Electronic Communications of the EASST
Volume 25 (2010)

Proceedings of the Workshop
Visual Formalisms for Patterns
at VL/HCC 2009

An Active Pattern Infrastructure for Domain-Specific Languages

Tihamer Levendovszky and Gabor Karsai

10 pages

Guest Editors: Paolo Bottoni, Esther Guerra, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Active Pattern Infrastructure for Domain-Specific Languages

Tihamer Levendovszky[1] **and Gabor Karsai**[1]

[1]tihamer, gabor@isis.vanderbilt.edu
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37205, USA

Abstract: Tool support for design patterns is a critically important area of computer-aided software engineering. With the proliferation of Domain-Specific Modeling Languages (DSMLs), the adaptation of the notion of design patterns appears to be a promising direction of research. This paper introduces a new approach to DSML patterns, namely, the Active Model Pattern infrastructure. In this framework, not only the traditional insertion of predefined partial models is supported, but interactive, localized design-time manipulation of models. Optionally, the infrastructure can be adapted to handling transactional tracing information as well as transactional undo and redo operations. Possible realizations of the framework are also discussed and compared.

Keywords: Design Patterns, Domain-Specific Modeling Languages,

1 Introduction

The use of design patterns [GHJV95] has brought revolutionary changes to object-oriented (OO) software design and development. Design patterns are reusable solutions to recurring design problems. This definition often implies intuitive techniques that can be very hard to express and document by models or program code. However, several patterns can be successfully represented as incomplete models that will be inserted into complete models. This latter category can be aided efficiently by engineering tools. The support for these patterns in OO design has mostly been automated by the insertion of certain incomplete models into UML diagrams, especially into class diagrams and sequence diagrams.

With the increasing popularity of Domain-Specific Modeling Languages (DSMLs), the demand for the notion of pre-defined building blocks that can be readily inserted at modeling time is observable in industrial development. The key issue of DSMLs lies in the tool support: this is provided by highly customizable metamodeling environments and their model transformation capabilities. Therefore, the adaptation of patterns for the DSML technology should include concepts for the tool support and its seamless integration into metamodeling environments. Here, we discuss only those patterns that can be expressed by modeling artifacts thus can be aided by tool support.

This paper is devoted to the description of our ongoing work with domain-specific applications of patterns. Our notion of DSML patterns is referred to as **Active Model Patterns** (AMPs) introduced in Section 2. Section 3 discusses the classical insertion approach to patterns. Section 4 summarizes related work, and Section 5 concludes the paper.

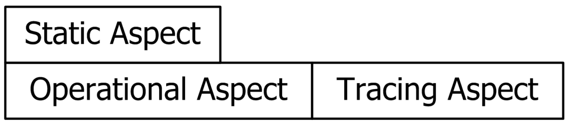


Figure 1: The architecture of the active pattern infrastructure

2 Active Model Patterns

In order to adapt the notion of design patterns to DSML environments, we need to generalize the original concept. In the OO world, there are several types of patterns: analysis patterns [Fow96], design patterns, architectural patterns [BMR+96]. DSMLs are meant to be used in arbitrary domains, thus, in accordance with [LLM09], DSML patterns are referred to as **model patterns**.

Another direction of generalization is the level of sophistication of the provided operations. The application of classical design patterns consists of an incomplete (partial) model, and an intelligent insertion operation that places the chosen pattern into the model. The insertion may involve binding to existing elements, creating new elements, or completing missing attributes from user input. We call this approach **static model patterns**.

Active Model Patterns (Fig. 1) are an extension of static model patterns with universal design-time model manipulations, optionally combined with tracing of the operations. The operations can be viewed as on-demand localized model transformations applied interactively. These manipulations constitute the **operational aspect** of AMPs.

The **tracing aspect** of AMPs provides detailed and transactional log information on the model manipulations. This can provide a basis for undo, redo, remove, and other operations. The detailed discussion of the tracing aspect is beyond the scope of this paper.

Incorporating the universal properties of static model patterns, AMPs have the following advantages. (i) Not only traditional UML diagrams are treated but patterns over models expressed in arbitrary DSMLs. (ii) Not only classical design patterns are supported but model patterns with arbitrary objectives; they could be produced for analysis or other purposes. (iii) Not only the traditional static model patterns are made possible, but any changes that can be described as a set of model manipulations, such as refactorings, layout adjustments and many more. (iv) Not only insertion of a pattern or the application of a model transformation is performed, but tracing information is also maintained automatically. Thus, undo and remove operations over the *entire* transaction of the pattern application are also possible.

Active Model Patterns can also be interpreted as the cross-fertilization of domain-specific design patterns and refactoring. Indeed, the tool infrastructure for AMPs includes the tool support for both the insertion of static patterns and the application of model refactoring.

2.1 The operational aspect of AMPs

Suppose, in case of a DSML for specifying user interfaces (Fig. 2), the designer wants to define operations for layout adjustment. While this cannot be specified as a ”classical” static model pattern, the operational aspect of Active Model Patterns provides a natural solution for this problem.

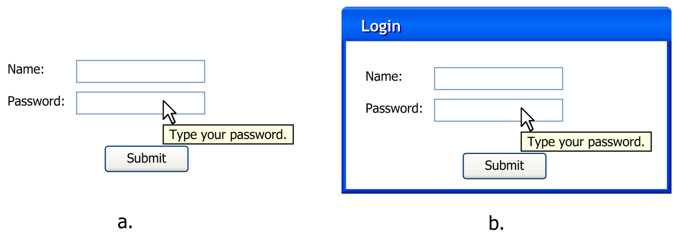


Figure 2: (a) Static model pattern. (b) Inserted static model pattern.

The realization of a universal operational aspect may happen in several ways. The most obvious ones are as follows. (i) Using a modified model transformation environment. (ii) Manipulating the models via the exposed model database APIs. (iii) Developing a proprietary domain-specific language and a related model processor.

In case of applying a model transformation system, the existing transformation environments must be slightly modified for this purpose. The pattern application is an interactive process controlled by the designer. Graph rewriting-based model transformation systems consist of transformation rules that have a left-hand-side graph (LHS) to be matched in the model and a right-hand-side graph (RHS) with which the match is replaced. For active model patterns, matching the LHS must be accomplished with the involvement of the designer: the tool should offer manual binding of certain LHS elements. The advantage of this approach is that the tracing aspect may be easier to store and restore. Transformation systems using triple graph grammars (TGG) or other tracing formalisms can gain the advantage of their mechanisms. The drawback of using a model transformation system is that these tools are quite complex and require a certain amount of insight not easy for the domain experts/modelers to master. However, the transformation engines can be used when one decides to take the DSML approach discussed below.

Programming using the object-oriented APIs exposed by a modeling tool is also a possible implementation of the model manipulation in the operational aspect. These APIs range from general, model traversal functions to generated domain-specific APIs. The abstraction level of operations may be low, and maintaining the tracing information could be very complicated with this approach. The domain experts and modelers cannot be expected to be familiar with an object-oriented programming language.

The third alternative is developing a DSML for this purpose. The main requirements of the language are the following. (i) Usability for domain experts and modelers. (ii) Provide enough information for generating the data structures for the tracing aspects. (iii) It should be possible that the pattern DSML's model processor generates input for either a model transformation engine or the APIs discussed above. We believe that this is an important open problem.

3 The static aspect of AMPs

The universality of the operational aspect makes it possible to describe static model patterns as well. However, a DSML describing model manipulations or a model transformation system does not provide an easy way to define ”classical” static model patterns.

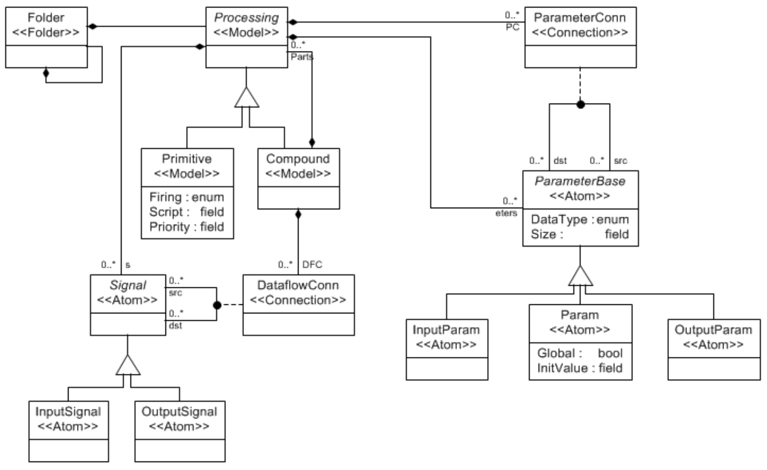


Figure 3: The original metamodel of the Signal Flow paradigm

In the example above, it is hard to specify the insertion of the pattern (Fig. 2) element by element with multiple insertion operations. Although a few model transformation systems allow the use of DSML model elements with their icons, it is hard for the designer to develop static patterns this way.

Therefore it is reasonable to provide a user interface on top of the operational aspect for defining and editing incomplete models of a given DSML. The application of the patterns can be accomplished either directly with an intelligent insertion operation, or the patterns can be translated to one of the platforms that realize the operational aspect (i.e. via model transformations or directly using the APIs).

Our goal is to define incomplete models in a given DSML. We assume that the metamodel of the DSML is available. For the realization of a static pattern definition environment, we have identified the following approaches. (i) The *metamodel extension* method modifies the metamodel preferably in an automatic way such that the modified metamodel provides a DSML environment for pattern definitions. (ii) The *tagged pattern* method turns the elements of the original DSML into patterns by tagging them.

3.1 Creating pattern environments with metamodel extension

When using the metamodel extension method, we process the original metamodel in a tool that generates a DSML for the pattern environment. In our case study, we use the metamodel of our Signal Flow paradigm to illustrate the different methods. Figure 3 shows the original Signal Flow metamodel.

The data processing blocks are organized into a composite pattern: the compound containers can recursively contain primitive blocks or further composites. Inside the processing blocks, signals can be defined as either input or output signals. The signals can be connected with dataflow connections. The processing units can contain parameters that can be connected with parameter connections.

The pattern environment generator takes the Signal Flow metamodel (Figure 3), and creates a

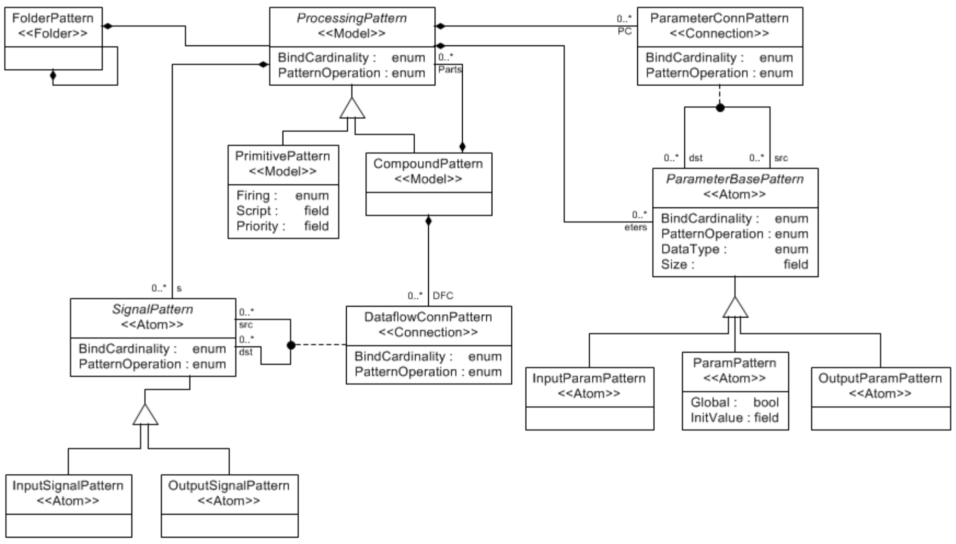


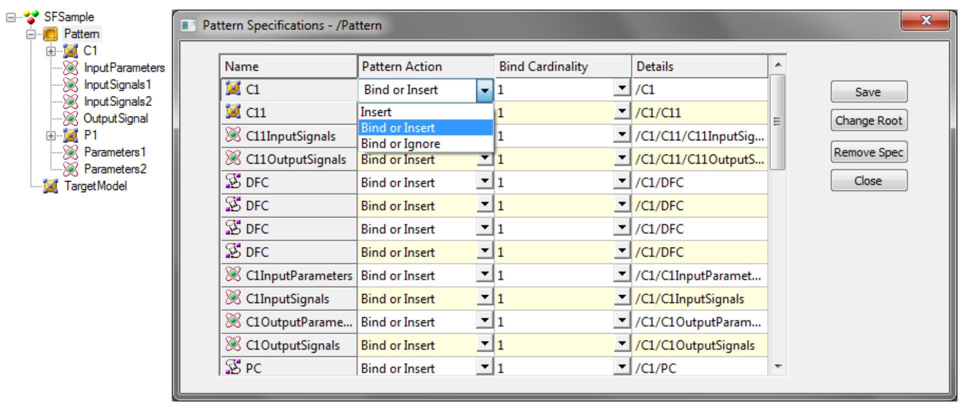
Figure 4: The extensions to original metamodel of the Signal Flow paradigm

new metamodel (Figure 4): the pattern environment.

The metamodel of the pattern environment consists of the original metamodel, a copy of the original metamodel where each element (class, association, etc.) is renamed with the extension *Pattern* and is extended with action attributes that hold insertion-time instructions for the patterns. The attributes are inserted at the roots of the inheritance hierarchy. In our case study, only the *SignalPattern*, *ProcessingPattern*, and *ParameterBasePattern* elements are extended with the action attributes, along with the connections. It is really important to notice that both the original metamodel and the elements of the pattern definition (Figure 4 constitute the final pattern environment).

An action attribute can hold the following values: *insert*, *bind or insert*, and *bind or ignore* that determine what happens with the pattern element at insertion time. The value *insert* means that the element must be inserted into the target model. The term *bind* refers to an insertion time activity in which the user of the tool can manually assign the pattern element to model elements that already exist. In this way, the user can define a context for the pattern. If the user decides to leave these elements unassigned at insertion time, the second action, namely, *insert* or *ignore*, will be performed.

When this method is used, a pattern will be contained in a folder, either because of the structure of the original language or because the metamodel extension tool automatically generated one for the patterns. It often happens that the incomplete model violates the structural or semantic constraints of the DSML in general. In our example above, edit boxes cannot exist without a form. As another example, assume UML class diagrams as the modeling language and a modeler who wants to define an Observer pattern. Although classes are placed in class diagrams as defined by the language, the class diagram itself should not be inserted. Deeper containment hierarchies are very common in DSMLs. In this case the elements that are enforced by the language but not to be inserted are marked as *ignore*.



Name	Pattern Action	Bind Cardinality	Details
C1	Bind or Insert	1	/C1
C11		1	/C1/C11
C11InputSignals		1	/C1/C11/C11InputSig...
C11OutputSignals	Bind or Insert	1	/C1/C11/C11OutputS...
DFC	Bind or Insert	1	/C1/DFC
DFC	Bind or Insert	1	/C1/DFC
DFC	Bind or Insert	1	/C1/DFC
DFC	Bind or Insert	1	/C1/DFC
C1InputParameters	Bind or Insert	1	/C1/C1InputParamet...
C1InputSignals	Bind or Insert	1	/C1/C1InputSignals
C1OutputParame...	Bind or Insert	1	/C1/C1OutputParam...
C1OutputSignals	Bind or Insert	1	/C1/C1OutputSignals
PC	Bind or Insert	1	/C1/PC

Figure 5: Tagging model elements

For the pattern elements marked as *bind*, there are two options. They can either be bound to one element or multiple ones. In order to specify this option, we use the *Bind Cardinality* attribute, which is considered only when the element is marked as *bind*, and can be either *1* or * (many).

The insertion process is implemented as follows. The original model element types must be located in the metamodel, based on the type of the pattern element, new model elements of the original type must be created, and then the attributes must be copied from the pattern element, except for the action attributes. The original model element type can be found by string operations or via a mapping table kept in the pattern environment. The drawback of the first solution is the string manipulation that the environment has to perform when instantiating the metamodel. For instance, our implementation environment, the Generic Modeling Environment (GME) [LBM+01] appends suffixes to the role names, or if they are blank, it generates a role name as the combination of the participant names. The additional table means external information and needs to be maintained separately, which is obviously an additional burden on the implementation.

3.2 Tagged Pattern Elements

In the previous section, the additional pattern information was added to the model as the part of an extended DSML. It required the modification of the metamodel. As opposed to modifying the metamodel, we can store the pattern information separately from the DSML creation mechanisms. We use the metamodel of the original DSML, and the pattern is created as any other regular model in the DSML. What makes a partial model a pattern is the presence of certain attached tags. In our approach, container type elements, such as UML packages and class diagrams, can be the root of a pattern. In GME, this is expressed by the meta-metamodel element *Model*. The root element of the pattern is marked as *PATTERN_ROOT*. This tag turns all the children of the root element into a pattern. In a containment hierarchy, there can be multiple pattern roots, which means that not only the whole hierarchy, but also certain parts can also be used as a pattern. Figure 5 illustrates the tagging process.

This method requires a tool that supports tagging model elements. GME assigns a registry tree to all model elements. The registry is a hierarchical key-value pair structure, where the key is of type string. The action attributes of pattern elements are also stored in the registry. Our implementation supports any container type element as a pattern root, as opposed to our implementation of the extension method, where only folders can have this role. Therefore, we do not need the *ignore* action anymore.

3.3 Attribute mangling

While using our tool, we identified a need to insert certain string attribute values other than literals. For example, we may want to add a unique suffix to the name of the pattern elements, because names must be unique in the target model. In order to address these issues, we introduced certain attribute operations that are performed at insertion time. These operations are represented by character sequences delimited by % used in the value of string-valued attributes of pattern elements. The operations behave similarly to environment variables of operating systems: they are replaced with actual values when the pattern is applied.

The first group of operations consists of counters. Counters can be local or global. Local counters are local to pattern elements: if a pattern element using a local counter is processed for the first time, its value is zero, the next time (in case of multiple bindings) it becomes one, and it is similarly incremented at every subsequent occasion. Global counters are incremented each time they are used. Local counters can be included in the attribute values by inserting the string $\%\$LOCAL\_COUNTER\%$, and global counters are embedded using the variable $\%\$GLOBAL\_COUNTER\%$. If one wants to embed a counter value without incrementing it, the value of the variable with the suffix $\_REF$ must be used. If a pattern element contains counter variables and references, the counter variables are evaluated and incremented first, then all the references reuse the result of the last increment.

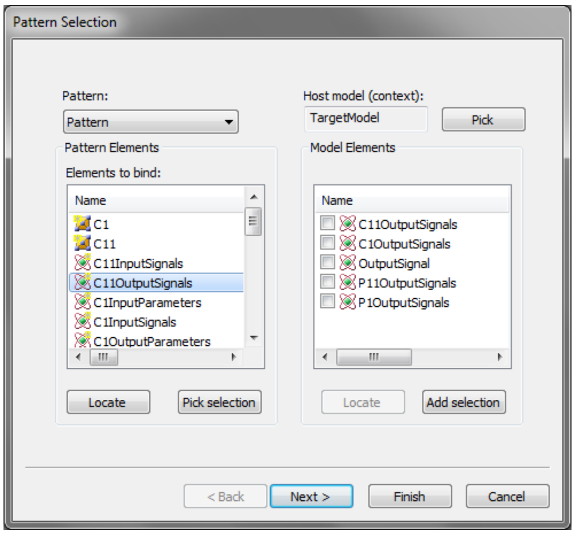
The counters discussed above are initialized at every execution of the pattern applicator tool. If one wants to guarantee a unique name between the executions, the $\%\$UNIQUE(core)\%$ and $\%\$UNIQUE\_REF(core)\%$ are used. The first operation takes the name *core*, an internal counter, and increments the counter until the *core* suffixed with the counter becomes unique in the subtree into which the insertion is performed. The variable is substituted with this value, similarly to the reference version of this operation.

Other operations include embedding a globally unique identifier (GUID), $\%\$USER\_INPUT\%$ asks the user at insertion time.

Another group of operations references the attributes of the model element to which the pattern element is bound. For example, $\%name\%$ is substituted with the name attribute of the bound model element.

3.4 Pattern Application

The insertion includes the following operations. (i) The user must specify a model that matches the type of the pattern root. This is the insertion scope of the pattern. (ii) The user binds the bindable elements or leaves them unassigned. This step is illustrated in Figure 6. On the left of the dialog box, the element marked as *bind* are listed. On the right-hand-side list box the


Pattern Selection

Pattern: Pattern

Host model (context): TargetModel

Pattern Elements — Elements to bind:

Name
C1
C11
C11InputSignals
C11OutputSignals
C1InputParameters
C1InputSignals
C1OutputParameters

Model Elements:

Name
C11OutputSignals
C1OutputSignals
OutputSignal
P11OutputSignals
P1OutputSignals

Figure 6: Inserting patterns

model elements with the matching type are enumerated. Marking the checkboxes pattern elements can be bound to model elements. In GME, one can use typed references that refer to model elements of a specific type. This construct can create a logical connection between model elements that reside in different places in the containment hierarchy. For these elements, the tool allows the user to bind elements outside of the insertion scope, however, if they are not used with reference targets, the insertion is canceled with an error. (iii) The user provides values to the $\%\$USER\_INPUT\%$ variables.

If the pattern is not legal with respect to the metamodel, the insertion is aborted with a paradigm violation error message. In GME, we use the transaction capabilities provided by the tool to roll back the whole insertion process.

4 Related Work

There are several mainstream tools that support UML design patterns, or describe design patterns with a general languages, as opposed to using the metamodel of the DSMLs. Moreover, there are several approaches for pattern formalizations. In this section, we reference the closest related work only.

Previous work [LLM09] has justified the demand for Domain-Specific Model Patterns by contributing several DSMLs. Moreover, it describes relaxation conditions for the metamodels in order to make metamodeling environments support the editing of incomplete models. As opposed to this paper, it deals with static model patterns only. In our approach, relaxations can be made on the metamodel of the pattern environment. The multiplicities can be substituted with the upper bound of the multiplicity set, dangling edges can be defined with ignored end nodes and transitive containment can be solved with ignored containers. Incomplete attributes can be implemented the same way.

[KFGS04] describes a UML-based language, namely, the Role-Based Metamodeling Language (RBML), which is able to specify domain-specific design patterns. This approach treats domain patterns as templates, where the parameters are roles. A tool generates models from this language. Our approach manipulates the metamodel of the DSMLs. Moreover, we have provided a vision for active model patterns.

In our architecture, the paper [KC09] proposes a formal way to specify the pattern embedding for the static aspect. The behavioral formalization is closely coupled with design patterns defined in UML.

The work described in [BGL09] formalizes the embedding, tracing, and synchronization between several pattern aspects that may be defined in different languages. These results constitute an excellent theoretical formalization of the tracing aspects for model patterns defined in the static aspect.

An early work [Kar01] introduced a transformational approach to representing and applying design patterns. The approach described there served as the foundation of the work presented here.

5 Conclusions

We have contributed an infrastructure for Active Model Patterns. AMPs are a generalization of classical design patterns in several ways. (i) They are applicable to arbitrary DSMLs. (ii) They can specify arbitrary design-time model manipulations as well. (iii) Optionally, the infrastructure can store and interpret trace information grouped by transactions.

We have outlined the tool support for the individual aspects, and compared the solutions. We have provided two novel approaches to a static DSML pattern definition environment applicable to an arbitrary metamodeling tool. Whereas existing solutions target the relaxation of the instantiation relationship, our method is based on the metamodel. We have implemented this environment within the tool Generic Modeling Environment (GME).

The metamodel extension method raises several practical issues. Wherever name-based type identification is performed by the modeling tool, the pattern elements will behave differently from their original counterparts. For example, if a decorator software component, which provides custom visualization for model elements in GME, presents a class according to the UML standard, the pattern elements will not show up as such, unless the decorators are modified. Also, the extension method causes the metamodels to become larger, and the whole architecture is extremely sensitive to language evolution.

This is why we decided to implement the tagging method. The drawback of this method is exactly the opposite. Since the patterns use the same types as the models, they can be differentiated based on the tags. Therefore, the model interpreters that process the models have to check the tags in order not to process the patterns.

Future work includes the design and implementation for the DSML approach of the operational aspect of AMPs.

Acknowledgements:

This work was sponsored by DARPA, under its Software Producibility Program. The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA or the US government.

Bibliography

[BGL09] P. Bottoni, E. Guerra, J. Lara. Formal Foundation for Pattern-Based Modelling. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*. Pp. 278–293. Springer-Verlag, Berlin, Heidelberg, 2009.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[Fow96] M. Fowler. *Analysis Patterns: Reusable Object Models (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[Kar01] G. Karsai. Tool Support for Design Patterns. NDIST 4 Workshop, December 2001. http://www.isis.vanderbilt.edu/sites/default/files/Karsai_G_12_0_2001_Tool_Suppo.pdf

[KC09] S.-K. Kim, D. A. Carrington. A formalism to describe design patterns based on role concepts. *Formal Asp. Comput.* 21(5):397–420, 2009.

[KFGS04] D.-K. Kim, R. France, S. Ghosh, E. Song. A UML-Based Metamodeling Language to Specify Design Patterns. In *Proc. Workshop Software Model Eng. (WiSME)*. 2004.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing domain-specific design environments. *IEEE Computer*, pp. 44–51, November 2001.

[LLM09] T. Levendovszky, L. Lengyel, T. Mészáros. Supporting domain-specific model patterns with metamodeling. *Software and Systems Modeling*, Mar. 2009. doi:10.1007/s10270-009-0118-3