Pattern-Based Development of Domain-Specific Modelling Languages

Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara
Universidad Autónoma de Madrid

***Abstract*—Model-Driven Engineering (MDE) promotes the use of models to conduct all phases of software development in an automated way. Models are frequently defined using Domain-Specific Modelling Languages (DSMLs), which many times need to be developed for the domain at hand. However, while constructing DSMLs is a recurring activity in MDE, there is scarce support for gathering, reusing and enacting knowledge for their design and implementation. This forces the development of every new DSML to start from scratch.**

To alleviate this problem, we propose the construction of DSMLs and their modelling environments aided by patterns which gather knowledge of specific domains, design alternatives, concrete syntax, dynamic semantics and functionality for the modelling environment. They may have associated services, realized via components. Our approach is supported by a tool that enables the construction of DSMLs through the application of patterns, and synthesizes a graphical modelling environment according to them.

***Index Terms*—Domain-Specific Modelling Languages, Meta-Modelling, Meta-Modelling Patterns, Modelling Environments**

I. Introduction

Model-Driven Engineering (MDE) [6] is a software engineering paradigm that aims at reducing the accidental complexity of software systems by promoting the use of models that focus on the essential complexity of systems. In this way, models in MDE are first-class artefacts of the software development process, from which typically a significant part of the application is derived. To take advantage of the knowledge and expertise within a domain, models are often described with Domain-Specific Modelling Languages (DSMLs). DSMLs enable the description of systems from the point of view of the problem domain while hiding the accidental complexity of the technical solution. This improves productivity and allows non-technical personnel to use and understand the DSML [22].

Hence, a recurring activity in MDE is the definition of new DSMLs and the development of modelling environments for them. Building a DSML is costly and requires specialized technical skills, as it involves defining its abstract syntax (normally through a meta-model), its concrete syntax, and its semantics (e.g., via a simulator or a code generator). Moreover, these different aspects must be supported by an integrated modelling environment with advanced functionality, such as support for model modularization or model filtering. Even though there are many software frameworks to ease the development of textual and graphical environments [17], [22], [35], nowadays, the creation of DSMLs is mostly an ad-hoc process and lacks the ability to build on existing knowledge coming from the previous construction of similar DSMLs.

To alleviate this situation, we propose the assisted construction of DSMLs by means of the use of patterns for different concerns of DSMLs, like frequent domain abstractions, design solutions, concrete syntax representations, dynamic semantics, and functionality of the modelling environment. Patterns may have variants to account for different pattern realizations, which can be selected through a feature model [19]. While some patterns are used in a constructive way, that is, they add elements to the meta-model to speed up the productivity, other patterns are used to configure the services that the modelling environment will provide. Services are realized via components, which can be interconnected through compatible interfaces to define how they will collaborate to make available the desired functionality in the modelling environment.

To demonstrate the feasibility of our approach, we have created a tool that assists in the correct application of patterns, and it is able to synthesize a full-fledged graphical modelling environment for the pattern-based customized meta-model of a DSML. The tool includes a catalogue of patterns, some of them identified by an analysis of the ATL meta-model zoo[1].

The rest of the paper is organized as follows. Section II presents a classification of DSML patterns. Then, Section III describes how patterns are defined, Section IV explains how to express and select variants, and Section V introduces pattern services. Section VI presents our tool. Finally, Section VII discusses related work and Section VIII concludes.

II. A Taxonomy of Patterns for DSMLs

A. What's in a Meta-Model?

DSMLs are described by meta-models, but how are these meta-models constructed? Do different meta-models share common features? If so, we could generalize the commonality into reusable assets (patterns) to speed up the construction of new meta-models. To answer these questions, we have analysed a public meta-model repository (the ATL meta-model zoo) to find common domain concepts arising in meta-models. The analysed repository contains 305 meta-models, out of which 20 could not be processed, thus we considered 285 meta-models in our study. Table I summarizes the results, where each row states how many meta-models included some occurrence of the ticked domains. A detailed view of the results is available at http://miso.es/dsets/atlMMzoo/.

We found that different meta-models contained similar concepts. For example, when a meta-model represented behaviour,

[1] http://www.emn.fr/z-info/atlanmod/index.php/Ecore

TABLE I
COMMON DOMAIN CONCEPTS FOUND IN THE ATL META-MODEL ZOO.

Domain Concepts					# meta-models
Workflow	**State Machine**	**Expressions**	**Component**	**Information Def.**	
√	−	−	−	−	10
−	√	−	−	−	4
−	−	√	−	−	34
−	−	−	√	−	9
−	−	−	−	√	23
√	−	−	√	−	1
√	−	−	−	√	1
−	√	√	−	−	1
−	−	√	√	−	1
−	−	√	−	√	7
−	−	−	√	√	4
√	√	√	−	−	1
√	−	√	√	−	1
−	√	√	−	√	1
√	√	√	√	√	5
16 occs.	10 occs.	42 occs.	15 occs.	39 occs.	89 mms, 122 occs.

it frequently included concepts from state machines. Elements from workflow languages, identifying different types of nodes and gateways [9], were also found in several meta-models. When the meta-model had the necessity to describe structure, one may encounter variants of component languages defining component types/instances and input/output ports. For DSMLs describing information, concepts allowing the representation of entities/classes, features and their relations were found. Finally, in DSMLs describing computations, we encountered elements of mathematical and arithmetical expression languages that included classes to represent operands and operators.

We call these concepts *domain patterns*: they are conceptualizations which occur across many meta-models. For small DSMLs, domain patterns may occur in isolation. However, we observed that some meta-models included several domain patterns. Our study shows that 89 meta-models (31%) contain at least one occurrence of the identified domain patterns: 68 meta-models use 1 pattern, 15 use 2 patterns, 3 use 3 patterns, and 3 use 5. Moreover, the joint occurrence of the Expression and Information Definition patterns is relatively high, as meta-models for programming languages contain facilities for defining both structures like classes and expressions.

Hence, since common domain concepts appear across different meta-models, it looks promising to have a means to gather and enact this knowledge to build DSMLs. In the next section, we argue that domain patterns are not enough to describe a DSML, but more information is needed in order to synthesize a customized modelling environment for the DSML.

B. A Taxonomy of Patterns

The definition of a DSML encompasses several aspects. The first one concerns its abstract syntax, which should gather the primitives of the domain, realized in a high-quality meta-model. The second aspect deals with the representation of the DSML, either textually or graphically, through a concrete syntax. In the case of a graphical syntax, aspects like layouting or zooming (e.g., through filters or hierarchical grouping) may also be specified. Third, the DSML semantics specifies the meaning of models, e.g., through simulation, execution, model transformation or code generation. Finally, the editing of models of a DSML is usually performed using a dedicated modelling environment which provides services like model persistence and model conformance checking. We propose patterns to address all these aspects:

- **Domain patterns**. These patterns characterize a family of DSMLs, gathering requirements of similar languages within a domain, and documenting their variability. For example, there may be patterns for workflow languages, expressions (e.g., arithmetic, logical), variants of state machines, query languages, and component/connector architectural languages. A single DSML may use several of these patterns, customized for a given need, and probably extended with other domain-specific concepts.
- **Design patterns**. These lower-level patterns are concerned with the meta-model design. Some examples include patterns describing different options to realise tree-like structures [2], lists, containment relations [8], connectors, or the type/instance relation [24].
- **Concrete syntax patterns**. They characterize families of DSMLs with similar representation [3]. For example, graph-based languages depict concepts using nodes and arrows; hierarchical graph languages represent in addition hierarchy; and tabular languages use columns and rows. Moreover, some domain patterns may attach a predefined instantiation of a concrete syntax pattern, customized for the domain. For example, state machines may be represented with a particular instantiation of a hierarchical graph pattern, where states are depicted as ovals with the name (and maybe other states) inside, and transitions are shown as arrows. Similarly, workflow languages may attach a graph pattern instance that represents each gateway type as a rhombus with a different decoration.
- **Dynamic semantics patterns**. These patterns describe the participant roles in different styles of semantics [4]: Petri-net like, variations of state machines, event-based semantics, data-flow semantics, etc. Alternatively, the semantics could be given by transformations into a semantic domain, or via code generation.

- **Infrastructure patterns**. These patterns identify services typically provided by modelling environments, but which need to be configured for a particular DSML. Some examples of this kind of patterns include model fragmentation strategies allowing the hierarchical decomposition of large models into folders and model fragments [15], model abstraction services to obtain a simpler view of a model containing the subset of elements of interest [11], and different layouts for graphical DSMLs.

Patterns can be used in two ways. First, as a means to raise the productivity, repeatability and reliability of the meta-model construction process, by incorporating the pattern elements (i.e., a meta-model fragment) to the DSML meta-model. Some elements of the pattern may already exist in the meta-model, in which case, only the missing ones are added. This usage is typical for domain and design patterns. In the second way, patterns are a means to configure functionality for the DSML by identifying the pattern elements (or roles) with existing meta-model elements. This is typically the case for concrete syntax, dynamic semantics and infrastructure patterns. In practice, one often has intermediate situations. Once a pattern is applied, the meta-model elements identified or created by the pattern are "annotated" with the pattern roles. In addition, patterns may offer services to be used together with other patterns, like a layout pattern for certain concrete syntax pattern. To define this interaction, patterns can publish the services they provide or require as pluggable components through suitable interfaces.

Next, we introduce a running example illustrating all kinds of patterns except semantics, left for a future contribution.

C. Running Example

Assume we need to build a DSML for the controller software of wind turbines[2]. The language should allow the definition of components, and specifying their behaviour using state machines. As we expect large model instances, the modelling environment should provide support to split and organize models in different files and folders, instead of in a single monolithic file. The environment should also incorporate a filtering facility to help detecting symptoms of incomplete or faulty designs, like states with no incoming transitions or components without ports. Finally, the environment should support the graphical editing of models, as well as their compact visualization using a tree-view.

Although one could build this DSML and its supporting environment from scratch, we propose the use of patterns. Hence, we can use two domain patterns to build the abstract syntax: one for components and another for state machines. For the concrete syntax, we can use the commonly accepted graphical representations for state machines and components. Finally, services like model splitting and model filtering can be configured using infrastructure patterns. The following sections illustrate the process of defining this DSML and synthesizing a modelling environment for it.

[2]The running example is based on a real case study of the MONDO EU project http://www.mondo-project.org, proposed by an industrial partner.

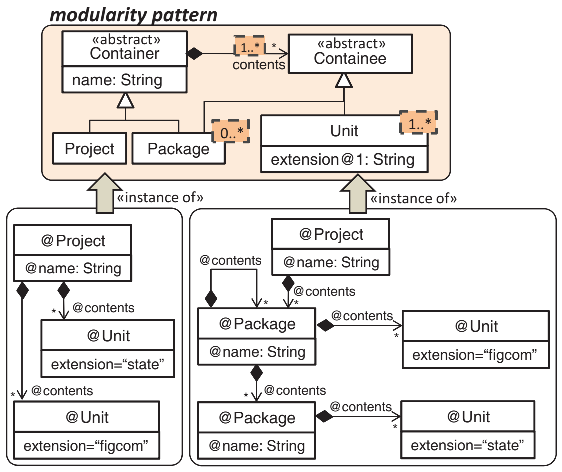


Fig. 1. Infrastructure pattern, and some valid instantiations.

III. Pattern Structure and Application

Our notion of pattern is meta-level independent, as patterns can be applied to models or meta-models. For simplicity of presentation, we assume that they are applied at the meta-model level only. This and the next two sections introduce the main ingredients of our patterns: structure, variant selection and services. In the next subsections, we describe how patterns are specified, applied and combined.

A. Pattern Specification and Instantiation

The structure of a pattern is specified by a meta-model, and its elements (classes, references, and attributes) are called *roles*. The allowed instantiability for roles can be configured through role cardinalities. This cardinality is an interval, possibly with unbounded top, which governs the number of times the role can occur in a pattern instance. If a role does not explicitly define a cardinality, then it is assumed to be [1..1]. Class roles can be tagged with the stereotype abstract. In such a case, the role cannot be instantiated, but it is a placeholder for attribute or reference roles that get inherited by children class roles. Since roles tagged as abstract cannot be instantiated, they do not have any cardinality.

As an example, Fig. 1 shows an infrastructure pattern called modularity to describe model fragmentation strategies. The purpose of this pattern is to offer services to organize a model into projects, packages and units, just like programming language environments organize programs. The associated service maps physically packages into folders, and units into files. This way, models are no longer monolithic, but they can be split across the file system [15].

Patterns are instantiated similar to meta-models, where roles with the stereotype abstract (like Container) cannot be instantiated, the instantiation of every role has to obey the role cardinality (e.g., 1..* for Unit), and attributes and references can only be instantiated if the owner class (for attributes) or incident classes (for references) are instantiated as well. Please note that reference roles have both a role cardinality and a multiplicity. For example, the role cardinality of reference

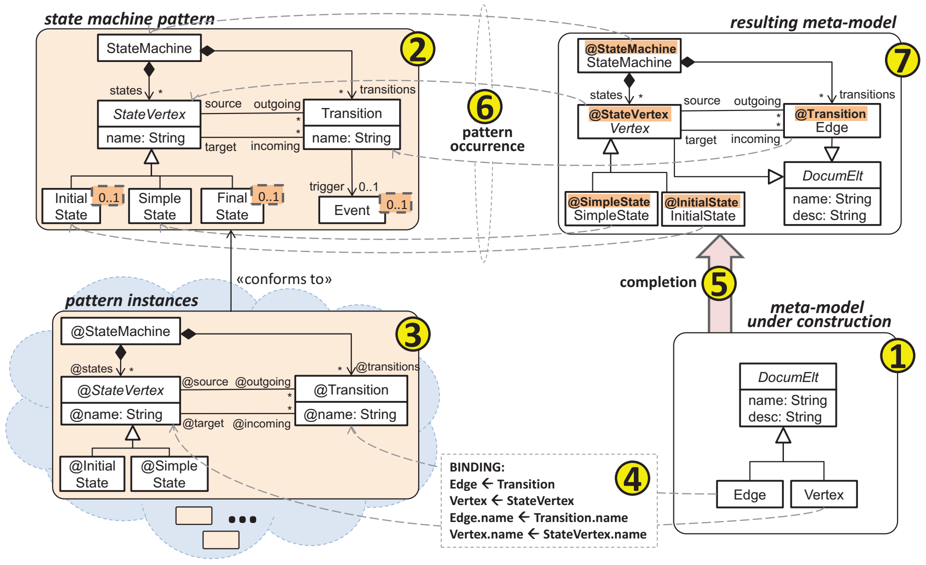


Fig. 2. Structure and application of a pattern for state machines.

contents is 1..*, which means that the role should be instantiated at least once whenever a subrole of Container is instantiated. However, the multiplicity of contents is *, which means that any instance of contents should have multiplicity *. Alternatively, multiplicities could be described using constraints instead of a concrete value (like *). This way, we could specify that all instantiations of contents should have minimum multiplicity 1 or bigger, and any value for the maximum multiplicity.

The bottom part of Fig. 1 shows two valid instantiations of the pattern, describing two fragmentation strategies. As the pattern indicates, both instantiations contain mandatorily an instance of Project. In the left instance, a model is organized in a project, and partitioned in two types of files. The pattern instance to the right enables a more sophisticated organization, where projects contain two kinds of packages with one unit kind each, and one of the package types can have subpackages.

Finally, we distinguish two kinds of attribute roles in patterns. The first kind corresponds to regular attributes (like name in Fig. 1), and the second kind are pattern configuration attribute roles which must receive a value in each instantiation (like extension, used to customize the extension of each file type). In the next section we will see that, in addition, each kind of attribute behaves differently when a pattern is applied.

B. Pattern Application

Patterns are applied to meta-models by selecting one pattern instance, and then binding the instantiated roles to classes, attributes or references in the meta-model. If a role is left unbound, then the corresponding class, attribute or reference is created new in the meta-model.

Resuming our running example, we will illustrate the pattern application process using a domain pattern. Fig. 2 shows the structure of a simple state machine pattern in the upper left (label 2). InitialState, FinalState and Event are optional, so they receive the cardinality $[0..1]$. The rest of roles, including reference and attribute roles, have $[1..1]$ cardinality (omitted). StateVertex is not an abstract role (i.e., it is not tagged as abstract), but it is a class role demanding the class it gets bound to be abstract; therefore, StateVertex can be instantiated.

Fig. 2 shows how the pattern (label 2) is applied to a meta-model (label 1). As abovementioned, conceptually, a pattern defines the set of allowed instantiations that result from the possible values for its different role cardinalities. In this case, as most roles have cardinality 1, the variability results from the instantiation or not of the optional roles InitialState, FinalState and Event. The meta-model designer selects one of such pattern instances (label 3); the one selected in the example does not contain the roles Event and FinalState. This pattern instance will be incorporated to the meta-model. However, not every pattern role of the chosen instance will be necessarily created new in the meta-model, but it is possible to *bind* existing meta-model elements to roles in the pattern (label 4). In the figure, Edge is bound to Transition, and Vertex to StateVertex.

Our bindings allow structural matching. This means that attribute and reference roles are requirements to be satisfied by the meta-model class to which the owner class role of the attribute or reference roles is bound. In this way, attribute name in the pattern instance should be bound to some String attribute of the meta-model class the StateVertex role is bound to, or to some superclass. Our support for structural binding takes into account that Edge and Vertex in the meta-model inherit an attribute name, and hence, it is possible to bind Edge.name to Transition.name, and Vertex.name to StateVertex.name.

Regarding the binding of attribute roles, the regular ones can be bound to existing meta-model attributes, and are created if they are left unbound. This is the case of attribute role name in Fig. 2. Pattern configuration attribute roles (like extension in Fig. 1) are always created new, and need to receive a value at the pattern instance/meta-model level. To deal with both kinds of attributes, we take ideas from multi-level modelling [1] and consider they have a potency, i.e., a natural number specifying at how many meta-levels below they receive a value. Pattern configuration attributes have potency 1 (indicated by @1 in the pattern) and receive a value in meta-models. Regular attributes have potency 2 (not shown as it is the default potency value) and receive a value in models.

Once the binding is performed, the unbound pattern elements are created in the meta-model. Moreover, both the created meta-model elements (like SimpleState) and the ones identified by the binding (like Vertex) are annotated with their role in the pattern (labels 5 and 7). This annotation can be considered a partial typing of the meta-model w.r.t. the pattern (label 6), which signals a pattern occurrence. In contrast to the conformance relation between a model and a meta-model, which is mathematically a total function (i.e., every model element is typed), the relationship between a meta-model and a pattern is partial, as only some meta-model elements are typed by some pattern role.

The presented example is the typical situation for domain patterns, where a few meta-model elements are bound, and the meta-model is completed with new elements derived from the unbound pattern roles. In contrast, infrastructure patterns (like the one in Fig. 1) hardly ever imply the creation of elements.

C. Combined Application of Patterns

Sometimes, it is useful to be able to enrich a pattern with others that cover complementary aspects, so that whenever the first pattern is applied, the rest are automatically applied as well. This could be used, for example, to assign a default graphical concrete syntax to state machines, that gets automatically created. For this purpose, patterns can define any number of secondary patterns that become applied when the main pattern is applied. The dependency between the main and the secondary patterns is declared in two steps: first, the secondary pattern is instantiated, and then, the roles in this instantiation must be bound to roles in the main pattern. This is similar to how patterns are applied to meta-models, but in this case, roles in the secondary pattern instance cannot be unbound. Using this mechanism, the roles in the main pattern "receive" roles from the secondary patterns, which get transferred to the meta-models where the main pattern is applied.

As an example, Fig. 3 shows how to use the concrete syntax pattern graph-based representation as secondary pattern of the state machine domain pattern. The concrete syntax pattern has roles for edges and for several graphical forms of nodes, which can be customized via pattern configuration attributes (i.e., attribute roles with potency 1). The pattern is instantiated in the bottom-right of the figure for state machines, assigning appropriate values to the pattern configuration attributes. Then, a binding maps elements in the main pattern to their graphical representation (e.g., InitialState to the Circle with radius 10), so that when the former elements are instantiated in a meta-model, their graphical representation is instantiated as well. Please note that secondary patterns are proper patterns that can be applied to a meta-model on their own.

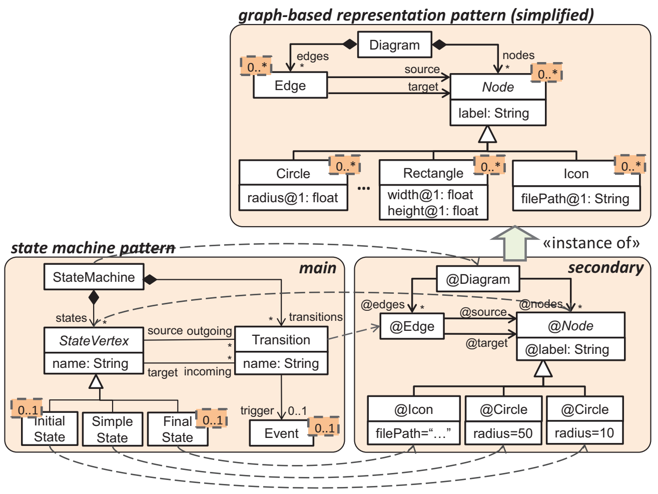


Fig. 3. Defining a secondary pattern.

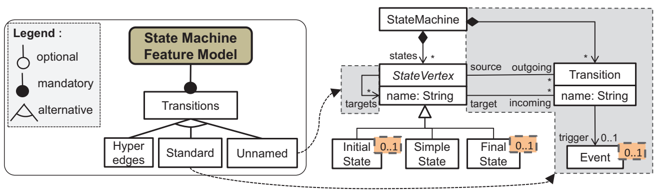


Fig. 4. Feature model for state machine pattern variants.

IV. Pattern Variants

The cardinality of pattern roles allows their fine-grained customization for a context. In addition, a pattern may have variants accounting for coarse-grained alternatives in the pattern structure. For example, transitions in a state machine may be represented as references between two states, as hyperedges connecting multiple source to multiple target states, or with an intermediate class storing properties of the transition. Only the latter two cases allow associating a trigger to transitions. This variability in the pattern structure would be challenging to express with role cardinalities alone.

For this reason, we equip patterns with a feature model [19] that defines the coarse-grained variability of patterns. As an example, the left of Fig. 4 shows the feature model attached to the pattern for state machines. It defines three alternatives for the design of transitions (Hyperedges, Standard and Unnamed), and each alternative is exclusive (i.e., only one can be selected at a time). Hence, transitions can be either a class like in Fig. 2 (variant Standard), a hyperedge, or a reference (variant Unnamed, in which case there are no classes for the roles Transition and Event, but there is an association starting and ending in StateVertex instead). Each variant has associated a meta-model fragment. When the pattern is instantiated, the designer selects the desired variant, and the associated meta-model fragment will be added to the pattern structure in that particular instance. The figure shows the fragments associated to the standard and unnamed variants as shadowed regions, while the one for hyperedges is omitted.

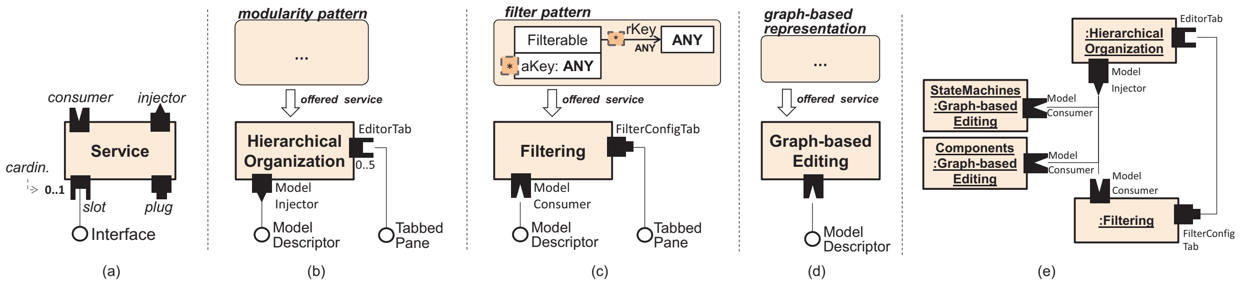


Fig. 5. (a) Schema of pattern services. (b,c,d) Services in the running example. (e) Service composition.

Technically, our pattern definition is created using superimposed variants [10]. Thus, the pattern structure is described by a meta-model that contains the elements of all possible variants, and each feature may have an associated *selection* reference pointing to the elements of the meta-model enabled by the feature and will be kept upon the feature selection. This form of variability is adequate in this context since patterns are generally small and contain few variants.

V. Pattern Services

Patterns may include services contributing functionality to the environment generated for a DSML. Fig. 5(a) shows the general schema of a pattern service. A service is encapsulated as a component, which may define any number of ports. Each port declares an interface and can be of four different types: *slot*, *plug*, *injector* and *consumer*. Services can be connected through their ports if their types and interfaces are compatible. Regarding type compatibility, slots are compatible with plugs, and injectors are compatible with consumers.

A slot represents a functionality "hole" in a service, to be provided by another service that declares a plug with a compatible interface. Moreover, slots have a cardinality which constrains the allowed number of plugs that can be connected to them. Typically, the functionality provided by a plug needs to be deployed in the context of a service with a compatible slot, while slots with a minimum cardinality 1 need to be connected to a compatible plug to obtain a proper behaviour. On the other hand, an injector port is an emitter of information populated by a service. This information can be used by a consumer port with a compatible interface. In this way, a connection between an injector and a consumer induces a dependency injection from the service defining the injector to the service defining the consumer.

For example, Figs. 5(b) and (c) show two infrastructure patterns that define two services that we would like to have in the modelling environment for the running example. The first one is the modularity pattern previously shown in Fig. 1. This allows generating a modelling environment where models are organized hierarchically into projects, packages and files of different types, similar to the organization of Java software projects. The associated service has a slot named EditorTab that allows other services to extend the environment with tabs that contribute further functionality. The service also defines an injector which supplies to consumers a ModelDescriptor object with information of the model unit selected in the environment.

The infrastructure pattern in Fig. 5(c) allows customizing model filters. The classes amenable to be filtered should be bound to Filterable, while the attributes and references to be considered in the filtering conditions should be bound to aKey and rKey, being possible to have any number of them. Since the actual type of the attributes bound to aKey and the references bound to rKey are unimportant, they are tagged as ANY. This pattern defines a service that is compatible with the previous HierarchicalOrganization service: on one hand, it has a plug named FilterConfigTab that contributes a tab to control the model filter behaviour; on the other hand, it has a consumer port named ModelConsumer from which the service will obtain the model unit to filter, selected in the environment.

The service in Fig. 5(d) is associated to the concrete syntax pattern graph-based representation shown in Fig. 3. This service generates an editor to build models using the defined graphical concrete syntax. The service needs a model descriptor, which will be provided by the HierarchicalOrganization service.

When a pattern is applied, its services become available. Services can be optional, in which case, the DSML designer can activate them or not. Since some services produce data to be consumed by other services, or define functionality to be plugged to other services, it is necessary to connect those services to make explicit their dependencies so that they can be resolved. For this purpose, we provide an automated composition mechanism that matches interfaces and connects ports whenever possible. Thus, our different port types describe how services should be composed, and are useful for pattern/service developers. Nevertheless, they are transparent for the DSML designer, because the matching and composition of services is automatic. In this way, the functionality of the environment for a DSML is obtained by the automatic composition of the services associated to the instantiated patterns.

Fig. 6 shows an excerpt of the running example meta-model. It has been created by applying patterns, and therefore, some elements are annotated with pattern roles. In particular, we have applied the state machine and component domain patterns. Both have a secondary graph-based representation concrete syntax pattern with an associated service that will generate a graphical editor for them. We have also applied two infrastructure patterns: modularity and filter.

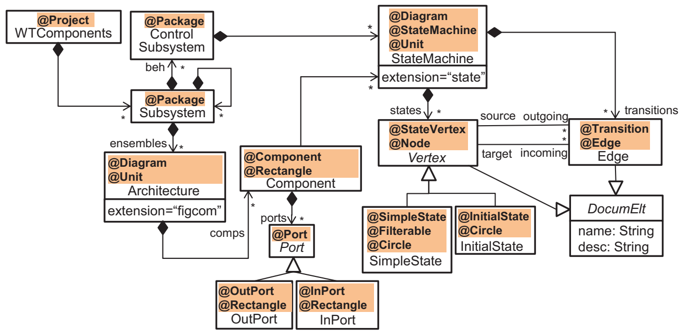


Fig. 6. Resulting meta-model for the running example.

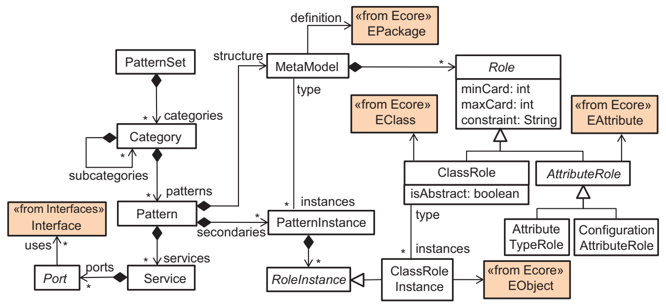


Fig. 7. Excerpt of DSL tao's pattern meta-model.

Fig. 5(e) shows the service composition induced for this meta-model. Our mechanism detects that the slot EditorTab and the plug FilterConfigTab have a compatible interface and connects both ports, and similar for the injector ModelInjector and the three consumers ModelConsumer.

VI. Tool Support

We have built a prototype implementation of our pattern-based approach atop Eclipse. The tool is called DSL tao, and is freely available at http://miso.es/tools/DSLtao.html. Next, we explain its extensible architecture, its functionality, and demonstrate its capabilities to generate a customized graphical modelling environment using the running example.

A. Architecture: DSL tao for Pattern Developers

DSL tao uses Eclipse/EMF as implementation platform to profit from its plugin-based architecture, which allows extending the platform to incorporate new functionality through extension points. This simplifies the task of adding new patterns to the system, and it is a natural deployment infrastructure for our generated DSML environments, as we will show next.

The tool includes an extensible catalogue of patterns. New patterns can be defined by providing its meta-model and role cardinalities. Fig. 7 shows an excerpt of the underlying meta-model for pattern definition. A Pattern declares Roles, which annotate the elements of the ecore Metamodel with the structure of the pattern. Patterns may have any number of PatternInstances, which contain RoleInstances of the roles declared by the pattern. Whereas roles in patterns point to meta-elements like EClass or EAttribute, role instances point to EObjects in the structure of the pattern instance. In addition to their own instances, patterns may have associated secondary instances from other patterns. Finally, patterns may define Services, which need to declare their Ports and Interfaces.

By means of extension points, the pattern developer is allowed to extend the base definition of a pattern to include pattern-specific validations, heuristics or services (if needed):

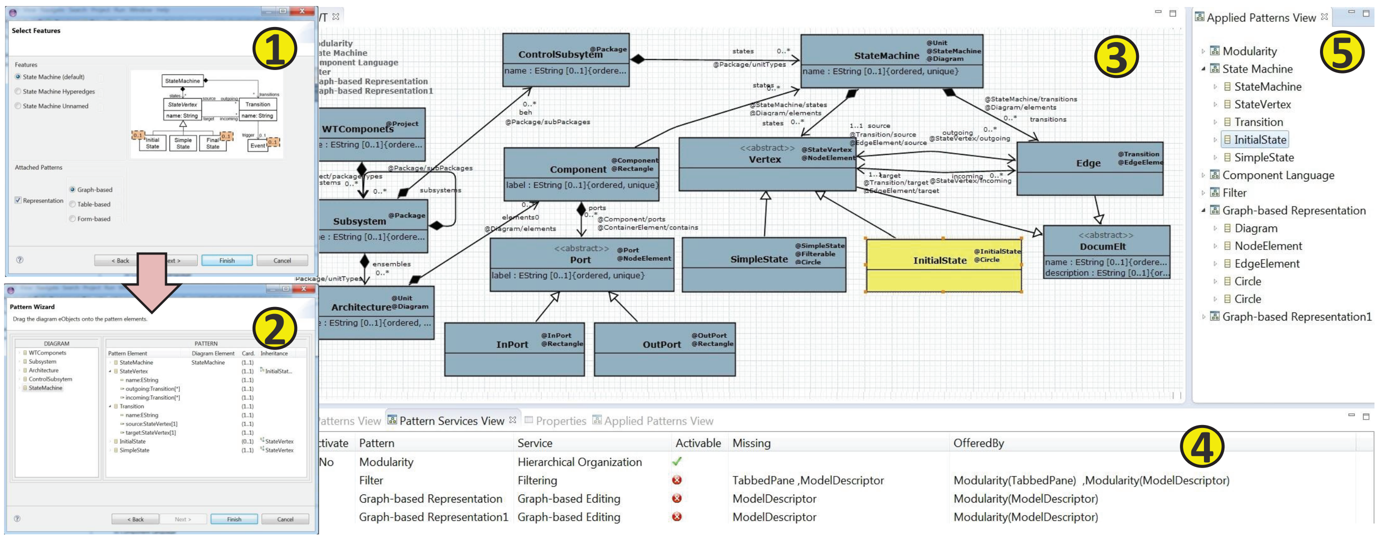
- *pattern-specific validations*: The pattern developer may include extra validations, e.g., expressed in OCL, to check whether a binding is correct for a given pattern. It is used only for pattern-specific validations, as DSL tao already performs generic correctness checkings (e.g., that if a meta-model attribute is bound to a pattern attribute, their owner classes are bound). An example of pattern-specific validation for the modularity pattern is checking that the class bound to Project is root (i.e., it contains all other classes, directly or indirectly).
- *pattern-specific heuristics*: These are heuristics that facilitate the correct instantiation of a pattern. For example, the modularity pattern identifies the root class of a meta-model as the optimal binding for the Project role.
- *services*: The pattern semantics can be realized via services, typically through code generation using Acceleo or any other code generation language, and it is encapsulated as an Eclipse plugin. Code generation is needed because the generated plugin needs to be customized with information of the pattern instance. For example, the plugin generated for the HierarchicalOrganization service needs to enforce the desired project/package/unit structure. Service dependencies are realized as plugin dependencies and appropriate instances of extension points.

B. Using DSL tao to Describe DSMLs

Fig. 8 shows a screenshot of DSL tao. The tool enables the construction of DSML meta-models by dragging elements from a palette into the canvas (label 3). A *Patterns View* (omitted in the figure) lists the patterns of the different categories. When a pattern is selected, a wizard (labels 1 and 2) facilitates its instantiation as follows: first, the pattern variant is selected, together with its secondary patterns (if any). In the figure (label 1), the StateMachine pattern is to be applied, and the designer may choose among three default visualizations (three secondary patterns). Then, the designer can bind meta-model elements to the pattern roles by dragging the former into the latter (label 2), and instantiate the pattern roles according to their cardinality.

Label 3 in Fig. 8 shows the resulting meta-model. To the right (label 5), the *Applied Patterns View* displays a tree containing each pattern instance and instantiated role. Selecting a role highlights the bound meta-model element in the canvas (InitialState in the figure). Each meta-model element shows its roles in patterns as annotations, and the canvas itself shows the list of applied patterns in the upper-right corner.

The tool includes a *Pattern Services View* (label 4 in the figure), where each row indicates a service instance, the pattern that provides the service, and if it is activable or not. If a service is optional, the designer can activate it. A pattern is



ctivate	Pattern	Service	Activable	Missing	OfferedBy
No	Modularity	Hierarchical Organization	✓		
	Filter	Filtering	✗	TabbedPane ,ModelDescriptor	Modularity(TabbedPane) ,Modularity(ModelDescriptor)
	Graph-based Representation	Graph-based Editing	✗	ModelDescriptor	Modularity(ModelDescriptor)
	Graph-based Representation1	Graph-based Editing	✗	ModelDescriptor	Modularity(ModelDescriptor)

Fig. 8. Using DSL tao. (1, 2) Applying the StateMachine pattern. (3) Resulting meta-model. (4) Services. (5) Applied patterns.

activable if all its port dependencies are resolvable. In case a service is not activable, the view informs of which ports are unconnected, and which patterns in the repository could resolve the given dependency. The figure shows the Hierarchical Organization service deactivated, which makes the Filtering and the two Graph-based editing services unavailable. Note that the DSML designer is unaware of the different port types (slot, plug, injector, consumer) and the composition of services, as this is automatically performed by the tool.

C. Generation of Modelling Environments with DSL tao

The modelling environment for a DSML can be synthesized once its meta-model is completed. For this purpose, DSL tao invokes the code generators of the active services associated to the applied patterns. This generates a modelling environment made of several contributing plugins, where typically, each plugin is generated by one service.

Fig. 9 shows some snapshots of the generated DSML environment for the running example. The environment is made of four plugins, contributed by the modularity pattern, the filter pattern, and two by the graph-based representation pattern (applied as secondary patterns when instantiating the state machine and component domain patterns). The modularity pattern organizes a model as an Eclipse project, where the model is broken into units (files) and packages (folders). By breaking the model, operations working on its parts become more efficient, as the whole model does not need to be in memory at the same time. Moreover, the hierarchical decomposition of models promotes their comprehensibility.

Label 1 in the figure shows the Eclipse *Package Explorer*, which contains a model of our DSML organized into packages and file units. Because the model is physically fragmented in the file system, the environment allows moving around packages and units, and updates the underlying model accordingly. Moving elements to locations that result in an incorrect model is prevented by the environment. This is accomplished with the help of Concordance [27], a model indexer that keeps track of inter-model references.

The environment has a dedicated editor for each file unit type, with tabs that provide different views of the unit content (e.g., as a tree, as a table, etc.). The Filtering service contributes with an additional tab (label 2) that permits defining filters for simple states, as this was the class annotated with the Filterable role. The filtering conditions can be defined using a form-like user interface, and the result of applying the filter can be viewed on the other tabs. The figure shows the condition to filter states without incoming transitions (label 2). The purpose is to find unreachable states, which is challenging to do by hand in case of big models. Labels 3 and 4 show the model before and after applying the filter.

In addition, the environment includes a graphical editor for state machines (label 5) and another for components. When opening a file unit, the environment offers the possibility of using either a tree editor or the appropriate graphical editor. The graphical editors are based on Sirius [29], which is a model-based framework that enables the specification of graphical modelling environments by defining a model that describes the graphical elements to be used (e.g., rectangles, circles), their mapping to meta-model elements, palette buttons, and so on. Our approach is to transform the concrete syntax pattern instances into Sirius models. As Sirius is interpreted, we include such Sirius models in the generated environment, in order to allow users their modification if so desired.

Altogether, the environment has 7200 lines of generated code for the modularity and filtering mechanisms, apart from the Sirius editor. This would be costly to develop by hand.

As a summary, we have illustrated how to use our novel approach to develop DSML meta-models by reusing domain patterns, combined and adapted to the application context, and how to generate functionality-rich modelling environments by using infrastructure patterns.

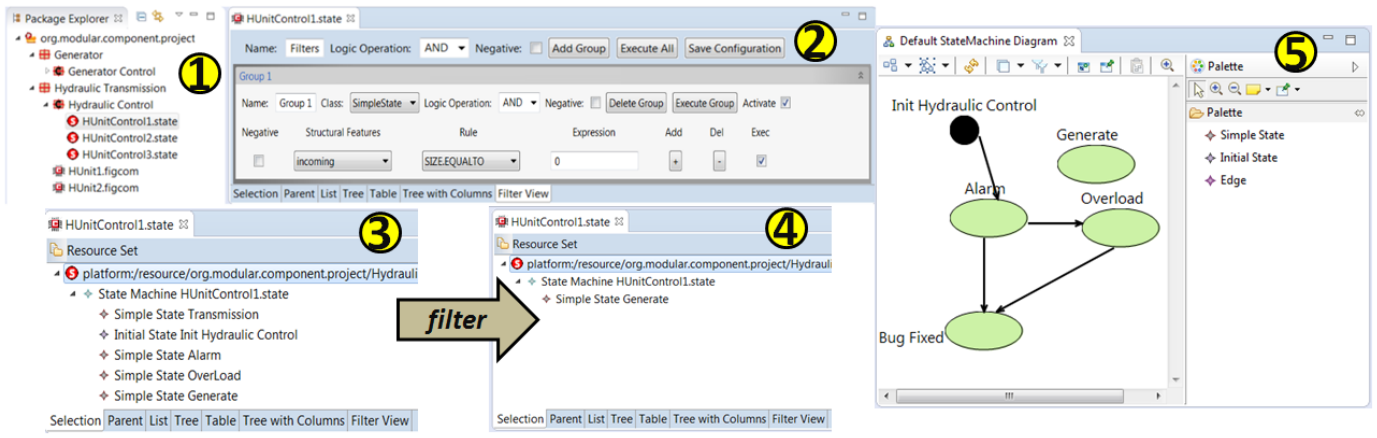

Package Explorer
org.modular.component.project
Generator
Generator Control
Hydraulic Transmission
Hydraulic Control
HUnitControl1.state
HUnitControl2.state
HUnitControl3.state
HUnit1.figcom
HUnit2.figcom
1
HUnitControl1.state
Name: Filters Logic Operation: AND Negative:
Add Group
Execute All
Save Configuration
2
Group 1
Name: Group 1 Class: SimpleState Logic Operation: AND Negative:
Delete Group
Execute Group
Activate
Negative
Structural Features
Rule
Expression
Add
Del
Exec
incoming
SIZE.EQUALTO
0
Selection Parent List Tree Table Tree with Columns Filter View
HUnitControl1.state
Resource Set
platform:/resource/org.modular.component.project/Hydrauli
State Machine HUnitControl1.state
Simple State Transmission
Initial State Init Hydraulic Control
Simple State Alarm
Simple State OverLoad
Simple State Generate
Selection Parent List Tree Table Tree with Columns Filter View
3
filter
HUnitControl1.state
Resource Set
platform:/resource/org.modular.component.project/Hydrauli
State Machine HUnitControl1.state
Simple State Generate
Selection Parent List Tree Table Tree with Columns Filter View
4
Default StateMachine Diagram
Init Hydraulic Control
Generate
Alarm
Overload
Bug Fixed
Palette
Palette
Simple State
Initial State
Edge
5

Fig. 9. Generated environment: filters and state machine graphical editing environment.

VII. Related Work

We revise works on the definition and application of DSML design patterns, the use of patterns to improve productivity in DSML construction, and the classification of DSML patterns.

Pattern definition. Several works formalize patterns and automate their application for some notation [5], [14]. Compared to [5], our patterns are more expressive due to the instantiation mechanism. The closest formalization to our approach is [14], which applies to UML diagrams. In this approach, patterns have roles with cardinalities. We enrich this pattern definition in several ways. First, we recast it in a multi-level setting, which permits attributes with potency 2 that can be bound to meta-model attributes, and attributes with potency 1 that receive a value. Second, we allow combined pattern applications and variants through a feature model. Moreover, our patterns include services to help creating DSML environments, and our tool supports pattern-based meta-model completion.

Patterns for DSML construction. Most works that use patterns for building DSMLs only handle one or two aspects of our pattern taxonomy, mainly design patterns. For instance, in [8], the authors define meta-modelling design patterns directed to design decisions, with no support for variability. In [28], the authors detect the lack of engineering processes for DSML construction, and propose documenting DSML requirements using use cases and design patterns. Spinellis [30] defines architectural patterns for DSML design. Altogether, design patterns in these works improve the inner quality of the DSML, but they are low-level and provide less gain in productivity compared to domain and infrastructure patterns.

Domain patterns capture domain knowledge and speed up the construction of DSML meta-models. Similarly, [26] argues on the benefits of building DSMLs by composing domain concepts. A domain concept is a meta-model, and its semantics is given as a model transformation. The authors define some composition operators, and aim at composing the respective transformations. Instead, our patterns reside at a higher meta-level than the DSML meta-model, and their instantiation mechanism is more flexible. Moreover, we support combined pattern application, variants, and the component-based construction of the DSML environment. Also related to our proposal, in [34], DSMLs can have attached a feature model, and the selection of features yields a meta-model variant. We, in addition, provide further flexibility by the instantiation of role cardinalities, and enable a way to generate customized modelling environments by associating services to patterns.

In the *grammarware* technical space, Neverlang [33] allows building textual languages from slices that contain fragments of the concrete syntax and semantics (a textual grammar and an evaluator). Slices are combined using a feature model. Our approach addresses the peculiarities of the *modelware* technical space. It provides extra flexibility for pattern instantiation, and it is possible to apply several patterns at once.

Finally, profiles are extensions of a base language, normally UML [32], where diagram types are fixed. Instead, our approach is extensible as new patterns can be added to our catalogue. Moreover, profiles do not provide additional services as our patterns do (e.g., modularization or filtering).

Generation of graphical modelling environments. Many tools to develop graphical tools for different applications have emerged along the years, like meta-CASE tools [22], diagram sketching [7], or multi-formalism modelling and simulation [12]. The advent of Eclipse has promoted frameworks to build visual editors as plugins, like GMF [16], Eugenia [23], Spray [31], Graphiti [18], or Sirius [29]. These tools are model-based, except Graphiti, which relies on a Java API and coding. Some of them generate artefacts for other lower-level approaches. For example, Eugenia is built atop GMF, and Spray atop Graphiti. In our case, the graphical editors produced by DSLtao are based on Sirius. All these frameworks rely on code generation except Sirius, which is interpreted. The way of specifying the concrete syntax varies: Eugenia requires annotating the meta-model elements, Spray uses a textual DSL, GMF and Sirius require building models that describe the graphical syntax, and Graphiti requires Java programming. Conceptually, our approach is closer to Eugenia, as our pattern applications result in meta-model annotations. However, DSL tao allows attaching concrete syntax styles to domain patterns, which speeds up the generation of graphical environments. This feature, and the ability to specify tool services via infrastructure patterns, are unique among the mentioned tools. In particular, none of these tools is able to produce graphical environments with model fragmentation capabilities.

Concerning concrete syntax patterns, the VL-Eli system [21] supports the creation of visual languages by defining a textual grammar for the abstract syntax, and choosing between several patterns for the concrete syntax. They consider patterns for lists, tables, and other basic visualizations. However, they lack reuse of domain patterns and infrastructure services.

Classification of DSML patterns. Several authors have acknowledged the difficulty of developing DSLs [20], [25], [28] and propose catalogues of patterns to facilitate this task. For instance, [25] identifies patterns that help in the decision, analysis, design and implementation of DSLs. Our pattern taxonomy does not cover the decision and analysis phases, but focuses on design and implementation. Our domain patterns can be seen as an incarnation of their so-called *language exploitation* pattern, and we give semantics to pattern instances using their so-called *generator* implementation pattern. In [13], Fowler outlines some design/implementation patterns and idioms for internal DSLs, whereas we assume external DSLs. Finally, [20] proposes design guidelines for DSMLs, categorised on language purpose, implementation, design and syntax. Our work is aligned with these guidelines, which recommend composing languages where possible (in our case enabled by the reuse of patterns), and providing organizational structures for models (enabled by our modularity pattern).

VIII. Conclusions and Future Work

We have presented a pattern-based approach for the creation of DSMLs, together with their supporting graphical environments via pattern services. Our patterns enable a fine-grain configuration via role cardinalities, and variant selection using a feature model. Patterns may be applied in combination, and may offer services for the resulting DSML environment, where service composition is entailed by pattern application. The approach is supported by a tool, which we have illustrated by the generation of an environment with rich functionality. Our work paves the way towards sound engineering methods for a more productive, sound, repeatable construction process of DSMLs, a core, recurring activity in MDE.

Currently, we are enhancing the graphical concrete syntax patterns, to allow their application using a dedicated wizard with advanced heuristics. We are also working on defining new patterns, and analysing further repositories to identify domain patterns and services for them (e.g., transformations). We also plan to perform an empirical validation of DSL tao with users.

Acknowledgements

Work supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

References

[1] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[2] M. Blaha. *Patterns of data modeling*. CRC Press, 2010.

[3] P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *VL/HCC*, pages 83–90, 2004.

[4] P. Bottoni, E. Guerra, and J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *J. Vis. Lang. Comput.*, 19(4):429–455, 2008.

[5] P. Bottoni, E. Guerra, and J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *IST*, 52(8):821–844, 2010.

[6] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2012.

[7] F. Brieler and M. Minas. A model-based recognition engine for sketched diagrams. *J. Vis. Lang. Comput.*, 21(2):81–97, 2010.

[8] H. Cho and J. Gray. Design patterns for metamodels. In *DSM*, 2011.

[9] J. S. Cuadrado, E. Guerra, and J. de Lara. A component model for model transformations. *IEEE TSE*, 40(11):1042–1062, 2014.

[10] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[11] J. de Lara, E. Guerra, and J. S. Cuadrado. Reusable abstractions for modeling languages. *Inf. Syst.*, 38(8):1128–1149, 2013.

[12] J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *FASE*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[13] M. Fowler. *Domain Specific Languages*. Addison-Wesley, 2010.

[14] R. B. France, D. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE TSE*, 30(3):193–206, 2004.

[15] A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara. EMF splitter: A structured approach to EMF modularity. In *XM@MoDELS*, volume 1239 of *CEUR*, pages 22–31. CEUR-WS.org, 2014.

[16] GMF. https://wiki.eclipse.org/Graphical_Modeling_Framework.

[17] Graphical modeling project. http://www.eclipse.org/modeling/gmp/.

[18] Graphiti. http://eclipse.org/graphiti/.

[19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis feasibility study. Technical report, CMU-SEI, Nov. 90.

[20] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel. Design guidelines for domain specific languages. In *DSM*, pages 7–13, 2009.

[21] U. Kastens and C. Schmidt. Visual patterns associated to abstract trees. *Electr. Notes Theor. Comput. Sci.*, 148(1):5–18, 2006.

[22] S. Kelly and J. Tolvanen. *Domain-specific modeling - enabling full code generation*. Wiley, 2008.

[23] D. S. Kolovos, L. M. Rose, S. bin Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *MODELS*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.

[24] R. C. Martin, D. Riehle, and F. Buschmann. *Pattern languages of program design 3*. Addison-Wesley, 1997.

[25] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, 2005.

[26] L. Pedro, D. Buchs, and V. Amaral. Foundations for a domain specific modeling language prototyping environment: A compositional approach. In *DSM*, 2008.

[27] L. Rose, D. Kolovos, N. Drivalos, J. Williams, R. Paige, F. Polack, and K. Fernandes. Concordance: a framework for managing model integrity. In *ECMFA*, volume 6138 of *LNCS*, pages 245–260. Springer, 2010.

[28] C. Schäfer, T. Kuhn, and M. Trapp. A pattern-based approach to DSL development. In *DSM@SPLASH*, pages 39–46. ACM, 2011.

[29] Sirius. https://eclipse.org/sirius/.

[30] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[31] Spray. https://code.google.com/a/eclipselabs.org/p/spray/.

[32] UML 2.4.1. http://www.omg.org/spec/UML/2.4.1/, 2011.

[33] E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability support in domain-specific language development. In *SLE*, volume 8225 of *LNCS*, pages 76–95. Springer, 2013.

[34] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26(4):47–53, 2009.

[35] Xtext. http://www.eclipse.org/Xtext/.