanings and therefore, a DSL might have different several semantic domains and mappings associated with them.

Since there is no real standard for defining semantics (like the metamodels are for the abstract syntax), there is generally a lack of precise and explicit specification of modelling language semantics. This creates the risk of semantic mismatch between tools or applications and it also makes the composition of semantics impossible since they are not specified.

1.3 Why DSL weaving

The action of weaving or combining DSLs provides many benefits to the process of DSL engineering, which in turn is a vital phase of domain-specific modelling, as explained before. In this section, the major applications of DSL weaving are discussed.

1.3.1 Reusability

Most of the time, language engineers have to develop DSLs completely from scratch, using only their own experience and expertise to do so [7]. This is in contrast to general purpose or third-generation programming languages (3GL), where an engineer is able to re-use the existing work of others to speed up the developing process or improve the quality of the product.

For DSLs, composition is one of the missing processes for enabling this feature, along with a more formal specification of them and a library of stable and established DSLs. This would bring to domain specific modelling the same benefits that software reuse brought to 3GLs [7].

In summary, some of the benefits that reusability brings are the following:

- Avoidance of duplication of effort (no overlap of concepts)
- Emergence of high-quality reusable DSLs (e.g. Statecharts, PetriNets)
- Recognition of key DSL modelling patterns and best practices
- Significant reduction in the time-to-market of DSLs.

1.3.2 Multi-paradigm modelling

When faced with the task of modelling a complex system, breaking it down into a set of models where each one deals with a different concern makes the task much easier [13]. Furthermore, these models can also be defined in different languages, each one better suited for the different view or aspect of the system that it abstracts. For instance, the behavioral concern of the system can be designed using State Charts based language while the structural part uses UML Class Diagrams. An example of this approach can be seen in the PhoneApps application, which is a multi-concern DSL [23] This is also more user friendly to the designers or stakeholders who might be more comfortable with a certain style of language and notation. Multi-paradigm modelling is therefore a powerful mechanism for achieving a higher level of abstraction, simplicity and modularity in modelling.

The problem arises when the design phase is finished and it is time to create the final application, which requires the smaller models of the system to be synthesized back together in some way. If the models were created with the same modelling language (i.e. they conform to the same meta-model), a *homogeneous* composition is needed. But if, on the other hand, the models were designed in different modelling languages or paradigms, they require a *heterogeneous* composition. The latter being, as expected, more difficult since it would require a translation mechanism between the languages by, for example, an interface.

1.3.3 Modularity

Modularity of modelling languages is a design technique that seeks to separate the functionality of a system into independent modules. The separation between models is clean explicit since they only interact with each other by means of well defined interfaces that effectively encapsulate them. This concept can be considered similar to multi-paradigm modelling (section 1.3.2) although modularity does not require different modelling paradigms or vice versa.

Modularity has benefits similar to those of encapsulation in software engineering, namely a clean separation of, in this case, the models. It also makes the system easier to maintain and extend because of its modular structure as opposed to a tangled one where two tasks become unnecessarily difficult.

But modularity also has the disadvantage of being less efficient performance wise than its counterpart, although this can be remedied by code optimization. The more modular a system is, the more steps are needed to perform a task. A simple example is the access of an object of a module many associations away from another one; in a modular system, a chain of consecutive accessing operations through all modules in between would be needed while in a non modular system this could be done directly with a single operation. An example of a modular language is the Discrete EVent System Specification formalism (DEVS) [39].

1.3.4 Liskov's substitutability

In object oriented engineering, Liskov's principle of substitutability is a fundamental concept. It states that if an object Sub has an is-a relationship with another object Super (for example an inheritance in OO programming), then in every situation where Super is used, Sub can be used as well. In other words Sub can substitute Super. This principle may seem simple but is also required in the MDE field and each of the composition techniques described in this paper must not violate it.

2 Background and Terminology

This section presents a list of definitions for a DSL composition framework that will be used throughout this paper. It is partly based on the definitions proposed by Bézivin et al. [3], [20] and [2].

Since the context of this paper is model-driven engineering, models are the central concept. A formal way to define models which is also useful for graphical modelling is representing models as (directed) graphs.

Directed Graph

A directed multigraph $G = (N_G, E_G, \Gamma_G)$ consists of three elements:

- N_G, a finite set of nodes,
- E_G, a finite set of edges,
- $\Gamma_G : E_G \to N_G \times N_G$, a function that maps edges to their source and destination nodes.

Model

A model $M = (G, \omega, \mu)$ is a triple where:

- $G = (N_G, E_G, \phi_G)$, is a directed multigraph,
- ω is a model (called the reference model) with an associated graph G_ω,
- $\mu : N_G \cup E_G \to N_\omega$, is a function that maps all the nodes and edges of G to nodes of ω.

The relation between a model and its reference model is called *conformance*. Thus we can say that a model *conforms* to its reference model.

Metametamodel

It is the metamodel of the metamodel. In practice it is also its own reference model, i.e. it conforms to itself. This “metacircularity” avoids endless “meta” prefixes.

Metamodel

A metamodel is a model such that its reference model is a metametamodel.

Terminal model

A terminal model is a model such that its reference model is a metamodel

Correspondence model

A correspondence model $C = (G_C, \omega, \mu)$ consists of links between elements of different models, such that:

- $S = \{M_i = (G_i, \omega_i, \mu_i);\ i = [1..n]\}$, is a set of different models,
- G_c has only two types of nodes: *links* and *link endpoints*,
- for each link endpoint in G_C, there is a link connected to it through an edge,
- each link endpoint in G_C refers to an element e (node or edge) of a model M_i of the set S.

Match operation

The match operation $C = Match(S)$ takes as input a set of models $S = \{M_i = (G_i, \omega_i, \mu_i);\ i = [1..n]\}$, searches for equivalences between their elements and produces a correspondence model C as output.

The semantics of the match operation are not fixed. They consist of a combination of comparison and conformance rules. Comparison rules determine *syntactic* similarities between model elements. Conformance rules determines if syntactically similar elements are also *semantically* compatible.

Transformation Rule

A transformation or *graph* rule $R : M_{IN} \rightarrow M_{OUT}$ is a function that takes a model M_{IN} as input, performs a matching of the LHS subgraph over it, replaces the match with another subgraph RHS and outputs the model M_{OUT}. A graph rule is parametrized by:

- LHS, a left-hand side pattern that should be present in the input model,
- RHS, a right-hand side pattern that describes how the LHS should be transformed after applying the rule,

- NAC, a negative application condition pattern that should not be present in the input model for the rule to be applicable.

Model Transformation

A model transformation is an operation $S_{OUT} = T(S_{IN})$ that takes a set of input models S_{IN}, executes a set of transformation rules S_R over the model elements and produces a set of models S_{OUT} as output.

A transformation is itself also a model and therefore all the general operations applicable to models may be applied to transformations (including transformations that are called higher-order transformations).

Compose operation

The compose operation $M_{AB} = Compose(M_A, M_B, C_{AB})$ takes two models M_A and M_B and a correspondence model C_AB between them and combines their elements into a new output model M_{AB}.

This operation is considered in this paper as a general and even abstract operation that all the other composition operations must specialize or instantiate. For this operation and its specializations, it is common that the models involved in the process conform to the same metamodel. If this is not the case, an interface between the (meta)models would be required.

Merge operation

The merge operation $M_{AB} = Merge(M_A, M_B, C_{AB})$ takes the same parameters as the more general compose operation. It produces a different output M_{AB} which includes all the elements from M_A and M_B.

DSL

A DSL is formally defined as a five-tuple containing the elements:

1. Concrete syntax C defines the notation used to express models which may be graphical, textual or mixed.

2. Abstract syntax A defines the concepts, relationships and constraints or well-formedness rules of the language.

3. Semantic domain S is a set of concepts (usually defined in some formal framework) in terms of which the meaning of the models of the language are explained.

4. Syntactic mapping $M_C : C \rightarrow A$ assigns a syntactic construct of the concrete syntax to the elements of the abstract syntax.

5. Semantic mapping $M_S : A \rightarrow S$ relates the syntactic elements of the abstract syntax to those of the semantic domain.

Aspect Oriented modelling

Aspect oriented modelling (AOM) translates the concepts and ideas applied at the code level by aspect oriented programming (AOP) [15] to the model level. Usually, a system contains one or more so called cross-cutting concerns spread along the system (e.g. Logging or Debugging). These cross-cutting concerns are difficult to maintain when tangled all over the system. AOM provides a way to isolate these concerns into centralized and easy to maintain models which are called *aspect* models. The way to keep these aspects where they should be located in the system is by defining two core concepts inside them: a *pointcut* and an *advice*. The pointcut defines all the joinpoints where the cross-cutting concern is located in the system. And the advice defines what functionality, either a code segment in AOP or model instances in AOM, to insert at the join points. The act of combining aspect models into a base model is called *weaving*. This terminology can be used in different contexts or implementations, for example when using rules for weaving, a rule can be considered as the aspect, where the LHS is the pointcut, the RHS is the advice, and the match is the joinpoint.

3 Model weaving techniques

In this section a review is presented of the current techniques for (meta)model composition where a special attention is taken into how they can be applied in the language weaving context. That is, for all of the DSL components: abstract syntax, concrete syntax and semantics. For the purpose of this paper, we will therefore give emphasis to the techniques for metamodel composition because they are more commonly used to represent the abstract syntax of a language. Even though, since metamodels are models themselves, model composition techniques can also be employed.

3.1 Merge

Model merging is a composition technique that does not assume an unbalanced combination, it is performed on two *peer* models. In terms of set theory it can be defined as the duplicate free union of both sets (in this case metamodels).

3.1.1 Merging based on correspondences by Pottinger

In [30], Pottinger and Bernstein propose an approach to model merging that depends on a set of user-defined correspondences between the metamodels. The authors provide a *generic* framework that can be used to merge all types of models.

Their approach requires first identifying which elements of each metamodel will be combined together. In other words, which elements of each of the metamodels should be merged together to form one in the output metamodel (being that it is a duplicate free operation). This information is then stored in the mentioned set of correspondences between them. The preliminary process of finding this set is called matching (or schema matching in databases). To put it in aspect oriented terminology, the correspondences can be considered as the join points between the two metamodels, although in this case, the roles of base and aspect model are not as clear as in AOM since merging does not assume an unbalanced combination. The elements which are not included in the set of correspondences are simply added to the result model without modifying them (being that it is a union operation).

With the mapping obtained in the matching step, the merge step performs the actual composition. This composition is not as simple as the set union since the detection of duplicates (which depends on what we define as a duplicate, its semantics) as well as their removal can be complex. In addition, the resulting merged model can also present constraint violations, or *conflicts*, that the Merge operation must resolve.

The authors provide the following categorization of the conflicts that can arise, based on the meta-level where they occur:

- Representation conflicts occur at the model level and are caused by conflicting representations of the same modelled (real world) concept. An example is having a model A which represents the concept of name by one element *Name* while model B represents it by two elements: *FirstName* and *LastName*. How the concept should be represented in the final model is a decision that is application dependent and is performed before the merging algorithm.

- Meta-model conflicts, are caused by the violation of the constraints of one of the two metamodels involved in the merging.

- Fundamental conflicts are caused by violations of constraints at the metametamodel level, the representation to which all models must conform to. In other words when the result of a Merge would not be a model due to violations of the metametamodel.

In Figure 1, an example can be seen of a general merge operation on two metamodels depicting different facets of a Role Playing Game (RPG), both defined in the UML class diagrams metametamodel. This simple example contains two different join points or correspondences which are located between the classes Tile and Hero of the same name in both metamodels. The remaining classes Treasure and Weapon are not included in the correspondences and thus they are both included in the final metamodel AB (along with their corresponding associations). Finally, the Tile class of the resultant model AB, an unresolved conflict can be observed. This conflict is caused by the fact that both input metamodels have two different ways of representing the concept of location of a Tile. Model A does this with the x and y coordinates while model B with the neighboring tiles of the four cardinal directions. This conflict is not properly solved in this example since both concepts have been included in the final model and its solution would require a decision by a user.

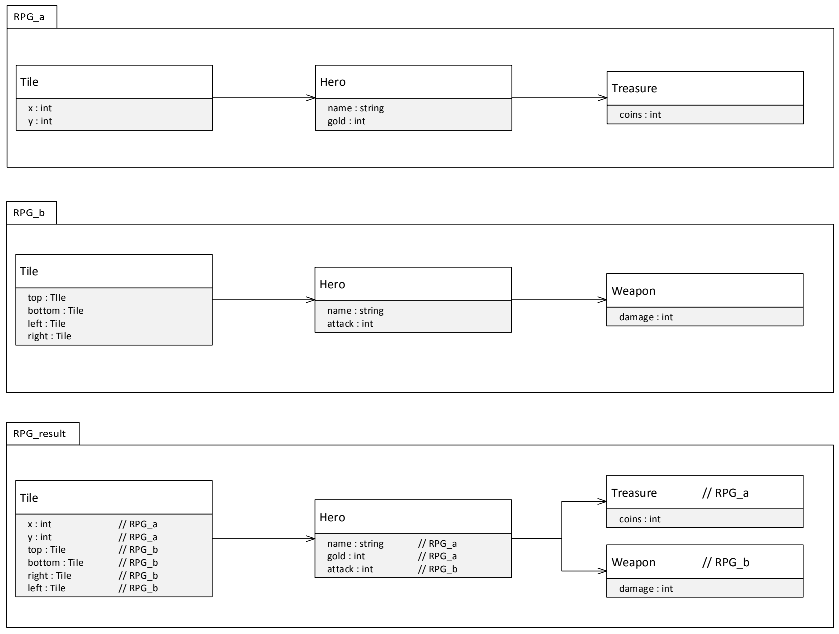

Figure 1: Merge operation of two class diagrams depicting two different meta-models of an RPG

3.1.2 Package Merge in UML 2

UML 2 [36] defines Package Merge as an operation for merging the contents of two packages together. The elements of the models or metamodels contained in the packages are merged if they share the same name and signature. Package merge aims at allowing the definition of metamodels in UML more modular. It is a directed relationship between two packages which indicates that the contents of the target package are merged into the contents of the source package. The composition occurs in two phases which apply a set of constraints and transformations [6]. First, the constraints are used to check if two elements match. When two elements match, the transformations take care of the actual merging. Elements that do not have a matching counterpart are simply carried over to the result model. Constraints and transformations are expressed declaratively through match rules and transformation rules. These rules are pre-defined for each metamodel type (class diagrams, sequence diagrams, etc) of the UML family of metamodels.

An important feature that is mentioned in the UML 2 specification and that is very important for all composition approaches is that a resulting element will not be any less capable than it was prior to the merge, in other words that the merge operation does not violate Liskov's substitutability principle. In this case it means that the resulting navigability, multiplicity, visibility, etc will not be reduced as a result of a package merge.

According to Vallecillo [37], the problem with this approach is that its current definition is neither precise nor sound and it does not consider possible conflicts between the structural constraints of the metamodels merged . As a result, it may break the well-formed rules (constraints) of the models that it combines. Furthermore, no traceability links are created between the resultant metamodel and the input models and finally it works exclusively for models defined with the UML metamodel.

3.1.3 Metamodel merge in GME

In [21], Lédeczi et al. propose an extension to UML that allows metamodel composition from existing metamodels. This extension consists in the addition of three new UML operators for using in metamodel combination:

- The *equivalence* operator is used to denote a full union between two UML class objects. The union includes all attributes and associations, including generalization, specialization, and containment, of each individual class. It can be thought of defining the join points for the source metamodels, similarly to the correspondences definition combined with the merging steps of Pottinger's approach (see section 3.1.1).

- With the *implementation inheritance* operator the child class inherits all of the parent class's attributes, but no associations except the containment ones where the parent functions as the container.

- While the *interface inheritance* operator allows no attribute but does allow full association inheritance, with the exception of containment associations where the parent functions as the container.

The union of the implementation and interface inheritance is the normal UML inheritance and their intersection is null. The three operators mentioned are simply a notational convenience or syntactic sugar since each of them has an equivalent “pure” UML representation. These equivalent representations, however, would make a diagram significantly more cluttered and difficult to understand.

The merging process of this approach is very similar to Package Merge except that the operation takes place at the *class* instead of the package level. The two classes being merged do not need to have the same name because of the use of the equivalence operator which sets up the correspondence between the two classes to be merged. This approach deals exclusively with UML class diagrams (metamodels) and is therefore even more specific than Package Merge which can be applied to all types of UML diagrams.

The authors mention that it is important that the composition leaves the original metamodels intact, so that they can still be used independently. Also, the newly composed metamodel should be capable of instantiating existing models created using the original metamodels (backwards compatibility).

An implementation of these operators has been added by the authors to the Generic modelling Environment (GME), a DSL design environment, developed in the Vanderbilt University.

3.2 Extension

Metamodel Extension consists in extending one model, called *pivot* or *initial* with the concepts of another, the *extension.* The new extension model includes concepts that were not originally present, but some of them may reference existing ones. An example is the case of designing a hierarchy of models or to extend, in a modular way, a model with new features.

3.2.1 Model extension by Barbero et al.

In [2], Barbero et al. propose a new kind of relation *extension* between two models along with the two relationships defined by the principles of MDE: *conformance* and *representation.* The conformance relation links one model to it’s metamodel or reference model, and the representation relation links terminal models (instances of models) to the systems they represent.

The additional relation called *extensionOf* links an initial model M_i, that represents most of the concepts of a system, with an extension model M_e, that defines some new concepts not present in M_i but that may reference existing concepts in it. The two models M_i and M_e must conform to the same reference model in this approach, which classifies it as a *Homogeneous* composition (see section 4.2)

The authors provide a conceptual framework for the operation as well as an implementation using the KM3 text-based meta-modelling language. Furthermore, the actual composition is similar to that of the Merge technique explained in section 3.1 since it also performs a duplicate-free union of the models. The composition is preceded by a user-defined specification ϵ of the correspondences between the elements to be composed.

This technique is more effective when the relationship between the two models to combine is *asymmetric*, i.e. the extending model complements the initial model, and *conservative*, which means that the extensions must not break the semantics of the initial model (since the approach does not support any kind of conflict resolution strategy).

On Figure 2, a simple example of an extend operation is shown. Two metamodels (that conform to the UML class diagram metametamodel) depict basic concepts of a RPG application. The initial model A describes the Tile, Hero and Treasure concepts while the extension model B specifies three new types of Tile: Obstacle, Trap and Door. Note that in the extension model B a reference to an element of another model is made, namely Tile from the initial model A. The correspondence in this case links the two Tile classes of both metamodels and this represents the join point where the extension takes place.

3.3 Templates

Templates are in essence an extension mechanism (section 3.2) with an extra feature, i.e. they provide extensibility for models which have not yet been specified. A template provides reusability by allowing the creation of general models that can later be further specified by *instantiating* pre-defined points called template *parameters*. Furthermore, it gives modularity because it is a kind of extension technique which in turn is a composition that provides this benefit as it was explained in section 1.3.

In Figure 3, a simple example of the template technique is shown. Two metamodels (that conform to the UML class diagram metametamodel) depict basic concepts of the same RPG application as in the previous examples. The template model *RPG_template* contains the Location, Character and Item classes while the extension model *RPG_tiles* specifies three new types of Tile: Obstacle, Trap and Door. The special feature in this technique is the symbol "|" before the Location class. This means that the Location class is a template parameter that has to be instantiated by a concrete element of another model (in this case a class since it is a homogeneous composition technique). In the *RPG_tiles* model, the Tile class instantiates or binds to the template parameter |Location.

3.3.1 Template Instantiation by Emerson and Sztipanovits

In [7], Emerson and Sztipanovits propose the use of template instantiation to overcome the limitations of techniques such as model merge and interfacing. The problem with these techniques, according to the authors, is that they are not well suited for the *multiple* reuse of metamodel fragments into the same

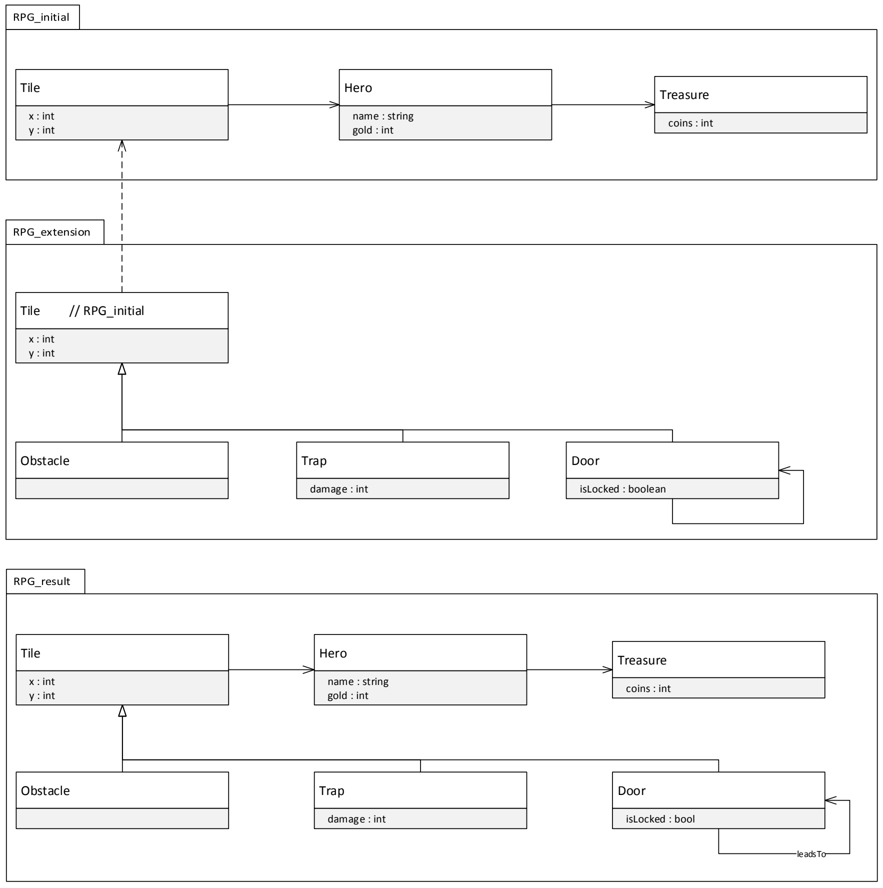

Figure 2: Extension operation of two class diagrams defining two RPG meta-models

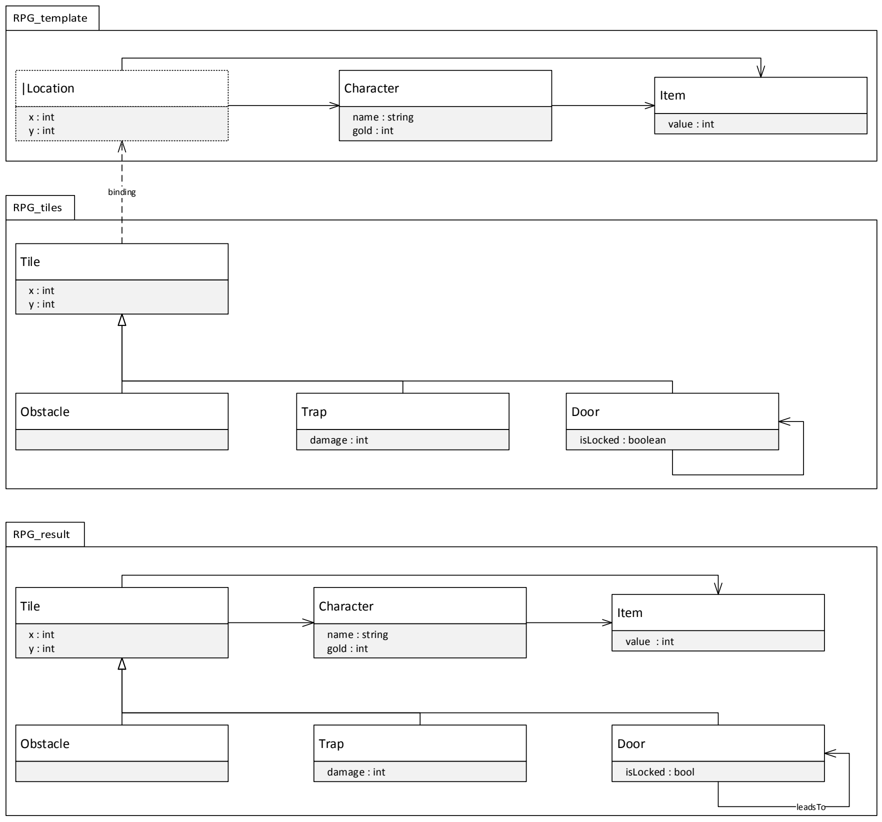

Figure 3: Example of the Template technique in the context of RPG.

composite metamodel. In other words, when performing a chain of consecutive compositions into one same metamodel, the weaving of a composition can be affected by the changes made when weaving a previous one.

Template instantiation, on the other hand, automatically creates new relationships between the pre-existing elements in a target metamodel with the template parameters of a common meta-modelling pattern. These common meta-modelling patterns are created from a set of templates of several metamodels that are commonly-occurring, e.g. State Charts, Data Flow graphs, Hierarchy, etc.

The authors provide a simple prototype for template instantiation in the GME meta-modelling language which guides users through the selection of the template to be used and the assignment of the template parameters or roles to domain specific concepts. Then, it automatically edits the domain-specific metamodel in order instantiate the template.

This composition technique can be applied to a broader spectrum of metamodels, the ones for which common metamodel patterns templates are provided and it is therefore heterogeneous. This means that it can be useful for abstract, concrete syntax as well as for the semantics of a language.

3.3.2 Templates in MetaDepth

MetaDepth [5] is a textual meta-modelling environment tool that allows deep meta-modelling, in the sense that it supports an arbitrary number of meta-levels. This tool also has mechanisms that allow the definition of the three components of a language: abstract, concrete syntax and semantics.

Furthermore, MetaDepth the contains following composition mechanisms:

1. *Extension* at the class/object level: inheritance. At the model level, an extension is a metamodel fragment that adds new elements to the base metamodel.

2. *Concepts* have the form of a (meta)model, however their elements are variables. These variables need to be *bound* to concrete elements of a (meta)model.

   Concepts can be used to express composition requirements, where some elements of a metamodel fragment are variables and a concept defines the requirements for binding those variables to the elements of another metamodel fragment.

   Also, operations can be defined over a concept so that they become reusable by any metamodel that fulfills the concept's requirements (in other words, that is bound to it). This allows reusability of those operations by being defined in a generic way with concepts.

   Hybrid concepts require a binding as well as an implementation of some of it's operations.

3. *Templates* enable a more flexible definition of model and metamodel fragments, as they permit their connection. A metamodel template is a

metamodel where some elements (meta-classes, features, associations) are variables. The connection requirements for such variables are expressed through a concept.

In [24], Meyers et al. shows how to compose modelling languages by using these composition mechanisms. They perform composition on homogeneous metamodels due to the fact that all models have been defined with MetaDepth and thus conform to the same meta-model.

It is worth mentioning that this is one of the few approaches that composes all three components of a language, including the often neglected concrete syntax. The disadvantages of the approach are that it deals exclusively with textual languages defined in Meta-Depth (it is not a *generic* approach). Although it is possible to create an extra step in the process where a model defined in any modelling language is transformed into a model that conforms to the MetaDepth metamodel.

For future work, the authors mention they plan the addition of more advanced mechanisms for composition that allow bidirectional binding of two templates or a more flexible binding.

3.4 Parametrization by Pedro et al.

In [27, 29], Pedro et al. propose an approach for DSL prototyping in a compositional and incremental way.

The fundamental notion of the approach is the *domain concept* which is defined as a metamodel that has attached to it a transformation to a target language, that is precise and provides its semantics. Semantics are defined with a set of transformations that translate certain elements of the source model (domain concept's metamodel) into elements of a target model. Consequently, a domain concept contains two main elements of a DSL: abstract syntax and semantics, the missing third one being concrete syntax (tackled by Pedro in the later paper [28]). Furthermore, a domain concept represents a basic idea that can act as a building brick in one or several DSLs. Finally, the authors claim that other approaches have only focused on **syntactic** composition.

Parametrization is the action of enriching a metamodel by means of another metamodel. A formal parameter f_p defines a template of what can be replaced in mm – the metamodel being enriched. The effective parameter e_p is a metamodel that replaces f_p after the parametrization. Finally the replacement has to respect a set of constraints on the f_p. This way a metamodel can be extended by parametrization

The paper also defines a set of composition operators that work at a more syntactic level than the parameterization. Nevertheless, depending on the operator used, transformations are adapted to cope with operator's semantics.

In order to compose DSLs or domain concepts semantically it is necessary to compose their corresponding transformations. This is performed for the two previously explained techniques: parametrization and composition. The parametrization is done similarly to that of metamodels (abstract syntax) by providing a formal parameter which is the transformation template.

Finally in [28], the authors extend the DSL prototyping approach to also support the composition of concrete syntax in DSLs.

3.5 Interfacing

In [7], Emerson and Sztipanovits describe briefly the concept of metamodel *interfacing*. The idea is to define an interface between two modelling languages consisting of elements that do not strictly belong to either of them in order to allow their composition. The description given by the authors is very broad and does not give further details. However the idea could be applied to the composition of heterogeneous models in a similar way as the semantic adaptation approach (see section 3.8). Although in the case of interfacing this could be done at a higher abstraction level, that is at the metamodel level instead of the model or instance level.

3.6 Embedding

Language *embedding* is an alternative approach to building DSLs from scratch. It consists of making a guest (embedded) language inherit the infrastructure of a host (global) language. This way, the guest language can reuse the syntax, module system, existing libraries, and tools of the host language. This embedding is defined by mapping the guest language concepts into host language concepts. This mapping is not explicitly defined anywhere and there is no explicit trace between the two languages, therefore losing the connection between the concrete syntax and the tools of the guest language.

UML is an example of a suitable host language since it is very expressive and well-known. It was in fact originally created by embedding three other languages into it [37].

3.7 Refinement

Similar to *embedding*, in *refinement*, elements of a guest model are hierarchically contained within a single construct of a host model. In this way a concept of the guest model can exist in the host only as a black box, providing modularity to the system.

This technique is briefly described in [7] although no implementation or example of it is provided.

The concepts of co-variance and contra-variance apply in this case of hierarchical containment. By which, in order to conform to the Liskov's substitutability principle (section 1.3.4), the domain of the guest DSL must be larger or equal than the host DSL's. And the image of the guest DSL smaller or equal than the host DSL's.

3.8 Semantic Adaptation

The concept of semantic adaptation was proposed in the approaches Ptolemy and Modhel'X [11]. This technique is geared towards the composition of models that represent the semantics of another model of a language. Semantic adaptation defines a set of laws for composing models with *heterogeneous* semantics and obtaining a meaningful result from it. This is done by defining interface blocks between the models to be composed, similarly to the interface approach for composition (see section 3.5) but at a lower level of abstraction since it deals directly with the inputs and outputs of the model. These interface blocks adapt the data, control and time between the two different model. Semantic adaptation is the "glue" that is necessary to compose heterogeneous models so that the resulting model has well-defined semantics.

In [25], Meyers et al. propose a DSL to aid in defining the interface blocks by effectively bridging the cognitive gap between the implementation and specification (modelling) of an interface block. This DSL supports the explicit modelling of the adaptation of data, time and control which is performed in an interface block. The approach enables the modeler to easily define the interfaces in a modular way since the involved models are left untouched. In our classification scheme of section 4, this feature fits into the category of low *intrusion*, because of the absence of modifications to the original models. This implementation provides the benefits of any DSM solution: constraining the modelling process to domain-specific concepts, separating the modeler from platform-specific issues by performing this translation automatically with a code generator.

3.9 Aspect Oriented modelling

The composition approaches reviewed in this section have in common that they are all applied in aspect oriented modelling (AOM) approaches. This means that they focus mostly on an asymmetric kind of composition where at least one aspect model, representing a cross-cutting concern, is composed into a base model. In [13], Jézéquel separates the process of weaving models in AOM into the following two main steps:

1. In the *detection* step the join points are determined around which the composition is performed.

2. And in the *composition* step the advice of the aspect is composed into the base model.

He also explains that weaving more than one aspect at the same join point can cause difficulties because previously woven aspects may have modified the joinpoint. In model composition in general, this can also be identified as a difficult problem.

The next subsections describe several AOM approaches and it focus on their model (or aspect) composition problem.

3.9.1 GeKo by Morin et al.

GeKo is defined as a generic aspect oriented weaver. It is generic in the sense that it can easily be adapted to any DSL with no need to modify the domain metamodel or to generate domain-specific frameworks. It relies on the definition of mappings between the pointcut and the advice elements, which are in turn defined in terms of the concrete syntax of the models by linking them with generic links that do not use any domain-specific knowledge. GeKo keeps a graphical representation of the weaving between an aspect model and the base model. It is a tool-supported approach with a clear semantics of the different operators used to define the weaving.

In order to describe point cuts more easily, the authors propose to construct an, on demand, more flexible metamodel from the original one. This more suitable metamodel does not contain any pre or post conditions or invariants, all features of the meta-classes (e.g. attributes in class diagram) are optional and it does not contain any abstract classes. This is similar to the of *RAMification* process described in [35] (not to be confused with the approach RAM of section 3.9.2).

The main idea of GeKo is the definition of five sets to which all the elements of the base, advice and pointcut models must be assigned. These sets are partitioned by means of two morphisms which allow the identification of the elements to be kept, removed and replaced. After all the sets are complete, the weaving is done in a straightforward way.

3.9.2 RAM Weaver by Kienzle et al.

The RAM Weaver is part of the Reusable Aspect Models (RAM) approach [16] [18]. RAM focuses on aspect oriented modelling with emphasis on scalable multi-view modelling.

The RAM Weaver currently supports the weaving of two different kinds of models or formalisms which is performed by the following techniques:

- Structural models, defined as class diagrams and composed with the symmetric model composition technique proposed by France et al. [32]. This technique supports merging of model elements that present different views of the same concept but they must both be instances of the same metamodel. The merging is done via name as well as attribute and operations matching of classes.
- Behavioral models, defined in sequence diagrams use the technique semantic based weaving of scenarios proposed by Klein et al. [17] where an aspect is defined as a pair of sequence diagrams: one for the pointcut (behavior to detect) and one for the advice (the behavior to add to the model).

To keep the aspect models as generic as possible, model instantiation is used via UML templates with template parameters. To instantiate a model, one then has to bind all the aspect's template parameters to target model-specific elements. The resulting model is context specific and can therefore be composed with a target model.

TouchRAM [1] is a multitouch tool for software design based on the RAM approach. It currently supports only the structural model weaving technique explained above.

3.9.3 MATA by Whittle et al.

MATA is an aspect-oriented modelling tool presented by Whittle et al. [38]. It considers aspect composition as a special case of model transformation. Composition of a base and aspect model is specified by a *graph rule*. The LHS part of the graph rule corresponds to the pointcut and the RHS to the advice of the aspect to be merged. MATA rules are defined over the concrete syntax of the modelling language in contrast to most approaches which do it at the meta-level (over the abstract syntax). This difference is important because no knowledge of the metamodel is needed in order to design graph rules.

MATA currently supports composition of UML class, sequence and state diagrams although it is possible and relatively simple to extend it to other modelling languages for which a metamodel exists. MATA performs some minor extensions to the concrete syntax of UML models, namely the addition of three stereotypes. The $\ll$ *create* $\gg$ and $\ll$ *delete* $\gg$ state that an element should be created or removed by a graph rule, respectively. The $\ll$ *context* $\gg$ stereotype states that a container element's are not affected by $\ll$ *create* $\gg$ or $\ll$ *delete* $\gg$.

MATA also provides some support for automatically detecting interactions between aspects. This is done with the technique of *critical pair analysis* (CPA) which can be used to detect dependencies and conflicts between graph rules. To solve a conflict, the order in which the rules take place should be changed or, in some cases, the rules themselves should be modified. CPA provides feedback for these cases and an assurance that a composition is correct.

It is worth noting that MATA does not define join points explicitly since composition is viewed as a special case of model transformation. This avoids the limitation of the approach to specific types of diagrams or models.

3.9.4 Generic Model Composition Framework by Fleurey et al.

In [9] a generic framework for automatic model composition is proposed. It determines two steps for composition: matching and merging.

The *matching* step is specific to a modelling language and it determines which and how two elements match. Precise guidelines are offered in the framework for the definition of the matching operator specific to a particular language and giving it composition capabilities. The *merging* step is independent (generic) from any language and therefore the behavior of the merging operator is already implemented by the generic framework. Since the composition is performed with a merging approach, the *modularity* of this technique is classified as *low* as in the other merge-based techniques of section 3.1. A class diagram of the framework is described in Figure 4.

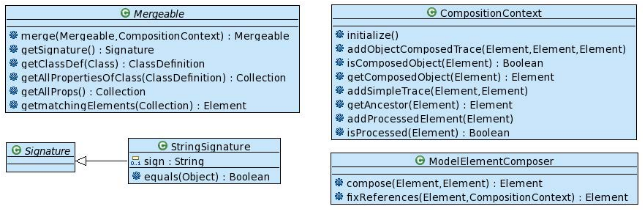
Mergeable
merge(Mergeable,CompositionContext) : Mergeable
getSignature() : Signature
getClassDef(Class) : ClassDefinition
getAllPropertiesOfClass(ClassDefinition) : Collection
getAllProps() : Collection
getmatchingElements(Collection) : Element
CompositionContext
initialize()
addObjectComposedTrace(Element,Element,Element)
isComposedObject(Element) : Boolean
getComposedObject(Element) : Element
addSimpleTrace(Element,Element)
getAncestor(Element) : Element
addProcessedElement(Element)
isProcessed(Element) : Boolean
Signature
StringSignature
sign : String
equals(Object) : Boolean
ModelElementComposer
compose(Element,Element) : Element
fixReferences(Element,CompositionContext) : Element

Figure 4: Fleurey's Generic framework for Composition [9]

Lastly, for potential composition conflicts detection and resolution, a generic composition directive language is included, based on the model composition directives defined by Reddy et al. [32]. Both the generic framework for composition and a specialization for the Ecore metamodel have been implemented in Kermeta and are available as the open-source tool *Kompose* [8]

The main limitation of this approach, as mentioned by the authors, is that the composition is focused heavily on the structure of models. This becomes a difficulty when working with behavioral languages such as sequence diagrams or state-charts where the merging algorithm would have to be redefined to allow composition of semantics.

3.9.5 Aspect Oriented Architecture Models by France et al.

In [32, 10], France et al. propose the Aspect Oriented Architecture Models (AOAM) approach. This approach includes, along the usual AOM capabilities, a signature-based composition technique and a set of *composition directives* for making the composition more flexible.

Currently, the approach only supports composition of class diagrams. The composition is specified with a provided composition meta model that can be seen in Figure 5. Elements of the models that can be composed have to realize the Mergeable meta-class and therefore implement the match-by-signature operation *sigEquals*. The composition phase applies a template technique by first requiring the template parameters of the aspect models to be bound to concrete modelling elements of the base model. After this binding, a merge is used with a signature-based approach. The original approach used a name-based merging method but in [32], this mechanism was extended to a signature-based method that allows matching according to syntactic properties like attributes or operations. For similar reasons like in the GMCP approach (section 3.9.4), the *modularity* for this technique is classified as *low* as well.

The composition directives are intended to refine the composition rules used to compose models by allowing the altering of model elements, specifying the order in which aspect models are composed, or overriding of the default composition rules. Directives such as "precede" and "follow" are present in the model as stereotyped UML dependencies between aspect models and represent a conflict resolution mechanism.

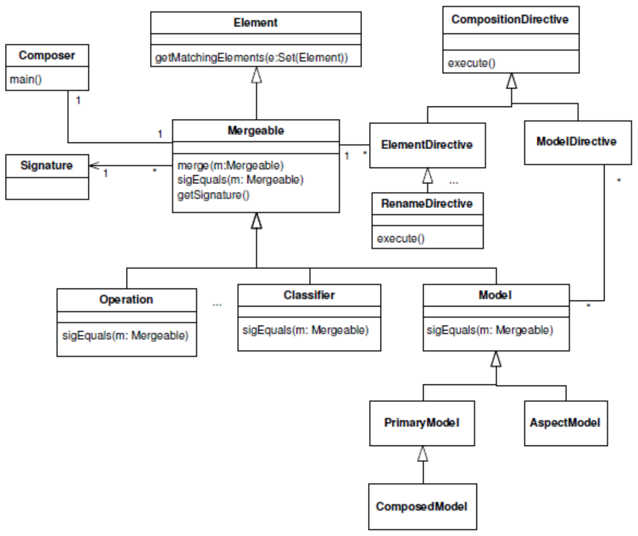

Figure 5: Core elements of the composition metamodel of France et al. [31]

3.9.6 Motorola WEAVR by Cottenier et al.

The Motorola WEAVR approach [4] is one of the few AOM tools that have been developed in an industrial setting. Its composition domain consists of models defined in UML or in the Specification and Description Language (SDL), a formalism for representing behavior similar to state and activity diagrams. In order to be able to reuse aspects, mappings have to be defined that link a reusable aspect to the application-specific context in which it is to be deployed. Furthermore, the approach also supports composition asymmetry, i.e. aspects can be woven into the base but not the other way round and the weaving is done statically at design time right before code generation. Finally, this technique supports model execution and code generation.

3.9.7 Semantic Weaving by Klein et al.

The behavioral aspect weaving approach of Klein et al. [17] focuses on weaving at the semantic level instead of syntactic like most other approaches. It is based on the Message Sequence Charts (MSC) language that is similar and can be adapted to sequence diagrams. The composition is specified at the modelling level so it is therefore independent of any implementation platform. It follows an asymmetric approach for composition since it is aspect oriented and it can only weave aspect model(s) into a base model. Both the pointcut and the advice models are modelled visually with sequence diagrams, an example can be seen in Figure 6.

Concerning the composition process itself, the approach presents a series of algorithms for the matching or detection phase. First, an algorithm unfolds the

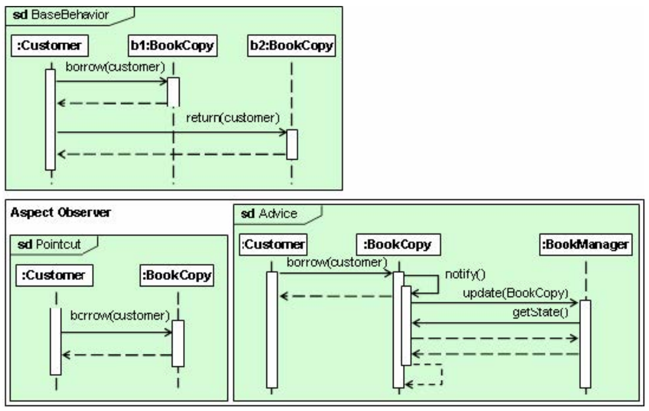

Figure 6: An example of the Semantic Weaving approach of Klein et al. [33]

base model to exhibit all potential matches of the pointcut. Next, one computes the parts of the potential matches that are actually matched by the pointcut. Then, an algorithm resolves all the loops in the model and lastly, a set of minimal acyclic paths where the pointcut matches is built. The weaving process ends with the composition step where all the acyclic paths that represent the final pointcuts in the base model are replaced by the advice of the aspect.

Finally, the authors mention several limitations to their approach. One limitation is that the matching process can only be performed if each join point appears inside a bounded fragment of behavior because otherwise the algorithm for finding potential matches would never terminate. Another limitation is that the base model should not contain two non-disjoint cycles where the pointcut matches that can only be checked during the weaving process since it is not a structural property of the model.

An implementation of the approach has been added to the UMLAUT model transformation framework [12].

3.9.8 Survey by Schauerhuber et al. [33]

Schauerhuber et al. perform a survey on eight representative UML-based AOM approaches. A selection of those approaches among others have been included in this section. The survey analyses the approaches with respect to many criteria grouped in categories. Some interesting conclusions pertaining to the composition process are summarized next:

- Composition is often deferred to implementation.
- Predominance of matching with name.
- Merge used as a default integration (composition) strategy.
- Missing tool support for composition and code generation.

- Missing guidance on when to use asymmetric vs. symmetric approaches.

3.10 Viewpoint Unification by Vallecillo

A technique for DSL combination is proposed by Vallecillo [37] called Viewpoint Unification. The technique only supports the combination of the metamodels of a language, i.e. its abstract syntax. It is based on the consideration that each language to combine provides a *viewpoint language* to describe the same system. The way of combining or unifying the languages is by providing a new language and a set of mappings between it and the viewpoint languages, with the condition that the mappings respect the constraints of the viewpoint languages correspondences.

The composition itself depends on the languages to combine and the authors recommend the user to *implement* one of the appropriate composition from merge, extension or embedding. Therefore this approach does not really provide a technique for composition but it is more focused on the step before it: the detection or matching. In this step the elements to compose are linked together with mappings. The matching step is similar to that of the Templates approach in the sense that elements of the input models are projected (or bound to) elements of the result model.

3.11 Semantic Anchoring by Chen et al.

Lastly, we present an approach that aims to precisely define semantics, which is a requirement to allow their composition. The goal of this approach is to provide a formal specification to the semantics of a DSL by 'anchoring' them to a metamodel. For this purpose, it first defines a set of canonical DSLs capturing fundamental types of behavior (by defining their operational semantics), called Models of Computations (MoCs). This DSLs, called in the paper *semantic units*, have to be minimal; in the sense that they are the simplest modelling languages required to describe a selected type of behavior. Semantic anchoring is then defined as the mapping between an arbitrary language L to a canonical MoC L_i.

Semantic units are specified in the Abstract State Machine (ASM) formal framework that allows to represent the three components of a semantic unit: abstract syntax, semantic domain and the mapping between them. To specify the mapping from the abstract syntax of the DSL and the abstract syntax of the modelling language used as a semantic unit, model transformation techniques are used. This is performed using the GReAT tool suite for model transformations.

The work presented in this paper provides a way to add semantics to DSLs but it does not define how to combine this semantic blocks together. The transformations from a DSL to the more precise domain of a semantic unit are not possible to compose either.

4 Classification Criteria

In this section a set of classification criteria is presented in order to perform a comparison between the composition techniques presented in the previous section. Special attention is paid to the benefits that each criterion can bring to the problem at hand: language composition. As already mentioned, language composition can be divided into the three separate compositions of its components: abstract, concrete syntax and semantics. Therefore it will be mentioned if a criterion is more relevant to one of these three compositions.

4.1 Symmetry

The symmetry of a composition refers to the relationship between the models to be composed. This does not necessarily mean that the models have this relationship in all contexts, but it only means that they fulfill these roles during the composition [33]. A composition can be either *asymmetric* or *symmetric*:

In an asymmetric composition, one model is designated as a *base* model and one as an *aspect* model. The role of base models is usually taken by those that can be used independently and autonomously while aspect models are usually only usable after being weaved into a base model. Aspect models are generally not of much use on their own. Examples of aspect models are debugging or security. Most *AOM* approaches (section 3.9) perform asymmetrical composition.

On the other hand, in a symmetric composition there is no designated base model. The models to be composed do not play clear roles as in asymmetric composition and can be considered as peer models. An example of symmetric composition is the *Merge* technique of section 3.1.

For the case of language composition, both types of symmetry are relevant, since the components of a language usually fill in the same roles (in terms of symmetry) as that of the language itself, and two languages can have any kind of symmetry relationship with each other.

Possible values for this criterion are: **asymmetric**, **symmetric** or **both**.

4.2 Uniformity

Uniformity refers to the nature of the models to be composed. More specifically, if they conform to the same meta-model or not. A composition technique can support *homogeneous* or *heterogeneous* composition.

The composition of two models is called *homogeneous* or uniform if they conform to the same metamodel. The abstract syntax of languages is usually defined by a structural metamodel like class diagrams and can therefore be combined with a homogeneous composition. Most of the techniques explained in the previous section fit into this category.

When the models conform to a different metamodel the composition is *heterogeneous*. This case poses more difficulties than its counterpart since the two different models have different semantics and notations. Usually this kind of composition requires an interface between the models to act as a "glue" and translate the elements of one model into the other. Techniques that focus on this type of composition are Interfacing and Adaptation.

In terms of language weaving, abstract syntax composition can generally be considered homogeneous since it is usually specified by a structural metamodel like UML class diagrams. Weaving instances of a language would also be homogeneous since both instances would conform to the metamodel of the language. On the other hand, the specification of semantics does not have such a standard approach and their composition is therefore heterogeneous.

Possible values for this criterion are: **homogeneous** or **heterogeneous** (which we see as equivalent to saying **both**).

4.3 Domain

This criterion specifies the domain of models that the approach can be applied to. Some approaches only work for certain types or families of models, e.g. UML diagrams, and are therefore specific. If an approach can be conceptually applied to any type of model, (assuming its metamodel has been formally defined) it is generic. Approaches that consist of frameworks for composition and require an extension to allow the composition of a certain type of model, are also considered as generic.

Possible values for the domain are: **Generic** or **Specific**, where the latter is further specified with two distinctions: **CDs** (specific to class diagrams) or **UML** (specific to UML models, possibly including CD if they are more than one). Further information on the domain is provided by the criterion described in section 4.4.

4.4 Nature of models

This criterion describes whether the domain (section 4.3) of the approach includes *structural* and/or *behavioral* models.

Structural models define the elements that must be present in a system and their relationships. They generally do not contain behavior.

Behavioral models emphasize what occurs in the system being modelled and they have a notion of execution, which inherently has a notion of time.

Possible values are: **Structural**, **Behavioral** or **Both**.

4.5 Time

In the case of behavioral models, since structural models do not have a notion of time, this criterion distinguishes between *discrete* time, a simplification of what happens in the real world, *continuous* time, an exact representation. Structural models can also be considered to have a notion of time if this one is a singleton with only the value *now*, which never changes.

As with heterogeneous models, models with different types of time notions require a translation or adaptation between them before they can be weaved.

Possible values are: **Discrete**, **Continuous** or the symbol "−" if it is not applicable (because the technique does not deal with behavioral languages in the first place).

4.6 DSL Weaving applicability

This criterion specifies for which of the three parts of DSL weaving the approach at hand is better suited. In other words, if an approach can be applied for the composition of abstract syntax (in the form of models like class diagrams) and/or semantics (e.g. in the form of transformations). Note that this criterion does not include concrete syntax since this is measured by the criterion of section 4.7.

Possible values are: **Syntax**, **Semantics**, **Both**.

4.7 Concrete syntax support

The concrete syntax is an important part of a language that is often neglected in composition techniques. In this criterion, the support for the composition of concrete syntax models of an approach is measured. For the case that a technique does support concrete syntax composition, the criterion specifies if it is either textual or graphical. Graphical concrete syntax includes textual as well.

Possible values are: **None**, **Textual**, **Graphical**.

4.8 Implementation

This criterion simply describes the stage of implementation of the approach (at the time of publication).

Possible values are: **None** (e.g. in a position paper), **Partial** (for approaches that provide a prototype of their technique) or **Complete** (for fully implemented techniques).

4.9 Moment of composition

Composition can generally happen at two moments: at *design* or *run* time. We call a composition at design time static or dynamic if it happens at run-time. The former is analogous to composition at compile time in the programming world and the former to that of the same name. Dynamic composition needs access to the semantic domains of the models since it occurs while executing the models. An example of dynamic composition is a co-simulation where the traces of the simulations of two behavioral models are weaved together at run-time.

Possible values for this criterion are: **Static** or **Dynamic**.

4.10 Matching Method

This criterion describes how the matching method of an approach is defined. In other words how the join points between the models to be composed are specified. The matching method can be either automatic, if it is done without user intervention, or manual when user input is required to specify the elements that match with each other.

Possible values for this criterion are: **Automatic** or **Manual**.

4.11 Conflict Resolution

This criterion describes the strategy of an approach to resolve conflicts caused or by a composition. It is possible that an approach has no conflict resolution strategy. A distinction is made between a conflict avoidance strategy, which takes place before a composition, and conflict resolution, which happens post-composition.

Possible values are: **Avoidance**, **Resolution**, **None**

4.12 Modularity

The modularity of a composition technique refers to how explicitly regulated the access to the two models to be composed is. This has consequences on how detailed or fine-grained the composition can effectively be.

A highly modular composition keeps the two input models as separated as possible in the resultant composed model. This is usually achieved through the definition of an interface between them.

A low or non-modular composition fuses the input models together into a result model where little or no distinction can be seen between them.

Composition modularity provides the same benefits as a modularity in a model: a cleaner and controlled interaction between modules (in this case the original models that are composed) which translates to higher extensibility and maintainability.

Possible values are: **Low** or **High** modularity.

4.13 Intrusion

This criterion focuses on how much, if any, modifications are needed to be performed on the models before they can be ready for composition or the modifications they suffer during it.

An approach is considered to have low intrusion if it does not have to modify the models it composes or if it only does it to a small degree. An example of a low intrusion approach are UML profiles since they allow the extension of a model by adding stereotypes that only change the syntax of a model slightly. However, these syntactic extensions can still change the semantics of it greatly or even break them, therefore in this example the approach would have syntactically low intrusion.

A high intrusion approach involves extension of models, redefinition of elements or even their removal.

Low intrusion preserves the original intentions of the input models designers but is not always possible because of incompatible and/or incomplete syntax and/or semantics. In those cases, modifications have to be made and therefore a high intrusion composition is required.

Note that intrusion may affect modularity (section 4.12) since modifications may make the resultant model less modular. On the other hand, approaches with a conflict resolution strategy (section 4.11) often affect intrusion since part of the solutions for the conflicts involves modifying the original models in a significant way.

Possible values are: **None**, **Low** or **High** intrusion.

5 Analysis

In this section, the set of classification criteria given in the previous section is used to classify the most relevant techniques described in this paper. The result of the classification is then analyzed in order to discover relevant research directions in model composition for future work.

An overview of this classification is presented in two tables. In each table the criteria are located along the horizontal axis and the approaches along the vertical axis. The first table can be seen in Figure 9 and it contains the criteria of the previous section. This set of criteria can be considered as more related to a modelling point of view while the second set, in Figure 11, focuses on an implementation perspective.

Next, we present the analysis of the criteria of the classification (that correspond to the columns of the table):

- The majority of approaches perform *asymmetric* composition. This is partly because of the large amount of AOM as well as extension approaches which are applied to models with this unbalanced relation.

- Only two approaches are classified as having a *heterogeneous uniformity* and one of them, Interfacing [7], is a general approach without a provided implementation or even formalization. *Heterogeneous* composition is relevant to, among others, multi-paradigm modelling (section 1.3.2) and, according to this classification, is an area that lacks research support at the moment. This is because, as explained in section 4.2, heterogeneous composition is more complex than its homogeneous counterpart.

- In the column of the *domain* criterion section 4.3 we observe that the number of generic and specific approaches are evenly spread. The specific approaches are solely focused on models of the UML family, class and sequence diagrams primarily. In Figure 7, a more detailed comparison can be seen between the *implementation* and *domain* criteria. From this figure it can be seen that approaches of all types of domain have been completely implemented. As explained in section 4.3, most of the generic approaches have been classified as having a complete implementation even though they usually provide a framework which has to be extended in order to allow the composition of models defined in a specific formalism. Non-generic approaches do not allow for such extensions.

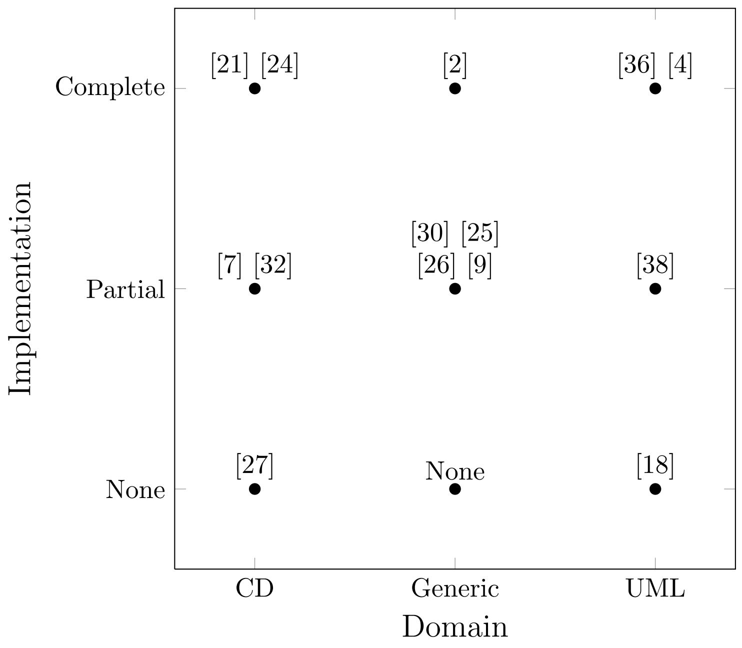

Figure 7: Domain vs. Implementation

- Most of the approaches focus on the composition of UML models and the half of this group on Class Diagrams. Since class diagrams are commonly used for defining metamodels (the abstract syntax of a language), a correlation can be seen for these approaches between the *structural* value of the *nature of models* criterion (section 4.4) and the *Syntax* value of the *DSL weaving applicability* (section 4.6) criterion. This can be explained by the fact that the specification of the syntax of a language is usually done with structural models.

- For the *Time* criterion, we can see that only the Semantic Adaptation approach supports composition of behavioral models with a *continuous* notion of time. This delineates an area where future research is needed: composition of continuous time models.

- It can also be seen that the majority of approaches focus on the composition of the abstract syntax of a language, represented in structural models like class diagrams. Semantics are often more difficult to combine because of their lack of precise and explicit specification, as explained in section 1.2.3. However, in order to compose a DSL, it is necessary to combine all its elements including the semantics. Therefore, composition of language semantics is a very relevant area for future research. A detailed comparison between this and the *implementation* criterion is presented in Figure 8. It can be seen in this figure that approaches for syntax composition (the majority) have several completed implementations which could prove useful for future work in implementation. Also, the approach of Meyers et al. [24] has been fully implemented and it is the only approach from the *Complete Implementation* row that tackles semantic composition of a DSL.

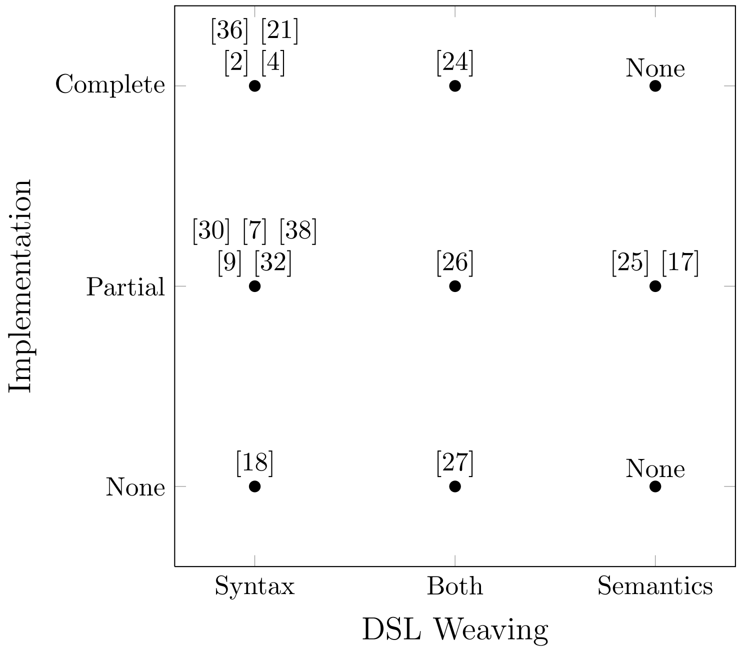

Figure 8: DSL Weaving vs. Implementation

Technique / Criterion	**Symmetry**	**Uniformity**	**Domain**	**Nature of Models**	**Time**	**DSL Weaving**	**Concrete Syntax**
Pottinger Merge [30]	Symmetric	Homogeneous	Generic	Both	Discrete	Syntax	None
Package Merge [36]	Symmetric	Homogeneous	UML	Both	Discrete	Syntax	None
GME Merge [21]	Symmetric	Homogeneous	CD	Structural	-	Syntax	None
Barbero Extension [2]	Asymmetric	Homogeneous	Generic	Both	Discrete	Syntax	None
Template Instantiation [7]	Asymmetric	Homogeneous	CD	Both	Discrete	Syntax	None
MetaDepth Templates [24]	Asymmetric	Homogeneous	CD	Both	Discrete	Both	Textual
Parametrization [27]	Asymmetric	Homogeneous	CD	Structural	Discrete	Both	Visual
Embedding [37]	Asymmetric	Homogeneous	Generic	Both	-	Both	-
Refinement [7]	Asymmetric	Homogeneous	-	-	-	Both	-
Interfacing [7]	Both	Heterogeneous	-	-	-	Both	-
Semantic Adaptation [25]	Asymmetric	Heterogeneous	Generic	Both	Continous	Semantics	None
GeKo [26]	Asymmetric	Homogeneous	Generic	Both	Discrete	Both	None
RAM [18]	Asymmetric	Homogeneous	UML	Both	Discrete	Syntax	None
MATA [38]	Asymmetric	Homogeneous	UML	Both	Discrete	Syntax	None
GMCF [9]	Both	Homogeneous	Generic	Structural	-	Syntax	None
AOAM [32]	Asymmetric	Homogeneous	CD	Both	Discrete	Syntax	None
WEAVR [4]	Asymmetric	Homogeneous	UML	Both	Discrete	Syntax	None
Semantic Weaving [17]	Asymmetric	Homogeneous	SD	Behavioral	Discrete	Semantics	None

Figure 9: First part of the overview of the classification of composition techniques

- *Concrete syntax support* is a feature for which not much literature was found: only two approaches discuss this. In [24], Meyers et al. use the MetaDepth tool to compose *textual* concrete syntax of languages. While the only approach that includes *visual* syntax is the one from Pedro el al. [28] that use the parametrization technique to compose graphical syntax, although they mention that it might not cover all the possible compositions. This is another area of DSL composition where future investigation is required.

- In the *moment of composition* column of Figure 11, we can see that only two approaches realize the composition at run-time (dynamic). All the other approaches do this at design time (static). Both dynamic approaches have in common that they support the composition of semantics. This may have to do with the fact that semantics can be operational which implies that they have to be executed (at run-time) in order to combine them. This is also a point of interest for future research.

- The automation of an approach was measured by the classification scheme in this paper with the *Match Method* criterion. This criterion could either be manual or automatic. Only a few techniques support the automatic matching method which is usually performed based on signatures or names. This matching method often requires input from the user anyway, either before the composition by specifying the criteria upon which a match is based or by resolving conflicts that the match method originated or could not resolve. For a model composition approach to be completely automated, the domain of models where it could be applicable would have to be very limited in order to cover all possible cases. Further research on this subject is needed to determine if this is at all possible.

- As mentioned before, conflicts are often a part of a model composition process. Only 5 approaches support *conflict resolution.* From Figure 10 we can see that only one approach that supports a kind of conflict resolution has been completely implemented: the Motorola WEAVR of Cottenier et al. [4], which supports conflict avoidance. A special note about Cottenier's approach is that they have not included comprehensive details about their implementation since their technique is the only one that is deployed in an industrial context (mobile phones). We can observe from Figure 11 that there is a lack of support in general for conflict resolution.

- The *modularity* of the approaches seems to be evenly distributed in the classification. It can be seen from the table that, as mentioned in section 4.13, there is no real correlation between this and the Intrusion criterion. It is also observed from the table that the template-based approaches [7, 24, 27] have high modularity. This may have to do with the fact that formal or template parameters are used which have to be instantiated, this provides a clean separation between the template and extension models which is the definition of modularity.

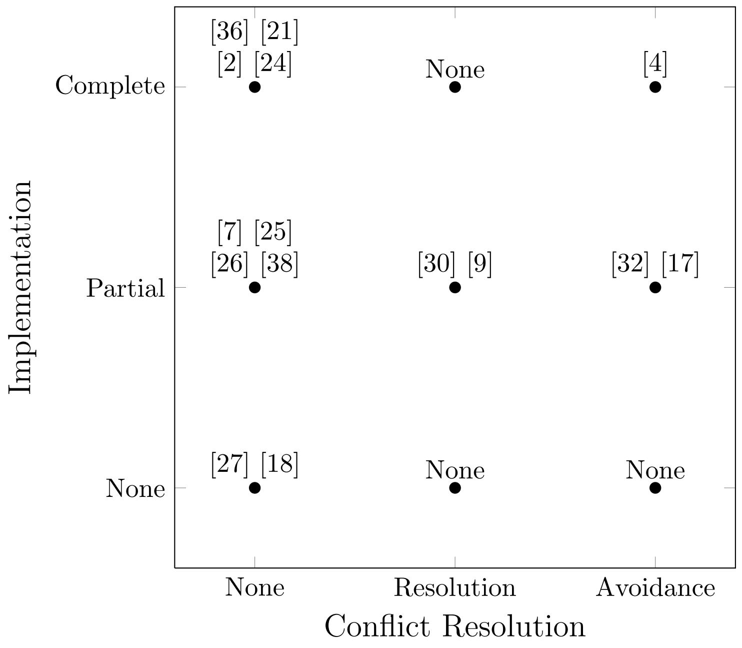

Figure 10: Conflict Resolution vs. Implementation

- Finally, the classification shows a predominance of *low intrusion* approaches. This is sometimes related to an existing conflict resolution strategy in e.g. [30] and [17] since the original models have to be modified in order to prevent or solve conflicts in the final model. How relevant the level of intrusion is to an approach is still not perfectly clear and is another area for future research.

Technique / Criterion	**Implementation**	**Moment of composition**	**Match method**	**Conflict Resolution**	**Modularity**	**Intrusion**
Pottinger Merge [30]	Partial	Static	Manual	Resolution	Low	High
Package Merge [36]	Complete	Static	Automatic	None	Low	Low
GME Merge [21]	Complete	Static	Manual	None	Low	Low
Barbero Extension [2]	Complete	Static	Automatic	None	Low	Low
Template Instantiation [7]	Partial	Static	Manual	None	High	Low
MetaDepth Templates [24]	Complete	Dynamic	Manual	None	High	Low
Parametrization [27]	None	Static	Manual	None	High	Low
Embedding [37]	None	-	-	-	Low	High
Refinement [7]	None	-	-	-	High	Low
Interfacing [7]	None	-	-	-	High	None
Semantic Adaptation [25]	Complete	Dynamic	Manual	None	High	Low
GeKo [26]	Partial	Static	Manual	None	Low	Low
RAM [18]	None	Static	Manual	None	Low	Low
MATA [38]	Partial	Static	Manual	None	Low	Low
GMCF [9]	Partial	Static	Automatic	Resolution	Low	Low
AOAM [32]	Partial	Static	Automatic	Avoidance	Low	Low
WEAVR [4]	Complete	Static	Manual	Avoidance	-	-
Semantic Weaving [17]	Partial	Static	Automatic	Avoidance	Low	High

Figure 11: Second part of the overview of the classification of composition techniques

6 Conclusion

In this paper, a literature review has been provided for the problem of DSL weaving which can be seen as a form of (multiple) model composition(s). In addition, a classification of solutions concerning this problem and its findings has been presented.

After analyzing the results of this classification we can conclude that the following areas are not common in the literature and are therefore relevant for future research in DSL composition:

- Heterogeneous model composition, which allows for multi-paradigm modelling and is a challenging subject that has not been explored enough according to our findings,
- Generic composition, is this "one size fits all" approach at all possible for DSL composition? What are its drawbacks and advantages over a specific approach?;
- The composition of the concrete syntax of modelling languages can be very subjective and difficult to realize and seems to be the part of a DSL where the least research has been done so far;
- Many authors agree that the formalization and composition of the semantics of a language is not an easy task and has to be investigated further;
- Continuous time behavioral models, which entail higher complexity than their discrete counterpart, are a more mathematical aspect of modelling for which little research has been conducted;
- Conflict resolution, an essential step in the composition process, has to be explored more in detail as well;
- It is debatable whether dynamic composition (at run-time) is applicable only for the (operational) semantics of a language or also for other cases. What are its benefits and drawbacks compared to a static composition (at design time);
- Furthermore, additional research is needed to determine the effects and importance of intrusion in composition techniques.

Finally, one area which has been well researched, is the composition (whether by merging or extending) of metamodels that are represented in UML class diagrams, which corresponds to representation of the abstract syntax of DSLs.

References

[1] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle. Touchram: A multitouch-enabled tool for aspect-oriented software design. In *Software Language Engineering*, pages 275–285. Springer, 2013.

[2] Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. A practical approach to model extension. In *Model Driven Architecture-Foundations and Applications*, pages 32–42. Springer, 2007.

[3] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard F Paige. A canonical scheme for model composition. In *Model Driven Architecture–Foundations and Applications*, pages 346–360. Springer, 2006.

[4] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. The motorola weavr: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, 32:44, 2007.

[5] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[6] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving uml package merge. *Software & Systems Modeling*, 7(4):443–467, 2008.

[7] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[8] Fleurey. Kompose: a generic model composition tool. `http://www.kermeta.org/kompose/`, 2007.

[9] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *Models in software engineering*, pages 7–15. Springer, 2008.

[10] Robert France, Indrakshi Ray, Geri Georg, and Sudipto Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings-Software*, 151(4):173–185, 2004.

[11] Cécile Hardebolle and Frédéric Boulanger. Modhelx: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.

[12] Wai Ming Ho, J-M Jézéquel, Alain Le Guennec, and François Pennaneac'h. Umlaut: an extendible uml transformation framework. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 275–278. IEEE, 1999.

[13] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008.

[14] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* Wiley-IEEE Computer Society Press, 2008.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming, in proceedings of the european conference on object-oriented programming (ecoop), finland. *Springer-Verlag LNCS*, 1241:16, 1997.

[16] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 87–98. ACM, 2009.

[17] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 27–38. ACM, 2006.

[18] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA*, 2007.

[19] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[20] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616. ACM, 2006.

[21] Ákos Lédeczi, Greg Nordstrom, Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On metamodel composition. In *Control Applications, 2001.(CCA'01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760. IEEE, 2001.

[22] Raphael Mannadiar and Hans Vangheluwe. Domain-specific engineering of domain-specific languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, page 11. ACM, 2010.

[23] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 21–28. ACM, 2010.

[24] Bart Meyers, Antonio Cicchetti, Esther Guerra, and Juan De Lara. Composing textual modelling languages in practice. In *Procs. of the Intl. Workshop on Multi-Paradigm Modeling (MPM'12)*, 2012.

[25] Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, Hans Vangheluwe, et al. A dsl for explicit semantic adaptation. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling at MODELS 2013*, 2013.

[26] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th international workshop on Early Aspects*, pages 11–18. ACM, 2008.

[27] Luis Pedro, Didier Buchs, and Vasco Amaral. Foundations for a domain specific modeling language prototyping environment. 2008.

[28] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. In *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pages 889–898. Springer, 2009.

[29] Luis Miguel Pedro. A systematic language engineering approach for prototyping domain specific modelling languages. 2009.

[30] Rachel A Pottinger and Philip A Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 862–873. VLDB Endowment, 2003.

[31] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model composition-a signature-based approach. In *Aspect Oriented Modeling (AOM) Workshop*, 2005.

[32] Y Raghu Reddy, Sudipto Ghosh, Robert B France, Greg Straw, James M Bieman, Nathan McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. In *Transactions on Aspect-Oriented Software Development I*, pages 75–105. Springer, 2006.

[33] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A survey on aspect-oriented modeling approaches. *Relatorio tecnico, Vienna University of Technology*, 2007.

[34] Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[35] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a model transformation language. In *Domain Engineering*, pages 211–237. Springer, 2013.

[36] OMG UML. 2.1. 1 superstructure specification (formal/2007-02-03). Technical report, Technical report, Object Management Group, February 2007. available at www. omg. org, downloaded at May 25 th, 2007.

[37] Antonio Vallecillo. On the combination of domain specific modeling languages. In *Modelling Foundations and Applications*, pages 305–320. Springer, 2010.

[38] Jon Whittle and Praveen Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *Models in Software Engineering*, pages 16–27. Springer, 2008.

[39] Bernard P Zeigler, Herbert Praehofer, Tag Gon Kim, et al. *Theory of modeling and simulation*, volume 19. John Wiley New York, 1976.