A Pattern-based Approach to DSL Development

Christian Schäfer
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
christian.schaefer@iese.fhg.de

Thomas Kuhn
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
thomas.kuhn@iese.fhg.de

Mario Trapp
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
mario.trapp@iese.fhg.de

Abstract

Tool support for the development of Domain-specific Languages (DSLs) is continuously increasing. This reduces implementation effort for DSLs and enables the development of rather complex languages within reasonable amounts of time. However, the lack of commonly agreed and applied language engineering processes, many times turns DSL development into a set of creative activities, whose outcomes depend on the experience of the developers involved. Consequently, outcomes of language engineering activities are unpredictable with respect to their quality, and are often not maintainable either. We have therefore developed an approach that transfers the concept of architecture and design patterns from software engineering to language development. In this paper, we propose this approach and evaluate its applicability in a case study.

Categories and Subject Descriptors D.2 [*Software*]: Software Engineering

General Terms Design, Languages

Keywords Domain-Specific Languages, DSL Development, Language Patterns

1. Introduction

Domain-specific Languages (DSLs) enable experts to express domain concepts in their problem domain, which is the domain that they are working in. DSLs are much more specialized and domain-tailored than general-purpose modeling languages or UML profiles and provide an intuitive façade to solution domains, which is the domain that a solution is implemented in. This is imperative in todays complex environments, where engineers become more and more specialized. Engineers that are capable of working in complex problem domains, e.g. hydraulics, are often not capable of transforming their solution efficiently to a solution domain, e.g, the C program for a 8-Bit microcontoller. This results in error-prone, expensive, and lengthy development cycles. DSLs address this situation by providing tailored languages for experts, and by hiding unnecessary details of solution domains. Code for particular solution domains is automatically generated through model transformations. This significantly increases the productivity of domain experts, and enables portability to other solution domains at the same time.

DSLs are no new topic in computer science. Better availability of technology that supports language development nowadays makes them applicable for a broader scope of applications. Therefore, the development of DSLs has become economical for smaller projects as well. Especially the increasing capabilities and configurability of modern systems provide areas for Domain-specific Languages; a good number of our past DSL projects yielded configuration languages that configured systems, system variants, and embedded behavior on a high level. However, in the past, research has focused on the technological aspects of Domain-specific Languages as well as on their visual presentation. Methodological language engineering is still an uncovered spot. This yields quality problems with respect to the developed languages; a trend that also hits uncontrolled software development. Existing studies indicate that about 30% of developed DSLs are too narrow or too broad to be useful, or that about 40% of all developed languages do not reflect the processes that they will be applied in (cf. [8])

We therefore developed a DSL development approach that transfers well-proven concepts of software engineering to language development. In particular, we use the concept of design patterns and traceability to improve language development. Design patterns capture language development best practices by conserving proven solutions and documenting their proper application. Traceability ensures that important decisions are documented and links parts of the language to the requirements that they originate from. Both do not only increase language quality, but also enable developers to better understand DSL metamodels, which both enables quality control activities and maintenance activities. Based on a real-world example, we show how our methodology supports language developers in creating high-quality languages.

The remainder of this paper is structured as follows: Section 2 surveys related work and provides a brief overview of existing tool chains for language implementation. Section 3 describes the fundamental principles of pattern-based DSL development and traceability. Section 4 illustrates our approach in the context of a real-world example. Section 5 draws conclusions and lays out future work.

2. Related Work

Even though engineering approaches to language development have not been heavily addressed by recent research activities, there exists a body of knowledge that needs to be discussed:

The authors of [8] discuss worst practices for DSL development that language developers should avoid. The study was based on several real-world DSL development projects; the mentioned bad practices were applied within these projects with a specific probability. Even though most of these projects were related to the DSL tool MetaEdit+, most of the discussed bad practices are tool-independent and do lower the quality and acceptance of DSLs created with them. Therefore, during the development of our DSL development approach, we put special emphasis on the construction-related worst practices listed in [8], and included means to avoid them. The findings of [8] are substantiated by the findings published in [16] and [7], which especially confirm the importance of language focus, role-specific requirements, and the need to avoid over-complete and overloaded languages.

The work presented by [10] lists DSL development approaches and compares them with each other. The authors structure DSL development into different phases and identify various patterns for each of these phases. Moreover, they provide qualified decision guidelines for language developers. In the work published in [15], approaches for the design and implementation of DSLs are analyzed and again recurring patterns are derived. However, in both publications the identified patterns are on a much higher abstraction level than the patterns that we have defined. They are therefore rather meant for categorizing different DSL development approaches.

In [4], the authors present an approach for a pattern specification language that is capable of specifying patterns for MOF-compliant metamodels. They also demonstrate how to derive a pattern detection algorithm from a given pattern specification, which is helpful for specifying and detecting anti-patterns such as the ones maintained in [3].

The authors of [9] present an approach for metamodel composition that is also applied to create UML-based Domain-specific Languages. Similar to our approach, several specialized composition operators are defined to support metamodel composition. The work presented in [5] proposes an approach based on language templates, which is similar to our pattern-based DSL construction approach. Here, a novel metamodel composition method called template instantiation is applied by the authors. In [13], the authors provide a methodology and a framework for defining the semantics of a DSML using a compositional approach. The authors define the notion of domain concepts, which attach semantic information to metamodels, and then show how these domain concepts may be combined to build a new DSL. In [14], this work is extended to also take the concrete syntax into account.

The work presented in [14] also describes an approach for DSL development, which is focused more on a formal backend than on guiding developers towards creating high-quality languages. This approach might be used as a technical foundation to which the concepts described in this work can be applied.

We therefore conclude that DSL concepts are developed by experts in creative processes whose outcomes depend on skills and experience. Results are hardly predictable and not repeatable. Even worse, known bad practices are applied in numerous ongoing language development processes due to inexperienced language developers and missing guidelines.

In addition to methodological approaches, tool support plays an important role in language development. In the following, we therefore provide a brief overview of various tools for DSL development.

Within the Eclipse platform, there exist a couple of plugins that support DSL development: The Eclipse Modeling Framework (EMF), for example, allows for easily defining domain-specific metamodels and basic infrastructure. Using the Graphical Modeling Framework (GMF), one can easily define graphical editors on top of the EMF model. For textual DSLs, the Xtext Framework provides means for creating the language syntax and sophisticated text editors. Other DSL-related plugins can be found in the Eclipse Modeling Project (EMP) [6].

MetaEdit+ is a commercial tool for building and using domain-specific modeling solutions [11]. The focus of MetaEdit+ is on the development of graphical, graph-based DSLs with optional code generation. It comes with its own proprietary metamodeling language for the definition of a DSL abstract syntax. A key distinctive feature is the symbol editor for the development of a concrete language syntax, which lets one easily draw DSL elements.

MagicDraw is a commercial UML modeling tool [12], which provides further capabilities for creating DSLs through its built-in DSL customization engine. In contrast to the aforementioned tools, rather than using plain metamodeling, the user builds a DSL on top of the UML using UML profiles. With the provided DSL engine, UML properties can be hidden, own symbols can be used, and new diagram types can be defined. As the underlying model is still a UML model, it can be exported to and processed by any other UML-related tool.

3. DSL Development

Our approach applies traceability and language patterns to support controlled language development. Language patterns ensure that proven and good solutions are used to create the DSL metamodel. Traceability from requirements to pattern application ensures that all language requirements are reflected in the DSL metamodel. It indicates whether all requirements are addressed, and that no superfluous language elements have been added to the DSL. Traceability also makes pattern application and therefore decisions regarding the metamodel explicit. While both approaches do not guarantee quality languages alone, they support language developers in creating well-formed, focused, and maintainable languages.

3.1 Requirements and Traceability

Formalization of language requirements and traceability support is realized through a model similar to use-case models of UML2. The model captures activities that describe overall tasks of the DSL. Each task is substructured into activities that are performed by persons, which are abstracted through roles. Roles define persons with common knowledge and duties regarding the DSL, the problem domain, or the solution domain. Each task uses a certain set of language elements that are found during elicitation meetings and potentially updated during language development. An example of such a requirements model is shown in Figure 4. Language patterns are then applied to create a DSL out of these language elements, e.g. by defining connections, containments and other modeling approaches. Each language pattern is traceable to a set of language elements and tasks from the requirements model (cf. Figure 5). This enables tracing of decisions regarding DSL development to concrete requirements and prevents superfluous concepts.

3.2 Language Patterns

In software engineering, patterns are reusable entities that conserve proven concepts. They need to be adapted and integrated into new contexts, and therefore require more effort than integration of pre-existing components. However, they also provide more flexibility. This fits perfectly to DSL development, because these languages need to be closely tailored to problem domains and domain experts. A pattern supporting development of data flow languages could conserve domain independent aspects of data flow, and still be adapted to concrete language needs. Language patterns therefore always need to be applied to a context, which consists of selecting one pattern out of a pattern pool, adapting it to the given context, and integrating it into an existing environment (e.g., an existing metamodel).

Each pattern has to provide particular information that support their application. This includes, e.g., their name, the problem to be solved, the provided solution, and the rationale behind the pattern. Each pattern also has to provide context information defining the contexts a particular pattern may be embedded into. In order to provide the necessary information in a structured way, there exist a couple of different notation styles [2], where each style basically contains the same information, but puts emphasis on different aspects of a pattern. For the language patterns, we choose and adapt the form described by J. Coplien:

- Name: A unique and descriptive name of the language pattern.
- Problem: A description of the design challenge. This may be described, e.g., in the form of typical application scenarios.

- Context: The context that the pattern is embedded into and the prerequisites that have to be met for being applicable.
- Forces: The fundamental principles that clarify and help in understanding the structure of the problem, also pointing out design trade-offs between different solutions.
- Solution: A description on how to solve the given problem. This typically contains a description of an excerpt of the language's resulting metamodel.
- Rationale: A detailed explanation for the provided solution and why it solves the issues of the problem.
- Related Patterns: Names of related patterns that provide solutions for the same or similar problems.

As an example we define the pattern "Unnavigable Nodes and Links" (cf. Figure 1), which provides a solution for the very common situation where two elements have to be connected by a relation element. The pattern provides the necessary information in the form described above. In particular, the solution section shows how to apply the pattern for deriving the respective part of a language's metamodel. Thus language patterns support the creation of DSL metamodels by taking an input metamodel and transforming it into an output metamodel. The language metamodel is created by a series of these transformations.

For the solution section of the pattern, we use our own formalism to describe how to proceed from a given source metamodel (the context of the pattern) and transform this into a new output metamodel with modified or possibly newly created metaclasses. This is done in two steps: First, an intermediate metamodel that consists of metaclasses for language concepts and language elements is created. Second, to obtain the final metamodel, the intermediate metamodel is stripped of language concepts to only contain metaclasses for language elements. In our example, the language pattern defines the concepts of *Nodes* and *Links* as two separate *Entities*, where *Entity* is a context concept already defined in the source metamodel. Each *Link* defines two associations to one or two *Node* elements: One association is named *from*, the second association is named *to*. In general, language concepts represent abstract concepts that are introduced by the pattern. For example, they describe how a relation, a substructure, or a complete condition monitoring system is modeled without defining the concrete elements. The concrete language elements (like, e.g., *Wire* and *Sensor*), which are specific to the DSL under development, are then introduced as metaclasses and specialize these concepts. The separation of a metamodel into concepts and language elements supports the definition of pattern contexts that document to which language elements a pattern is to be applied, and which language elements will be changed by the pattern.

As stated above, we defined a new formalism for the description of the language pattern's solution. The basic elements of this formalism are depicted in Figure 2, which are the name of the pattern, context concepts, language element concepts (either Singleton or Multiton), and various different types of relations (Generalization, Association, Aggregation).

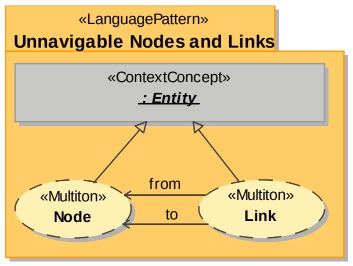
Pattern name:
Unnavigable Nodes and Links

Problem:
Language entities need to be connectable.

Context:
Language entities have already been created, but are still decoupled.

Forces:
Many (graphical) Domain-specific Languages built up on interconnected nodes, with nodes and links being the most general abstraction level.
Navigability between nodes is not an issue, i.e., nodes themselves do not need to know about connected nodes.

Solution:
Existing language entities are specialized into nodes and links. Links reference their source and destination node. Nodes have no references to links.



Rationale:
Specialization of entities into links and nodes clarifies their roles in the language, which can also give hints regarding preferable entity presentations (e.g., circles for nodes and lines for links).
Nodes do not need to know about how they are connected. It is sufficient to store this information only within the links.

Related Patterns:
Navigable Nodes and Links

Figure 1. Pattern "Unnavigable Nodes and Links"

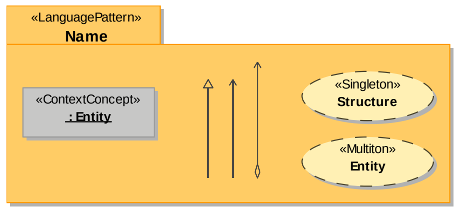


Figure 2. Elements of Language Patterns

3.3 DSL Meta Models

Formally, a metamodel $M = (s_{name}, C, R_{Gen}, R_{Ass}, R_{Agg})$ is defined as a five-tuple. The property s_{name} contains a primitive value describing the name of the metamodel. C is a set with all metaclasses of the metamodel. A metaclass $c \in C$ is defined as a two-tuple (s_{name}, A) consisting of the class name and a set of attributes A. Attributes $a = (s_{name}, T, n_{min}, n_{max}) \in A$ consist of a name, a primitive type $T \in \{String, Integer, Bool, Float\}$, and of a minimum and maximum multiplicity n_{min} and n_{max}. Here, a value of -1 represents unlimited multiplicity. R_{Gen}, R_{Ass}, and R_{Agg} represent generalization, association, and aggregation relations. Generalizations $r \in R_{Gen} : (C \times C)$ are represented by a pair of metaclasses. A particular generalization $r = (c_1, c_2)$ therefore links metaclass c_1 to metaclass c_2. The first metaclass is the specialized element, the second element is the general element. Associations $r = (s_{name}, n_{min}, n_{max}, c_1, c_2) \in R_{Ass}$ additionally define three primitive values containing the name of the association, and its minimum and maximum multiplicities. Aggregations $r = (s_{name}, n_{min}, n_{max}, c_1, c_2) \in R_{Agg}$ are defined in a similar manner.

Metaclasses are abstract language elements. Concrete language elements, which are the elements that language users interact with, additionally contain presentations. Thus, a concrete language element $p = (c, pr)$ is a relation that links a metaclass c to a presentation pr. The set P contains all concrete language elements of a

DSL. A view V_n is a set of concrete language elements that belong together. Views thus present parts of the information that is stored in a model of the DSL in a way that supports one domain-specific activity. A DSL usually contains several views V_n, which address different activities and the needs of different roles. The views together with the concrete language elements form the concrete syntax of the DSL. All views V_n of a DSL are collected in the set V. Transformation rules for one solution domain $t \in T_n$ represent transformations from the problem domain of a DSL into one solution domain. A language may support multiple sets of transformations T_n for several solution domains. The set $T = \{T_n\}$ contains all transformation sets of a DSL for all solution domains. A Domain-specific Language $L_{DSL} = (M, P, V, T)$ is therefore defined as a tuple consisting of a metamodel, presentations of language elements, views, and transformations. In this paper, we will focus on the construction of the metamodel with the help of our language patterns.

3.4 Pattern Application

The overall procedure for applying a pattern to a given input metamodel is described in Figure 3. First of all, all language element concepts defined within the pattern are added to the metamodel as abstract metaclasses. For the application of our example pattern from Figure 1, this would mean that the language element concepts *Node* and *Link* are added to the metamodel. In the next step, all generalization relationships defined within the pattern are established in the metamodel. With the "*Unnavigable Nodes and Links*"-pattern, this leads to adding the generalization relationships between *Node* and *Entity* and between *Link* and *Entity* to the metamodel. In the third step, missing language elements are added. This depends on the concrete language that has to be developed. For example, a DSL for a condition monitoring system needs to model *Sensors* and *Wires* between them, which would thus be added as metaclasses to the metamodel if they are not already contained. The fourth step then connects those concrete language elements to the language element concepts originating from the pattern by creating generalizations between them. Hence, the language element *Sensor* would specialize the concept *Node* and a *Wire* would specialize *Link*. The fifth step is then to simplify the generalization hierarchy by removing unnecessary generalization relationships. For example, a *Sensor* might be already contained in the metamodel with a generalization relationship to the language element concept *Entity*. As *Nodes* also specialize *Entities*, the generalization between *Sensor* and *Entity* can be deleted, maintaining only the generalization to *Node*. The last step in applying a pattern is to establish all the associations and aggregations between language elements according to the specifications in the pattern. This would, e.g., create an association named *from* between the language elements *Wire* and *Sensor* in the metamodel.

It is important to note that the application of a pattern is not a fully automated process that would require no additional input from the language developer. In fact, the language developer needs to think about which elements of the language are supposed to be affected by the pattern. For example, he needs to choose which language elements need to be created, how they specialize the language element concepts from the pattern, and how relationships defined in the pattern are adapted between them. However, by using the patterns and following the aforementioned process, the impact of design decisions becomes much more obvious to the language developer, raising the overall quality of the resulting language significantly.

In the following, we formally define the steps that were presented informally in the previous section, which enables, e.g., better tool support for processing and modification of metamodels as well as for verifying the results of the individual steps. Here, the definitions for metamodels and patterns as described in the previous section apply.

1. Add language element concepts as abstract metaclasses to the metamodel
2. Establish generalization relationships between language element concepts from step 1 to already existing language element concepts
3. Add missing language elements as metaclasses to the metamodel
4. Establish generalization relationships between language elements to the language element concepts introduced in step 2
5. Simplify the inheritance hierarchy by removing superfluous generalization relationships
6. Establish association and aggregation relationships between language elements

Figure 3. Pattern Application Algorithm

The application of a pattern P to an input metamodel M starts by adding P's language element concepts to M. The result is again a metamodel. This is described by the function

$$f_{addConcept}(P,M) := (s_{name}, C \cup L_P \cup L_S, R_{Gen}, R_{Ass}, R_{Agg})$$

which adds the sets of language element concepts L_P and L_S defined in P to the set of metaclasses C defined in M, which is possible since language element concepts themselves are abstract metaclasses. In the following, we write C' for this new set of metaclasses. The other parts of M remain unchanged.

In the next step, the generalization relationships of P are added to M, which is performed by the function

$$f_{addPatternGens}(P,M) := (s_{name}, C', R_{Gen} \cup RP_{Gen}, R_{Ass}, R_{Agg})$$

This is only valid if the set of context elements defined in P is also part of the set of metaclasses defined in M, thus if $K \in C$ holds true. If this is not the case, the application of the pattern to M is not allowed, which can be checked as a prerequisite prior to its application. The union of R_{Gen} and RP_{Gen} is written as R'_{Gen}.

In the next step, possibly missing language elements are added to the set of metaclasses in M. These elements come from the requirements of the DSL and will fill in the roles envisioned by the language element concepts of the pattern. Of course, M might already contain the necessary language elements, in which case no further elements need to be added.

In the fourth step, generalization relations between elements and concepts are added to M, which clarify the roles of language elements with regard to the pattern's language concepts. This is described by the function

$$f_{addGens}(P, M) := (s_{name}, C', R'_{Gen} \cup G, R_{Ass}, R_{Agg})$$

where $G : C \times L$ is a set of generalization relationships that specialize language element concepts from the pattern. The union of R'_{Gen} and G is written as R''_{Gen}. As C might contain language element concepts from previous patterns, adding generalizations might lead to illegal cycles in the inheritance hierarchy. In order to detect such cycles, we do not allow adding any generalization relation that would allow for an inheritance path in M other than the ones defined in the set

$$\begin{aligned} H_{R''_{Gen}} = \big\{ & (r_1, r_2, ..., r_n) \mid r_i = (r_{i,1}, r_{i,2}) \in R''_{Gen, i \in \{1,...,n\}}, \\ & \wedge \forall j \in \{1, \dots, n-1\} : r_{j,2} = r_{j+1,1} \\ & \wedge \forall k \in \{1, \dots, n\} : \forall m \in \{1, \dots, k\} : r_{k,2} \neq r_{m,1} \big\} \end{aligned}$$

The fifth step deals with the simplification of the inheritance hierarchy by removing generalization relationships that add no further information. That is, we delete any generalization relationship $r = (r_1, r_2)$ from R''_{Gen}, for which the predicate

$$\begin{aligned} p(r) := \exists h = (h_1, h_2, \dots, h_n) \in H_{R''_{Gen}} : \\ n \geq 2 \wedge h_{1,1} = r_1 \wedge h_{n,2} = r_2 \end{aligned}$$

holds true, thus leading to the final generalization set R'''_{Gen}.

In the final step of the pattern application process, associations and aggregations are added to the metamodel. So, for an association relationship $r_p = (s_{name}, n_{min}, n_{max}, x_1, x_2) \in RP_{Ass}$ corresponding associations $r = (s_{name}, n_{min}, n_{max}, c_1, c_2)$ where $(c_1, x_1), (c_2, x_2) \in R'''_{Gen}$ are created and added to the sets R_{Ass}. Accordingly, aggregation relationships are created and added to the set R_{Agg}. For the new sets, we write R'_{Ass} and R'_{Agg}, respectively, so that in the end, after applying a pattern, the new metamodel is defined as follows:

$$M' = (s_{name}, C', R'''_{Gen}, R'_{Ass}, R'_{Agg})$$

With subsequent application of several patterns, the metamodel will contain a lot of language concepts that may bloat it up and thus make it more complicated to be understood. Therefore, it makes sense to strip it of these. So, let X be the set of all language element concepts in a metamodel M and $R_X = \{(r_1, r_2) \in R_{Gen} \mid r_1 \in X \vee r_2 \in X\}$ be the set of all generalization relationships coming from or going to a language element concept. Then the function

$$f_{strip}(M) := (s_{name}, C \setminus X, R_{Gen} \setminus R_X, R_{Ass}, R_{Agg})$$

removes all pattern-related parts in M. However, one should keep in mind that once this function is executed, application of further patterns is more difficult, as required context concepts of a pattern are now missing and first need to be added to the metamodel manually.

4. Case Study

We applied the proposed pattern-based development approach in several projects. One of them was the development of a DSL for a real-world Condition Monitoring System (CMS). This project was conducted together with a small mechanical engineering company and therefore shows the applicability of our approach in a real-world industrial context. The specifics of the developed system are detailed in [1].

4.1 Requirements Model

Development of DSLs is similar to software development to some extent: first requirements are elicited, then subsequent development and quality assurance stages are executed. For DSL development, we first model each language requirements as shown in Figure 4. This model defines DSL requirements, language users, and an initial set of language elements. During language development, this model is extended; each concrete language element must be traced to at least one requirement of this model. This ensures absence of superfluous elements. Language patterns will be traced to language requirements as well to provide traceability of metamodeling concepts to requirements.

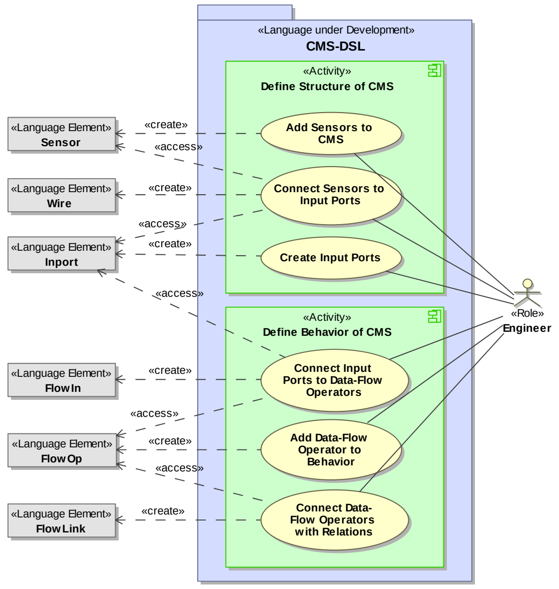

«Language under Development»
CMS-DSL
«Activity»
Define Structure of CMS
Add Sensors to CMS
Connect Sensors to Input Ports
Create Input Ports
«Activity»
Define Behavior of CMS
Connect Input Ports to Data-Flow Operators
Add Data-Flow Operator to Behavior
Connect Data-Flow Operators with Relations
«Language Element» Sensor
«Language Element» Wire
«Language Element» Inport
«Language Element» FlowIn
«Language Element» FlowOp
«Language Element» FlowLink
«create»
«access»
«Role» Engineer

Figure 4. Use Cases and Language Elements identified in Phase 2

4.2 Metamodel Construction

The existing requirements model already defines concrete language elements. However, right now, these are unstructured and it is not yet clear which of them are going to be included in the DSL metamodel and how. Also, relationships between language elements are not yet defined and additional auxiliary language elements might still be missing. These aspects will now be dealt with the construction of the DSL metamodel.

Metamodel construction starts with an empty metamodel $M =$ (“CMS”$, \emptyset, \emptyset, \emptyset, \emptyset)$. The first step now is to add the *Entity* language element concept. This can be seen as applying a very simple, initial pattern that defines no further context elements and only introduces this most general kind of language element concept. All language elements that are added to the metamodel at some point should directly or indirectly specialize this concept. Because of its simplicity, only steps 1, 3, and 4 of the pattern application algorithm (cf. Figure 3) have an effect. Hence, after adding the *Entity* concept, the already identified language elements are added and the specialization relationships to it are established. Formally, the metamodel M is then defined as follows:

$$\begin{aligned} M = (&\text{“CMS”}, \{Entity, Sensor, Wire, Inport, FlowIn, FlowOp, FlowLink\}, \\ & \{(Sensor, Entity), (FlowOp, Entity), (FlowIn, Entity), \\ & (Inport, Entity), (Wire, Entity), (FlowLink, Entity)\}, \emptyset, \emptyset) \end{aligned}$$

The entity pattern includes language elements in the metamodel of the DSL, but since it is a very generic pattern, it typically does not support direct realization of any particular requirement other than the generic presence of language elements. More complex patterns in subsequent steps will be related directly to requirements. For the CMS, for example, we apply the pattern “*Unnavigable Nodes and Links*” next, which was described in subsection 3.2 and which supports realization of three requirements (cf. Figure 5). The pattern adds the language concepts *Nodes* and *Links* and is applied to several language elements, transforming them into either *Nodes* or *Links* by adding the respective generalizations. In addition, association relationships coming from *Links* are established between language elements. A graphical representation of the resulting metamodel is depicted in Figure 6. It shows the inheritance hierarchy between language elements and language concepts and the association relationships between elements.

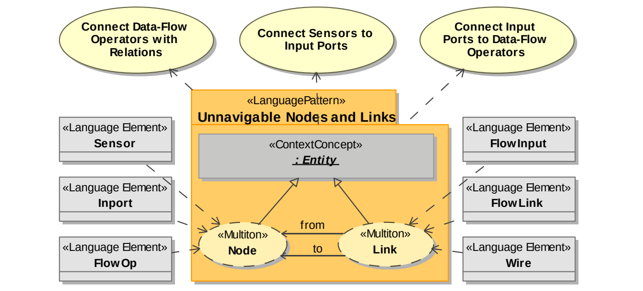

Connect Data-Flow Operators with Relations
Connect Sensors to Input Ports
Connect Input Ports to Data-Flow Operators
«LanguagePattern» Unnavigable Nodes and Links
«ContextConcept» : Entity
«Multiton» Node
«Multiton» Link
from
to
«Language Element» Sensor
«Language Element» Inport
«Language Element» FlowOp
«Language Element» FlowInput
«Language Element» FlowLink
«Language Element» Wire

Figure 5. Application of "*Unnavigable Nodes and Links*"

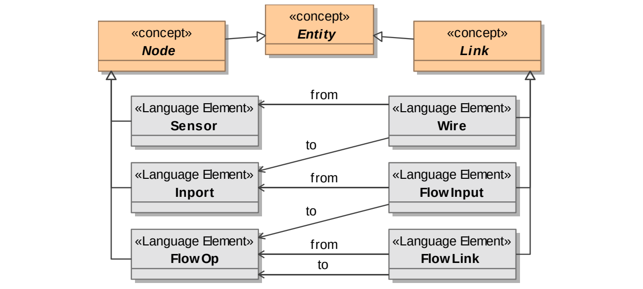

«concept» Node
«concept» Entity
«concept» Link
«Language Element» Sensor
«Language Element» Wire
«Language Element» Inport
«Language Element» FlowInput
«Language Element» FlowOp
«Language Element» FlowLink
from
to

Figure 6. Metamodel after "*Unnavigable Nodes and Links*"

Application of other patterns finally yield the complete meta model. Each step of the meta model construction can be traced to the application of a pattern that provides a concept, and to a requirement that requires that concept, as well as the concrete language elements that were affected by that concept. Figure 7 shows the complete DSL metamodel.

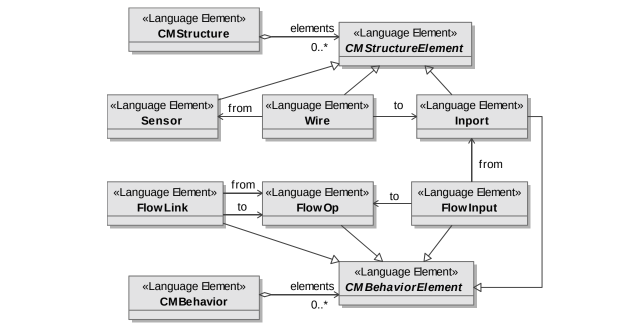

«Language Element» CMStructure
elements
0..*
«Language Element» CMStructureElement
«Language Element» Sensor
«Language Element» Wire
«Language Element» Inport
from
to
«Language Element» FlowLink
«Language Element» FlowOp
«Language Element» FlowInput
«Language Element» CMBehavior
«Language Element» CMBehaviorElement

Figure 7. Final Metamodel of the CMS

5. Conclusion

In this article, we have presented a systematic development approach for Domain-specific Languages. Our approach is built upon the use of language patterns, which provide proven solutions to common metamodeling problems. We showed how the language patterns can be defined and applied in a way that allows an iterative development of high-quality DSL metamodels. Furthermore, we demonstrated how the patterns support traceability of design decisions regarding the metamodel to their originating requirements. With the formal foundations for the definition of metamodels, patterns, and their application on metamodels, we laid the ground for tool supported processing of metamodels. This may assist the language developer in applying the patterns and in facilitating their usage. We successfully applied our approach in several case studies, one of them being the development of a DSL for Condition Monitoring Systems. The results we achieved also show that our approach is ready to be used in real-world industry projects.

In future, we plan to extend our approach to also integrate patterns for the definition of DSL presentations and semantics. From providing mechanisms to tightly integrate patterns for these different DSL facets, we expect to achieve a high speed-up and quality increase in DSL development. By integrating more and more patterns we strive for the definition of pattern catalogs or pattern languages, which may support creation of DSLs in many different domains.

References

[1] D. Barkowski, T. Kuhn, C. Schäfer, and M. Trapp. Domain-Specific Modeling as an Enabling Technology for Small and Medium-sized Enterprises. *Proceedings of the 10th Workshop on Domain-Specific Modeling*, pages 13–18, 2010.

[2] J. O. Coplien. *Software Patterns*. SIGS, 2000.

[3] M. Elaasar, L. C. Briand, and Y. Labiche. Metamodeling Anti-Patterns, 2010. URL `https://sites.google.com/site/metamodelingantipatterns`.

[4] M. Elaasar, L. C. Briand, and Y. Labiche. A Metamodeling Approach to Pattern Specification and Detection. Technical Report SCE-06-08, Carleton University, March 2006.

[5] M. Emerson and J. Sztipanovits. Techniques for Metamodel Composition. *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006.

[6] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009. ISBN 978-0-321-53407-1.

[7] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design Guidelines for Domain Specific Languages. *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[8] S. Kelly and R. Pohjonen. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009.

[9] A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti. On Metamodel Composition. *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA '01).*, 2001.

[10] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[11] MetaCase. Domain Specific Modeling with MetaEdit+, 2011. URL `http://www.metacase.com`.

[12] No Magic, Inc. MagicDraw UML, 2010. URL `http://www.magicdraw.com`.

[13] L. Pedro, V. Amaral, and D. Buchs. Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach. *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, 2008.

[14] L. Pedro, M. Risoldi, D. Buchs, B. Barroca, and V. Amaral. Composing Visual Syntax for Domain Specific Languages. *Proceedings of the 13th International Conference on Human-Computer Interaction. Part II: Novel Interaction Methods and Techniques*, pages 889–898, 2009.

[15] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56,(1):91–99, 2001. ISSN 0164-1212.

[16] D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51(3):265–290, 2004.