Acta Cybernetica **24** (2019) 5–16.

Towards a Classification-Based Systematic Approach to Facilitate the Design of Domain-Specific Visual Languages*

Sándor Bácsi[a] and Gergely Mezei[b]

Abstract

Domain-specific visual languages (DSVLs) are specialized modeling languages that allow the effective management of the behavior and the structure of software programs and systems in a specific domain. Each DSVL has its specific structural and graphical characteristics depending on the problem domain. In the recent decade, a wide range of tools and methodologies have been introduced to support the design of DSVLs for various domains, therefore it can be a challenging task to choose the most appropriate technique for the design process. Our research aims to present a classification-based systematic approach to guide the identification of the most relevant and appropriate methodologies in the given scenario. The approach can be capable enough to provide a clear and precise understanding of the main aspects that can facilitate the design of DSVLs.

Keywords: domain-specific visual languages, modeling, classification

1 Introduction

In software development, there has always been a big demand for improving the productivity and the speed of the development process by increasing abstraction. This is the main reason why model-driven software development [1] has become a promising paradigm among software developers and researchers in the past decades. Software models are mainly used for designing complex structures or systems in order to be able to specify the requirements on a higher abstraction level. Thus, the model can give a better overview of the system and help to understand the concepts of the targeted domain.

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]Department of Automation and Applied Informatics, Budapest University of Technology and Economics E-mail: bacsi.sandor93@gmail.com

[b]Department of Automation and Applied Informatics, Budapest University of Technology and Economics E-mail: gmezei@aut.bme.hu

DOI: 10.14232/actacyb.24.1.2019.2

Domain-specific languages (DSLs) [10] are specialized modeling languages that can efficiently raise the level of abstraction by using the concepts and the characteristics of the specific problem domain. DSLs are in a contrast to general-purpose languages like C, Python or Haskell that are designed to let developers write any sort of program with any sort of logic broadly applicable across domains. DSLs allow the effective management of the behavior and the structure of software programs and systems. Domain-specific visual languages (DSVLs) [15], compared to textual DSLs, can further improve the expressiveness and the usability of the given model. As a well-designed DSVL raises the level of abstraction, it also helps in hiding irrelevant, technical details and in emphasizing the domain-related parts in the models. In order to achieve this, it is essential to find the best visualization in design-time in order to satisfy the needs of the targeted domain.

There are several advantages of using a DSVL. The richness of the visual representation can simplify the modeling process and increase flexibility, thus DSVLs can be intuitively usable. As most of the people tend to associate a visualization for their problems, visual models can facilitate to understand the concepts and the main relations in the targeted domain. It can be easier to explain the main characteristics of a domain problem by using visual notations.

Compared to textual languages, DSVLs may have their drawbacks. DSVLs can be restrictive, since they may limit the freedom of creating complex language constructs. The visual entities representing a complex code can be hard or impossible to grasp in one glance. It can be challenging to find the effective visual way of expressing some advanced concepts, such as type systems, that can be found in most of the general-purpose textual programming languages. If DSVL is badly designed and it is used in a particular situation, the advantages may easily turn into disadvantages, thus it is essential to avoid counter-productive decisions by choosing the most appropriate representational concepts in design-time. A guideline can help in providing a clear and precise understanding of the main aspects to design the most suitable DSVL.

The high level of customization possibilities has its price: unlike in UML, each problem domain requires a custom, different visual representation to meet the requirements of the targeted domain. The exactness of the choice depends on how expressively the chosen concepts describe the DSVL and on the specific needs of the targeted domain.

In this paper, we present the main results of our classification methodology for visual domain-specific languages. We analyzed a wide range of existing DSVL methodologies and also created several illustrating examples for different domains to exemplify the most relevant graphical and structural characteristics. We used two metamodeling frameworks (Eclipse Modelling Framework [6] and Visual Modeling and Transformation System [18]) and a visual programming editor builder (Google Blockly [2], [7], [12]) to examine the most applicable methodologies.

The paper is organized as follows: Section 2 presents the background and the related work. Section 3 introduces our approach in order to give an understanding of the main concepts. Section 4 presents some of the illustrating examples which we elaborated, while concluding remarks are outlined in Section 5.

2 Related work

Various kinds of classifications have been created in the past to support the design process of DSVLs. However, most of these approaches are quite old, there is no relevant publication in the field for more than ten years. Due to the increasing use of DSVLs, a wide range of new tools and methodologies have been introduced recently based on completely new ideas. Our research aims to analyze and compare the most relevant methodologies on a larger scope which can support the design of new domain-specific visual languages with the new technologies. Different classifications of DSVLs have been presented in the literature. Basically, these classifications serve a completely different purpose than the one introduced in this research.

The principles in [9] are aimed at creating a hierarchy for visual languages which is based on the constraint multiset grammar formalism. The approach also takes into consideration the expressiveness and the cost of parsing for different classes. This approach is mainly based on formalism, rather than on the pragmatic use of DSVLs.

Myers [11] discusses programming systems and it is divided into categories using the orthogonal criteria of being visual programming or not, example-based programming or not, and interpretive or compiled. Similarly, in another paper [5] the authors presented a classification system, in which visual languages are categorized based on the visual programming paradigm they express and different visual representations.

There is another classification approach [3] which presents a suite of metamodels as a basis for a classification of visual languages. This approach introduces general metamodel patterns which can serve as a basis for different aspects that can facilitate the design of DSVLs. However, the approach does not take into consideration the possible non-metamodeling concepts and the pragmatic use of DSVLs.

There is a wide range of existing professional general-purpose modeling languages in the field of software engineering. For example, UML [17] and SysML [16] are intended to provide a standard way to visualize the design of different systems. Here, it is important to emphasize that our research focuses on creating new domain-specific visual languages considering the requirements of a certain domain, thus universal, standardized visual languages are not taken into account.

Our classification-based approach is not intended to be superior to other classification-based methodologies, it serves supplementary purposes. Our approach is mainly based on the nature of the graphical objects that compose the visual language, the connection types among the graphical objects, the composition rules and the visual representations. We also consider non-metamodeling approaches and compare them to metamodeling methodologies. In this way, we can provide a clear and precise understanding of the main aspects that can facilitate the design of DSVLs. We introduce a step-by-step guide on how to use our systematic approach in different design scenarios.

3 Classification-Based Systematic Approach

In this section, we present the steps of our classification-based systematic approach. Each subsection represents a step of our methodology. It is important to emphasize that not all of the steps can be used directly in all possible design scenarios. Some of the steps (Section 3.1, Section 3.2 and 3.3) are meant to decide between a couple of mutually exclusive choices, while others (Section 3.4 and Section 3.5) are used only as a supporting step helping to fine-tune previous decisions.

3.1 Step 1: Flow type

Based on the flow type, domain-specific visual languages can be grouped into three subclasses: data flow languages, control flow languages and languages with no flow.

Data flow languages visualize the steps of data processing. Data flow concepts are based on the idea of disconnecting computational actors into stages that can execute concurrently. Data flow DSVLs visualize the processes that are undertaken, the data produced and consumed by each process, and the accumulative graphical objects needed to hold the data. It is possible to visualize what the system will accomplish by the flow of data.

Control flow visual languages visualize the logic of computation by describing its control flow. Control flow DSVLs graphically express the order in which instructions or statements are executed or evaluated. The graphical objects mainly represent the control structures and conditional expressions of the language, thus it is possible to visualize how the system will operate by the flow of control.

There are DSVLs which are neither data flow nor control flow because they target a static domain problem. These languages are used mainly to represent the structure of a system or a program, therefore no flow has to be described. A widely used example of no-flow graphical modeling languages is the UML class diagram, in which the structure of the system is described by the classes and the connections among them.

3.2 Step 2: Relation type

Based on the relation type, domain-specific visual languages can be grouped into two subclasses: containment-based and connection-based subclasses.

In the case of containment-based languages, entities are limited to embed in each other to express sentences of the targeted domain, no other types of connections (e.g. association, or inheritance) are supported. As the customization of embedding, graphical entities may be attached to other entities (e.g. representing methods and their parameters) and chained together (as in a call stack). To support this behavior, a predefined set of containment rules or constraints have to be specified to restrict the way of embedding, attaching and chaining. Blockly and Scratch [13] are widely used examples of containment-based languages, both support building blocks that can be connected like puzzle pieces in order to create easy-to-understand visual sentences.

Connection-based languages consist of two different kinds of building elements: entities and connections, i.e. nodes and edges. While data is usually expressed by entities, the flow of the model and the relations among entities are defined by connections. Connections may also have properties to ensure the customization of the relations among entities. Moreover, connections may also interpret containment as the container and the contained elements can be connected by a specialized containment-typed edge. This means that this category is more general, however its complexity is not needed in many practical cases.

3.3 Step 3: Methods of the abstract syntax definition

There are two key methods for the abstract syntax definition of a DSVL: metamodeling-, and non-metamodeling approaches.

Metamodeling methodologies provide methods for defining DSVLs based on the abstract notion of visual entities and of relations among them. These frameworks are capable of specifying the abstract syntax of a DSVL and expressing the additional semantics of existing information. The metamodel can expressively define the structure, semantics, and constraints for a family of graphical models. On the other hand, when a metamodel is instantiated, its elements become types, which can be instantiated in the instance models. Hence, complex structures and relations can be described in a flexible way by the usage of metamodeling concepts.

While metamodeling methodologies are based on various kinds of instantiation techniques, non-metamodeling approaches provide a somewhat simpler, template-based structure for creating visual entities. The main characteristic of non-metamodeling approaches is that they have a limited set of features which can be used on the different abstraction levels, thus complex structures cannot be visualized flexibly and expressively. One of the newest non-metamodeling approaches is Blockly. It supports a large set of features for different domains. In Blockly, the graphical objects are called blocks which can be customized as the basic building elements of the language. However, due to the template-based and weakly typed structure, complex type constraints cannot be applied.

3.4 Step 4: The way of the problem description

This is a fine-tuning step, since this step rather depends on the specific nature of the problem domain and also on the needs and preferences of the users. Based on the way of the problem description, domain-specific visual languages can be grouped into two subclasses: imperative and declarative languages.

Declarative visual languages describe the logic of computation. For example, SparqlBlocks [4] is a declarative DSVL developed in Blockly. Declarative languages visualize sets of declarations or declarative statements. Each of these visual declarations has a meaning depending on the targeted domain and may be understood independently. A declarative style of visualization helps to understand the problems of the targeted domain and the approach that the system takes towards the

solution of the problem, but is less expressive on the matter of mechanics which describe the flow of the system.

Imperative visual languages consist of visual statements that change the state of a program or a system. For example, Scratch is an imperative visual programming language. The visualized statements express the way of execution of which results in a decision being made as to which of two or more visualized paths to follow. In imperative languages, the visual sentences can be created by sequences of commands, each of which performs some action. These actions may or may not have a dedicated meaning in the targeted problem domain.

3.5 Step 5: Visual representation

This is also a fine-tuning step, since it is related the concrete syntax of the language and it strongly depends on the needs and preferences of the users. DSVLs have a visual concrete syntax used for the representation of graphical elements and connections. Based on the visual representation, there are two key design aspects: iconic and diagrammatic visual representation.

In the case of iconic languages, entities are visualized by icons. For example, Lego Wedo 2.0 Software [8] provides an iconic visual language for educational purposes. The iconic language is a structured set of related icons. An icon can be attached to or composed of other icons, thus expressing a more complex visual concept.

Diagrammatic languages are mainly composed of elements with a pre-defined symbolic representation of information. The building blocks of diagrammatic languages such as geometric shapes are often connected by lines, arrows, or other visual links. Chart-like, schematic-like and graph-based visual languages are the most widely used examples.

The most important difference between iconic and diagrammatic languages is that icons are pre-defined and they have limited flexibility, while graphical building blocks of diagrammatic languages can be calculated and customized freely.

4 Illustrating examples

We can investigate some advantages and disadvantages of different approaches by solving various domain problems. In this section, we introduce two illustrating examples to present different design scenarios built upon the classification-based approach presented. Through the examples we only investigate the mutually exclusive steps from Step 1 to Step 3 because they specify the structural characteristics of the DSVL.

4.1 Logic gates

The first illustrating example demonstrates the domain of logic gates. In this domain, logic gates can perform logical operations on one or more binary inputs

and produce a single binary output. For the sake of simplicity, we can use AND, OR and NOT gates. There are visual entities which can only transmit a binary signal, while other visual entities can only receive the signals, therefore it is possible to create entities with a single input or output.

Step 1: We have to make our first design decision based upon the first step of the systematic approach. It is certain that we have to design a control flow language, because logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction and transmission of all of Boolean logic, and therefore, all of the mathematics and algorithms that can be described.

Step 2: The next question is whether the DSVL fits the connection-based or the containment-based approach. The answer is not that simple as it seems at first glance. If we choose the connection-based option, logic gates can be represented as nodes that can be connected with edges, creating thus a connection-based language. Node-like model elements can be connected to each other, where we use ports instead to define the interface of a node. For example, a logic gate OR can have two input ports for the operands and a single output port, for its result. If we choose the containment-based option, it is very difficult to express the connection among logic gates, since no edges can be used. On the other hand, it can be hard to customize the interface of logic gates. In conclusion, the DSVL fits better the connection-based approach.

Step 3: In this step, we have to make our decision regarding the abstract-syntax definition. It is clear that complex structures and relations can be described in a flexible way by the usage of metamodeling concepts. Taken the previous structural decisions into account, it can be more effective to use a metamodeling methodology. We used VMTS to define the abstract syntax of the language and to set up custom visualization for the graphical editing. To support this behavior, VMTS allows to define the so-called meta ports on nodes. Figure 1 shows a half adder model as an example in VMTS. Here, it is important to emphasize that due to the connection-based nature the output result of the given node can be used for more inputs, therefore particular nodes with the same logic do not have to be duplicated.

Alternative solution: For the sake of completeness, we tried a different design scenario in our classification-based approach to prove the importance of the appropriate design decisions. Let us assume the following design scenario: Unlike the previous design scenario, after Step 1, we can make a different decision. In Step 2 we choose the containment-based approach even if we are aware of the fact that this is not the better option. In Step 3 we decide to use a non-metamodeling approach. We used Blockly to create the containment-based variant of this example. While it is easy to define the blocks themselves, it is very difficult to express the connection among logic gates, since no edges can be used. Blocks have to be duplicated and there can only be one output on a block - the left output. Figure 2 shows the same model as in the connection-based example, but visualized in Blockly. In this example, the AND logic has to be used twice from input A and B to be able to implement the half adder logic. Even the input A and B visual entities have to be duplicated.

As the illustrating example shows, dealing with multiple connections in a control

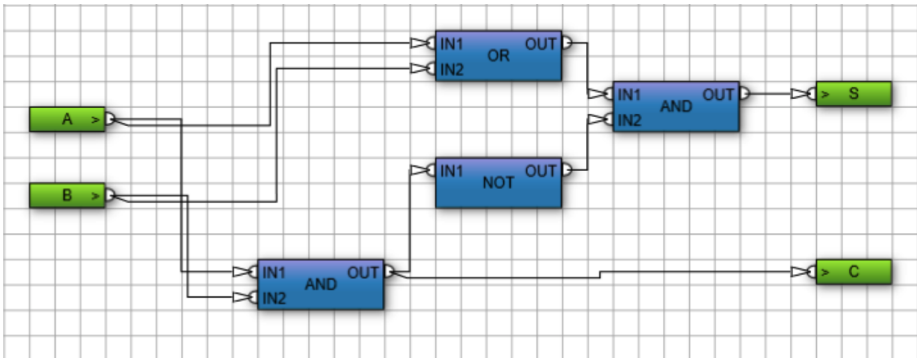

Figure 1: Half adder example in VMTS

flow domain may have its drawbacks in the containment-based approach. It is more expressive to use the principles of the connection-based approach to graphically express the order in which instructions or statements are executed or evaluated.

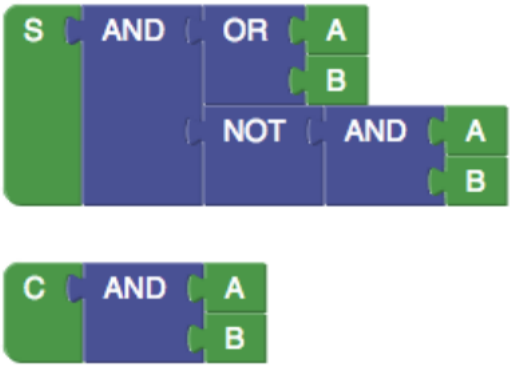

Figure 2: Half adder example in Blockly

4.2 Departments of a company

In this illustrating example, we present a simple DSVL for modeling the departments of a company. Let us assume the following specification: *A company has different departments. Employees work in departments. Employees may have a principal and every department has exactly one director.*

Step 1: At first, we have to make our first design decisions to identify the flow type of the domain. It is certain that we have to design a no-flow language, because no flow has to be described, only the static relations among entities are to be modeled.

Step 2: The next question is whether the DSVL will be connection-based or containment-based. If we choose the connection-based option we can definitely visualize the employee - principal relationship with some kind of visual links. On the other hand, it can be difficult to visualize the employee - department relation, because after a certain amount of employees the visual entities can be hard or impossible to grasp in one glance. In conclusion, besides the connection-based approach it would be advantageous to use additional containment-based nature for the employee - department relation.

Step 3: It can be easier to express the aforementioned structural characteristics by using a metamodeling methodology. We used EMF to create the connection-based variant of the DSVL. EMF provides effective features to express the basic relations among entities in order to define the abstract syntax of the DSVL. Based on EMF, Sirius [14] provides useful features for the customization of concrete syntax. Figure 3 shows a visualized model as an example. In this simple demonstration,

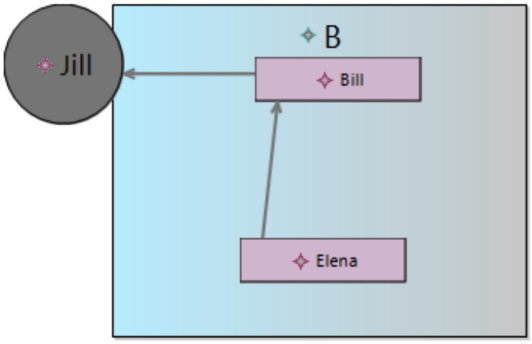

Figure 3: Departments example in Sirius

we used a rectangular box notation for the departments. Arrow notations are used to express the employee-principal relationship and a circle notation is used to visualize the head of the given department. Beside the connection-based patterns, this example has containment-based nature since employee notations can be embedded in departments. Here, it is worth to emphasize that due to the connection-based structure no entity has to be duplicated visually, because they can be connected with the arrow notations to express the employee-principal relationship.

Alternative solution: As the second solution, we elaborated a different design scenario. After Step 1, we can make a different decision. In Step 2 we choose the pure containment-based approach even if we know that it will be hard to express every relation by using just only the principles of the containment-based approach. In Step 3 we decide to use a non-metamodeling approach, Blockly to create the pure containment-based variant of this illustrating example. We used a container block to express the department relationship. The head of the department can be

connected to the *department block*. *Person-principal blocks* can be embedded into the department block to express which employees work in the given department. *Person-principal* blocks can express the hierarchical relationship among employees and principals, however it is more inconvenient and less expressive than in the connection-based approach because blocks have to be duplicated. Figure 4 shows the same model as in the previous example visualized by the principles of the containment-based approach.

In general, for a no-flow domain it is not recommended to use exclusively the concepts of the containment-based approach. On the other hand, in some cases it can be advantageous to let embedding of visual entities even for connection-based languages.

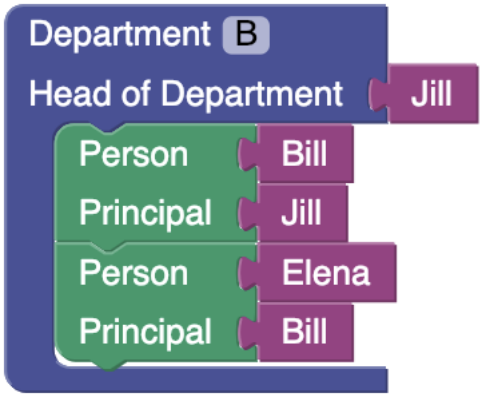

Figure 4: Departments example in Blockly

5 Conclusions

In this paper, we presented several aspects of the classification-based systematic approach for domain-specific visual languages. We believe that the approach can be used as a guide while designing DSVLs. With the help of these guidelines it is now easier to analyze the characteristics of the language and to associate it to an appropriate solution.

We also analyzed the features of Eclipse Modeling Framework, VMTS and Blockly based on different illustrating examples that we created for our classification methodology. We realized that due to the limitations of Blockly, many complex problems cannot be described expressively because aggregations, references and composition rules are missing from its developer framework. Despite the limitations of Blockly, it provides a flexible and easy way to learn to design DSVLs based on containment-based aspects. Unlike Blockly, both EMF and VMTS provide a large feature set for the abstract syntax definition, but they are not as effective and intuitive as Blockly in the definition of containment-based languages.

Further investigations are necessary to validate the kinds of conclusions that can be drawn from this paper. In the future, we aim to create a framework to support the design of visual-domain specific languages based on a questionnaire built upon the methodology presented. It would be beneficial to capture a description of a DSVL from an end-user perspective and give recommendation based on the specification and the specific needs of the targeted domain. The framework should also support an intuitively usable way of designing DSVLs even for complex language constructs and it could assist to align the design of DSVLs to best practices and also benchmark and analyze different design processes. Further studies should investigate how to consider the extensions of existing languages (e.g UML profiles) in the context of our methodology. Therefore, we are also working on new illustrative examples and analyzing other existing approaches to create a more detailed classification-based systematic approach.

References

[1] Beydeda, Sami, Book, Matthias, Gruhn, Volker, et al. *Model-driven software development*, volume 15. Springer, 2005.

[2] Blockly website. https://developers.google.com/blockly/. Accessed: 2018-08-22.

[3] Bottoni, P. and Grau, A. A suite of metamodels as a basis for a classification of visual languages. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 83–90, Sept 2004. DOI: 10.1109/VLHCC.2004.5.

[4] Bottoni, Paolo and Ceriani, Miguel. Sparql playground: A block programming tool to experiment with sparql. In *VOILA@ISWC*, 2015.

[5] Burnett, Margaret M. and Baker, Marla J. A classification system for visual programming languages. *J. Vis. Lang. Comput.*, 5:287–300, 1994.

[6] Emf website. http://www.eclipse.org/modeling/emf/. Accessed: 2018-08-25.

[7] Fraser, Neil. Ten things we've learned from blockly. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*, pages 49–50. IEEE, 2015.

[8] Lego wedo 2.0. https://education.lego.com/en-us/downloads/wedo-2/software. Accessed: 2018-08-21.

[9] Marriott, Kim and Meyer, Bernd. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, pages 375 – 402, 1997.

[10] Mernik, Marjan, Heering, Jan, and Sloane, Anthony M. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. DOI: 10.1145/1118890.1118892.

[11] Myers, Brad A. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123, March 1990. DOI: 10.1016/S1045-926X(05)80036-9.

[12] Pasternak, Erik, Fenichel, Rachel, and Marshall, Andrew N. Tips for creating a block language with blockly. In *Blocks and Beyond Workshop (B&B), 2017 IEEE*, pages 21–24. IEEE, 2017.

[13] Scratch. https://scratch.mit.edu/. Accessed: 2018-08-25.

[14] Sirius website. https://www.eclipse.org/sirius/. Accessed: 2018-08-20.

[15] Sprinkle, Jonathan and Karsai, Gabor. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004.

[16] Sysml. https://sysml.org/. Accessed: 2018-08-29.

[17] Uml. http://www.uml.org/. Accessed: 2018-08-29.

[18] Vmts website. www.aut.bme.hu/Pages/Research/VMTS/Introduction. Accessed: 2018-08-21.