DSL-Maps: From Requirements to Design of Domain-Specific Languages

Ana Pescador, Juan de Lara
Computer Science Department
Modelling and Software Engineering Research Group
http://miso.es
Universidad Autónoma de Madrid (Spain)

ABSTRACT

Domain-Specific Languages (DSLs) are central to Model-Driven Engineering, where they are used for creating models for particular domains. However, current research and tools for building DSLs focus on the design and implementation aspects of the DSL, while the requirements analysis phase, and its automated transition to design is largely neglected.

In order to alleviate this situation, we propose DSL-maps, a notation inspired by mind-maps, to represent requirements for DSLs. The notation is supported by a tool, which helps in the automated transition into an initial meta-model design, using a customizable transformation and recommendations from a catalogue of meta-model design patterns.

CCS Concepts

•**Softw. and its engineering** → **System modeling languages; Designing software;** *Design languages;*

Keywords

Domain Specific Languages; Model-Driven Engineering; Domain Analysis; Meta-Modelling Patterns

1. INTRODUCTION

Model-Driven Engineering (MDE) promotes models as the principal assets during the development process. Models in MDE are frequently described using DSLs containing powerful primitives for particular domains [7]. Therefore, a recurring activity in MDE is the development of DSLs and their modelling environments [16].

In MDE, DSLs are constructed using a meta-model describing its abstract syntax. Many meta-modelling tools and DSL workbenches have emerged along the years [7, 8, 12, 13]. In most cases, their entry point is the description of a meta-model. However, a meta-model is already a detailed design of the DSL, because it contains a concrete realization of the domain concepts. Instead, notations supporting the requirements analysis phase, documenting needs of the domain, promoting discussion among stakeholders, and helping in a disciplined transition to design, would complement current meta-modelling tools, as they largely neglect this phase.

In order to alleviate this situation, we propose DSL-maps, a notation to represent requirements for DSLs, and help in their disciplined transition into a design meta-model. The goal of DSL-maps is twofold. On the one hand, to provide a flexible, simple notation promoting discussion among stakeholders. For this reason, the notation is inspired by mind-maps [4], a popular, flexible diagrammatic notation, which is sometimes used by agile methodologies for requirements elicitation [9]. Second, to provide automatic support for the transition into a meta-model design. For this purpose, the approach relies on a catalogue of meta-model design patterns, and a pattern recommender that suggests suitable patterns from a lightweight natural language analysis of the requirements. The initial meta-model design is produced by a customizable transformation that employs the recommended patterns. We provide tool support for DSL-maps within Eclipse, integrated with DSL-tao, a tool for the pattern-based development of DSLs [12]. The integrated tools are available at http://miso.es/tools/DSLtao.html.

The rest of the paper is organized as follows. Section 2 analyses approaches to represent DSL requirements. Section 3 describes the DSL-maps notation. Section 4 explains how requirements are transformed into a meta-model. Section 5 shows tool support and a preliminary evaluation and Section 6 concludes the paper.

2. DSL REQUIREMENTS APPROACHES

How are requirements for DSLs represented? While some notations like Feature Oriented Domain Analysis (FODA) [6] have been proposed, the most common practice is to jump into a meta-model skipping an explicit representation of the requirements [10]. One cause of this situation is that building the meta-model is the starting point of most DSL construction tools.

In [10] three different types of analysis methods for DSLs are suggested: extract from code, informal and formal. In the first case, the DSL is created by generalizing an existing legacy system. In the second (and most common [10]) the domain is analysed informally. In the third case, some methodology, probably adapted from software analysis is used. The authors suggest using FODA, which requires the construction of a feature model capturing commonalities and variabilities of the domain. This way, commonalities are built-in into the DSL, which needs to offer primitives to configure the variabilities. While feature models can be a good

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970328

notation to *design* the variability, it lacks the flexibility and simplicity to gather "user wish lists", and DSL requirements in the early stage. In other cases, specialized notations, like *syntax maps* have been proposed [5]. This is a lightweight meta-modelling notation for mapping concrete syntax icons to abstract syntax classes. However, this notation is yet too close to a design. Instead notations fostering brainstorming and participation of domain experts would be more appropriate.

Looking at other sub-areas of software engineering, mind-maps [4] are increasingly recognised as an enabler of collaboration in activities like requirements engineering, test planning, to organize user wish lists, and to capture simple design choices without resorting to more complex and heavy notations like UML [3]. Mind maps have been proposed as a lightweight notation for requirements representation in combination with agile processes [9] to foster brainstorming and collaboration. Fig. 1 shows a mind-map excerpt containing some requirements for a library information system. The mind-map has a root idea (labelled as "Library Info. Systems Reqs.") from which other ideas representing increasingly refined requirements stem in branches.

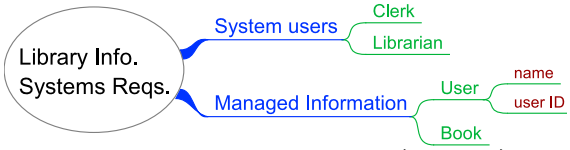

Figure 1: **Simple mind-map (excerpt)**

Mind-maps have also been employed in connection with conceptual modelling. In [14], mind-maps are used to gather requirements for software systems and then are automatically transformed into a class diagram. However, the transformation is fixed, generating an isomorphic class diagram that replaces ideas by classes and hierarchical edges by composition associations. In [15] the technique was evaluated, showing good results (in terms of time taken and quality of results) with respect to producing a conceptual schema without creating a mind-map first. The *astah* UML tool [2] supports mind maps as an informal requirements representation technique. Then, the ideas in the mind map can be manually dropped into digrams to create uses cases, or individual classes in a class diagram.

Altogether, we observe the need for notations to represent DSL requirements, which should be flexible enough to be used by all stakeholders in early phases of the DSL development. For this purpose, we will use mind-maps as a base notation, due to its popularity and flexibility. While some proposals use mind-maps as a simple software requirements notation, the level of automation towards an initial design is poor. This way, we will tailor mind-maps for DSL development and provide advanced support for the transition into a design.

3. DSL-MAPS

DSL-maps is a notation to visually gather and organize requirements for DSLs, inspired by mind-maps [4]. The objective is to foster brainstorming among stakeholders in the initial phases of DSL construction, and to organize hierarchically the different requirements along axes. It increases mind-maps with the possibility to connect ideas in different branches (to enhance flexibility and expressivity), and profits from existing knowledge for meta-model construction to automate the transition into a design (as it will be shown in next section).

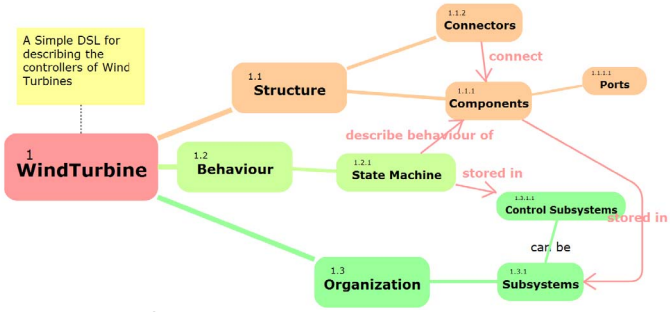

Figure 2: **DSL-map diagram for the running example**

As a running example we use a DSL inspired by one of the industrial use cases of the MONDO EU project (http://mondo.org). The DSL aims at representing the embedded software components of wind turbines. Fig. 2 shows a DSL-map with some of the requirements. The nodes in the diagram are called *ideas*. There is a *root* idea named WindTurbines representing the DSL, and three further ideas (Structure, Behaviour, Organization) stem from the root. Different branches represent the different aspects or concerns to be considered in the DSL. Hence, in this DSL, the aim is defining a structural part, via components; their behaviour, via some state-based notation; and organize these elements into subsystems. Branches are coloured differently to ease distinction between different DSL concerns. Ideas are indexed taking into account their distance to the root (Structure and Behaviour are indexed as 1.1 and 1.2, Component is labelled 1.1.2). Ideas can have attached notes (shown as yellow rectangles), which are useful to explain or emphasize some aspect of the idea. In Figure 2, the root idea has an attached label, shown as a yellow rectangle. Ideas may have further descriptions in natural language (ommited in the figure).

DSL-maps consider two types of edges: hierarchy edges and references. The former connect an idea to its parent, and can optionally be labelled. For example, the connection between WindTurbines and Structure is a hierarchical edge. Ideas and hierarchical edges form a tree structure. Hierarchy edges can be labelled. By default a hierarchy edge is interpreted as a refinement of the parent idea by the children ideas, to provide more details. In the figure, we have labelled one hierarchy edge as "can be" to convey that some subsystems may be control subsystems. Hence labelling can be used to override the semantics of hierarchy edges, and will be used by our mechanism to produce an initial meta-model design.

Reference edges may connect ideas from different branches of the DSL-map. These are directed edges and can also be labelled. Fig. 2 shows four reference edges: "connect" stating that Connectors connect Components, two "stored in" edges to convey that components are stored in subsystems, and state machines into control subsystems; and "describe behaviour of" between State machine and Component. While hierarchy edges refine an idea into more details, reference edges relate ideas in different branches of the map. Ideas may have a description and a list of features, represented as key-value

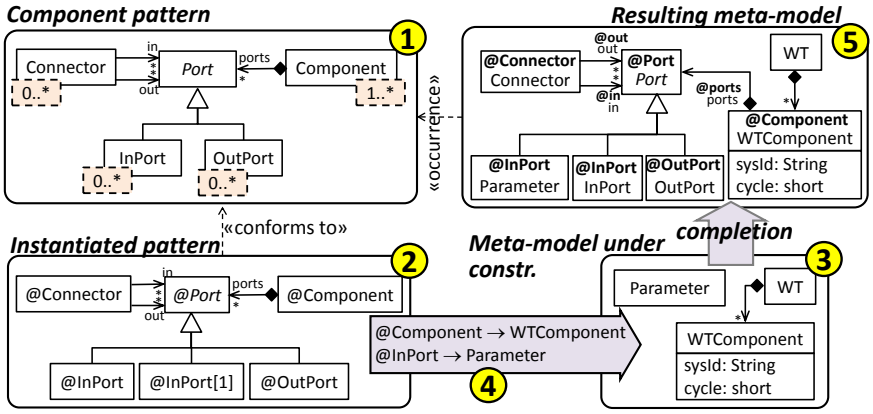

Figure 4: **Pattern application process**

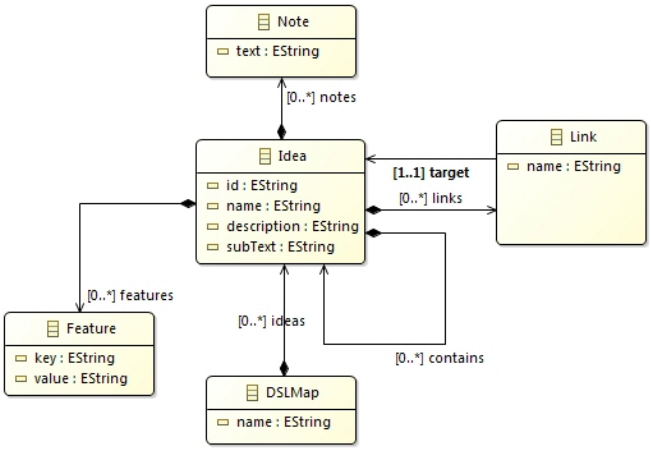

Figure 3: **Meta-model of DSL-maps**

pairs (not shown in the figure). These are used to provide further details or features of the container idea.

Fig. 3 shows the meta-model for DSL-maps with the mentioned concepts. References are modelled by the Link class, while hierarchical edges are modelled by the composition reference contains, which ensures a tree-like structure for this kind of edges. The optional subText attribute is the label of the hierarchy edge connecting an idea to its parent. Please note that we allow more than one root idea in a DSL-map.

4. TRANSITION TO DESIGN

In addition to provide a notation for DSL requirements, we automate their transition into a design. The automation is based on a repository of patterns for the development of DSLs [12], and a customizable transformation into an initial meta-model design draft. Our approach to using patterns for designing DSLs is revised in Section 4.1, the recommendation mechanism is described in Section 4.2 and the transformation is explained in Section 4.3.

4.1 Patterns for the Construction of DSLs

In [12] a pattern-based approach to develop DSLs was presented. The approach is based on the observation that some concepts are recurring across different domains. For example, it is common to use variants of state machines in different domains to describe behaviour, possibly enriched with domain-specific concepts. Similarly, many architectual languages share core concepts, like components, ports, and connectors, while information definition languages (e.g., class and entity relationship diagrams) share the notion of information class, attribute or relation. Hence, *domain patterns* represent reusable meta-model fragments, enriched with a variability. Such variability encodes optional features and variants of the pattern. Domain patterns can be embedded in a meta-model by instantiating the variability, and selecting the elements in the meta-model to which the pattern instance should be glued.

Fig. 4 illustrates the pattern instantiation and application process. Step (1) shows a pattern for component based systems, which the modeller wants to incorporate to the DSL meta-model. A pattern is just a meta-model where the elements (classes, attributes, references) are called *roles*. A role has an allowed variability, expressed as a (possibly unbounded) interval. In the figure, class role Component has allowed variability 1..*. This means that a pattern instance must contain at least one instance of the Component role. Step (2) in the figure shows a possible pattern instantiation, where every role has been instantiated once, except InPort, which has been instantiated twice. Role instances are shown preceded by a "@". Please note that pattern instantiation is performed just like meta-model instantiation.

Given a meta-model under construction (like the one in Step (3) of the figure), we can bind the role instances of the pattern instance to the meta-model elements, as shown in Step (4). In this case, we bind role @Component to class WTComponent and role @InPort to Parameter. Then, the meta-model becomes enlarged with new elements, created from the unbound role instances in the pattern instance (i.e., @Port, @Outport, etc). In the resulting meta-model, the initially bound elements and the created ones get annotated with the role instances, to signal a pattern occurrence in the meta-model.

The cardinality of pattern roles permits expressing fine grained variability for the pattern. In addition, we support another mechanism to select among a more coarse grained variability. This selection is made through a feature model, as shown in Fig. 5. This feature model contains just two options: shallow and deep. While the shallow variant is the one shown in Fig. 4 (1), the deep one accounts for the need to both define and

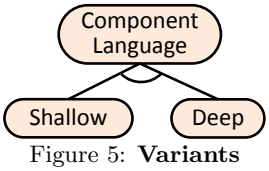

Figure 5: **Variants**

instantiate components, and includes elements like component types and instances, connector types and instances, and port types and instances.

The pattern catalogue contains domain patterns for state machines, workflows, expression languages, query languages and information definition languages, among others. The catalogue is extensible and users can add their own patterns. In addition to domain patterns, in [12] other kinds of patterns were identified, all with the same structure as a domain pattern. *Design* patterns account for meta-model design decisions, like different alternatives to represent a tree. *Concrete Syntax* patterns describe the visualization of meta-model elements, like in form of a graph, or a table. *Infrastructure patterns* configure the functionality of the final modelling environment, like filtering facilities, or model fragmentation mechanisms. Infrastructure patterns are realized via code generation services, which produce functional modules for the modelling environment, and are configured via the pattern occurrences. For example, in the case of the filtering, the pattern occurrence contains the classes to filter.

4.2 Pattern Recommendation

In order to facilitate pattern search (and to profit from a pattern recommender), we have extended the pattern definitions with a lightweight concept ontology. Hence, each pattern is related to a set of concepts, which can be connected to other concepts by general/specific relations. An small excerpt of the ontology is shown in Fig. 6. It shows a few concepts (e.g., structure, architecture, semantics) and some of their general/specific relations (shown by solid arrows). For example structure is a more general concept than module. In addition, the figure shows how two patterns are related to some of these concepts. If a pattern is related to some concept, indirectly, it is also related to all more general concepts. This way, the component pattern is related to module and architecture, and hence indirectly to structure.

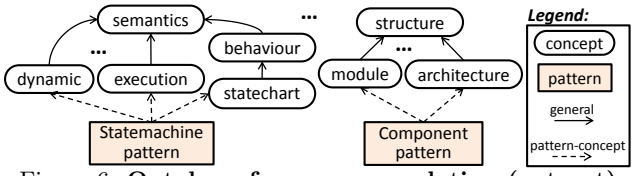

Figure 6: **Ontology for recommendation (extract)**

We profit from this ontology to recommend patterns to use in the meta-model given the ideas in a DSL-map. The purpose is that, by recommending patterns during the DSL-map construction, we will be able to automate the transition from the DSL-map into an initial meta-model draft. The recommendation is performed by a lightweight natural language analysis of the idea names and associated description, and a comparison with the concepts related (directly or indirectly) with the patterns, as well as their role names.

Hence, given an idea we want to obtain a recommendation for, we tokenize its title and description, as well as the names of the sub-ideas. Tokenization is made by considering camel-case, and in case of "-" separated words, we use as tokens both the separate and compound word. Then, words are normalized, converting them to singular using a lexical inflector. Normalized tokens are compared with: (a) the pattern names and role names within them, and (b) with the ontology terms associated with the patterns, where we check both the directly associated terms (e.g., statechart) and the more general terms. Word coincidence is checked directly, and looking for synonyms, obtained from Wordnet [11], a lexicon database for the English language. A score is calculated for each pattern variant according to the number of coincidences, where coincidences in pattern names score higher than in roles or associated tags. Similarly, coincidences in idea names score higher than in idea descriptions, or sub-idea names. The score is used to produce a ranking of suitable pattern candidates. Then, the user is in charge of choosing the different variants of the suggested pattern(s), or to discard the suggestions. Once a pattern is selected, it becomes attached to the idea. In addition, ideas up and down in the DSL-map hierarchy are inspected, so that they may become attached to *roles* of the selected pattern.

As an example, Fig. 7 shows the result of the recommendation, when the recommender is invoked over the Structure idea. In this case, the recommender lists first the Component pattern in its deep variety, followed by the shallow variant. The reasons is that the deep variant has more coincidence of role names, because the pattern meta-model has role classes to account both for types and instances (like Component/ComponentInstance, Port/ PortInstance, etc). Selecting the shallow variant, the recommender binds the Structure idea with the pattern variant, while it binds sub-ideas to pattern roles: Connectors to Connector, Components to Component and Ports to Port. The binding of roles to ideas is performed by tokenizing idea names, convertion to singular, and checking name or synonym matches.

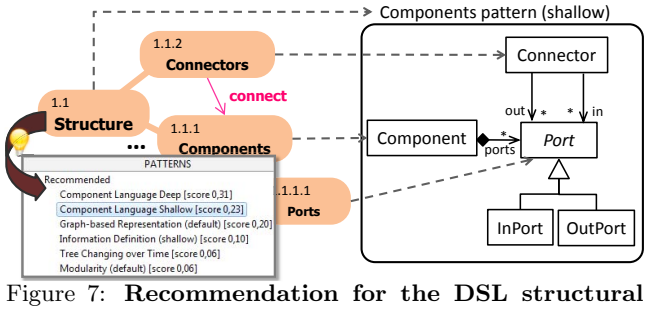

Figure 7: **Recommendation for the DSL structural part**

4.3 Synthesizing an Initial Meta-Model

Once a first version of the DSL-map has ben constructed, it can be transformed into an initial meta-model draft. This transformation takes into account the recommended patterns, uses heuristics that follow accepted quality guidelines for conceptual schemas [1] and is configurable in several steps (in the implementation these options are selected through a wizard). The transformation proceeds as follows:

1. A meta-model package is created for the DSL-map.
2. A class is created for each idea, and stored into the package. By default, class names are created in singular form.
3. By default, the transformation produces composition references between an idea and its sub-ideas. However, this may change depending on the label of the hierarchy edge. If the edge is tagged as "may-be", "can-be", or "is-a", then an inheritance relation is produced instead As an heuristic, an idea whose children ideas are all related by "may-be" is transformed into an abstract class (with all children

ideas inheriting from it). As an exception, the generated class is made concrete if it only has one subclass. This is in line with accepted conceptual schema quality issues [1].

4. If a hierarchical edge in the DSL-map is translated into a composition reference in the meta-model, some heuristics can be applied. First, there is the option to generate bidirectional references between the created parent and children classes. Regarding the cardinality of the created compositions, by default it is set to 1..1. However, if the DSL-map edge is labelled "can", "may", "may have" or "can have", the lower bound is set to 0. If the target idea is a plural name, the upper bound is set to *. Plural names are detected checking terminations and a catalogue of irregular forms.

Fig. 8 shows an excerpt of the transformation of the running example. Class names are generated in singular form. The hierarchical edge between 1.3 and 1.3.1 is transformed into a composition, while an inheritance is created for the other edge. Class Subsystem has only one child, so it remains concrete. The name of the created reference (subsystems) is the name of the sub-idea (converted into lower-case). Its cardinality is set to 1..* due to its plural name.

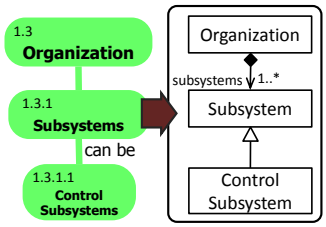

Figure 8: **Transforming hierarchy edges**

4. Relations between ideas in the DSL-map are translated into references. The heuristics for setting the cardinalities are the same as for hierarchy edges.

5. Key-value pairs attached to ideas are translated into class attributes. The type of the attribute is induced from the value of the feature. Hence, if the value is an integer, then the type is "int"; if a floating point number, then the type is "float", etc. Alternatively, there is the option to create a class for each feature.

6. Notes are transformed into meta-model annotations.

7. If some idea has a recommended pattern attached, the selected variant is produced in the generated meta-model. For this purpose, we generate classes, references and attributes for those pattern roles that were not bound to any idea. We avoid generating references in the pattern application if they have already been produced by the transformation of hierarchical edges or relations between ideas. For example, in Fig. 9 only one composition is created between Component and Port, even though both ideas are linked, and the pattern contains a composition as well.

In addition, some well-formedness checks and refactorings are needed on the generated meta-model. First, as inheritance relations are produced, the transformation takes care that no class defines and inherits two references (or attributes) with same name. If this case is detected, it performs a renaming.

Fig. 9 shows the generated meta-model from the 4 ideas of Fig. 7. The transformation has generated the pattern instance, while class Behaviour is linked to Component and Connection (through bidirectional composition references) due to the hierarchical edges between ideas. On the contrary, Component and Port are only linked once. Finally, a reference connect has been created due to the connection between ideas Connectors and Components. The cardinality of the reference is set to * due to the plural name of the Components idea.

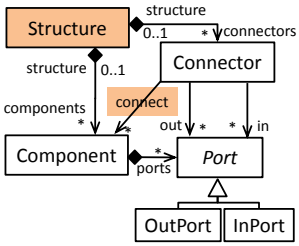

Figure 9: **Obtained meta-model excerpt**

5. TOOL SUPPORT & EVALUATION

We have built a modelling environment for DSL-maps, as an Eclipse plugin, and integrated it with DSL-tao [12]. Both tools are available at http://miso.es/tools/DSLtao.html.

Fig. 10 shows a screenshot of the tools. Label 1 shows the workspace with the analysis diagram, and a generated meta-model design. The DSL-map of the example is being shown in the canvas labelled as 2. The editor permits creating the hierarchy of ideas, links, and features as well as adding notes to the ideas. Addition of elements to the diagram is done by drag & drop from a palette. Root ideas are created dropping the idea in the canvas, while sub-ideas are created dropping the idea on top of the parent. The colour and the classification indexes are automatically handled by the tool.

Ideas have a name and description, and associated features as lists of key/value pairs, which can be edited from the property view, in the "Features" tab (label 3). The figure shows values asynchronous, synchronous as two possible values for property kind of Connectors. In the figure, we have applied the pattern assistant, which has recommended the patterns "State machine" and "Components" for some ideas in the diagram.

Once the DSL-map is ready the wizard for generating an initial meta-model can be invoked (label 4). This permits configuring some aspects of the transformation, as explained in Section 4. The generation profits from the two identified patterns, generating a meta-model with 18 classes, 1 enumeration and 6 inheritance relations. Label 5 in the figure shows a small excerpt of the generated meta-model, which shows an enumerate type for the property kind. The view with label 6 displays the two applied patterns. Clicking on them highlights their occurrence in the meta-model.

We conducted a preliminary evaluation with 7 engineers, to better understand the perceived strengths and weaknesses of DSL-maps. Subjects ranged from novices to experienced MDE developers, with a median experience of 3-6 years. None had used before a specific notation for DSL requirements, which stresses the need for a requirements notation like DSL-maps. Understandability of DSL-maps was valued as high (average 4.15 out of 5). This was confirmed by 3 questions, obtaining 20 correct and 1 incorrect answer.

Then, subjects were asked to build a meta-model out of the DSL-map in Fig. 2, and all produced a meta-model close to the one produced by the tool without the pattern assistant. However, 6 of them had at least some minor quality issue (wrong cardinalities or missing compositions). This shows the benefits of the automated transformation. None of the participants was able to produce a solution with the

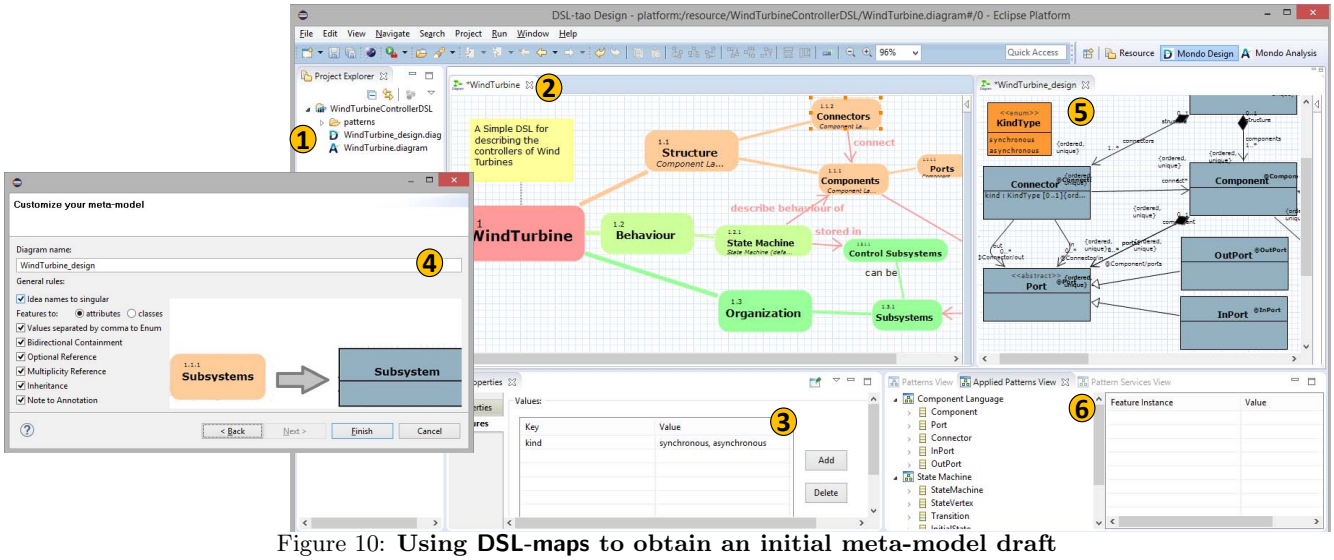

Figure 10: **Using DSL-maps to obtain an initial meta-model draft**

level of detail provided by the pattern assistant, which was rated as very complete (4.3 out of 5) and useful (4 out of 5). This suggests that the knowledge stored in the patterns could be valueable for the developer. Finally, suggestions were collected. One participant proposed using the same colour in meta-model classes as in the DSL-map ideas, to improve traceability. Another participant suggested the ability to generate meta-model packages from some ideas (e.g., from Structure and Behaviour).

6. CONCLUSIONS & FUTURE WORK

We have presented DSL-maps, a notation inspired by mind-maps to represent DSL requirements. The notation emphasises flexibility, and promoted brainstorming among the involved stakeholders. The approach supports the automated transition into a design by profiting from a library of meta-modelling patterns and a recommender. DSL-maps is backed by a tool, integrated within DSL-tao.

We are considering interoperability of our tool with mind-mapping tools. In addition to class colouring, we are working on traceability mechanisms between the meta-model and the DSL-map. This will avoid overwriting manual changes in the meta-model when the DSL-map evolves.Finally, we are improving the transformation to permit generating meta-model packages from the ideas.

7. ACKNOWLEDGEMENTS

Work supported by the Spanish Ministry of Economy and Competitivity with project Flexor (TIN2014-52129-R), the EU commission with project MONDO (FP7-ICT-2013-10, #611125) and the Madrid Region with project SICOMORO (S2013/ICE-3006).

8. REFERENCES

[1] D. Aguilera. *A method for the unified definition and treatment of conceptual schema quality issues*. PhD thesis, Uni. Politecnica Catalunya, 2014.

[2] Astah. `http://astah.net`.

[3] A. Binstock. Mind maps: The poor man's design tool. *Dr Dobbs*, October, 2012.

[4] T. Buzan. *The Mind Maps Book*. Dutton, 1993.

[5] H. Cho, J. Gray, and E. Syriani. Syntax map: A modeling language for capturing requirements of graphical DSML. In *APSEC*, pages 705–708, 2012.

[6] K. C. 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, 1990.

[7] S. Kelly and Tolvanen. *Domain-Specific Modeling. Enabling Full Code Generation*. Wiley, 2008.

[8] 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.

[9] C. Larman. *Agile and Iterative Development: A Manager's Guide*. Ad.-Wesley Prof., 1993.

[10] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[11] G. A. Miller. Wordnet: A lexical database for english. *CACM*, 38(11):39–41, 1995.

[12] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara. Pattern-based development of domain-specific modelling languages. In *MoDELS*, pages 166–175, 2015.

[13] M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[14] F. Wanderley, D. Silveira, J. Araújo, and Moreira. Transforming creative requirements into conceptual models. In *RCIS*, pages 1–10, 2013.

[15] F. Wanderley, D. Silveira, J. Araújo, A. Moreira, and E. Guerra. Experimental evaluation of conceptual modelling through mind maps and model driven engineering. In *ICCSA*, volume 8583 of *LNCS*, pages 200–214, 2014.

[16] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.