Use of MS DSL Tools in the development process of a Domain-Specific Language

André Rosa¹, Vasco Amaral¹* and Bruno Barroca¹*

¹ Research Center for Informatics and Information Technologies, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Caparica, Portugal
andreifr@gmail.com, vasco.amaral@di.fct.unl.pt, mailbrunob@gmail.com

Abstract. Given the popularity of Domain-Specific Modeling (DSM), metamodeling language workbenches have emerged to help the language designers. In this paper we analyze a specific tool, MS DSL Tools, and its adequacy to the development process of a diagrammatic Domain-Specific Language for the domain of Augmented Reality. We report some of the main challenges and suggest opportunities for improvements. Finally, we collect the more important statements that are valid and pertinent for the evolution of metamodeling language workbenches.

Keywords: Domain-Specific Language, Metamodeling Language Workbench, Development Methodology, Model Driven Engineering, Language Interaction Metamodel.

1 Introduction

Given the popularity of Domain-Specific Modeling (DSM), metamodeling language workbenches (MLWs) have emerged to help the language designers. This paper refers to the development process of a Domain-Specific Language (DSL) using a MLW. MLWs are used to simplify the development process of DSLs, but are still not mature enough. Software language engineers use MLWs to design their DSLs and rapidly prototype their respective graphic editors to be used by the domain experts.

On this paper, we describe the development process of a diagrammatic DSL using the MLW Microsoft DSL Tools [1] as the case study. DSL Tools is, together with MetaEdit+ from MetaCase [2] and EMF/GMF, Eclipse Modeling Framework [3] / Graphical Modeling Framework [4], one of the main MLW used in the industry. And we find it to be a good representative of the current state of the art in MLWs.

This DSL targets the domain of Augmented Reality, described in [5] and [6]. It is a medium complexity language that had a demanding and complex implementation over the MLW. For these reasons, its development process is well suited to evaluate the abilities of the used MLW.

* The results are partially supported by FCT- MCTES project PTDC/EIA/65798/2006.
The objective of this work is to analyze a specific MLW and evaluate its adequacy to the development process of a DSL. We highlight some of the main challenges and flaws, and suggest opportunities for improvements. It is not the focus of this work to detail all the features of the MLW, or to describe the DSL that was created using it. Also there are already some papers where the features of MLWs (e.g. exchange formats, model transformation support, specification of metamodel syntax and semantics), including MS DSL Tools, are analyzed [7]. However, none of them mention the usability and interaction problems we want to highlight. In spite of this work focusing only on one DSL and a specific MLW, we find it relevant and pertinent for the evolution of MLWs. The rest of this paper is organized as follows: on section 2, the background of this work is presented. Section 3 describes the analysis of DSL development using the MLW from the point of view of the software language designer. The domain experts usability report is shown on section 4. Finally on section 5, conclusions and future work are presented.

2 Background

2.1 Domain-Specific Language

A Domain-Specific Language (DSL) is a language that sacrifices generality for proximity to the problem domain. It is focused only on expressing problems from a specific domain, by using concepts from that domain to express problems at a higher level of abstraction. The conceptual and semantic distance between the problem space and the language used is reduced. Because the concepts presented on the DSL are from the target domain and not generic computer terms from the solution domain [8].

The main feature of any DSL is its expressiveness on the target domain. A DSL explores the regularities and standards of the domain. Using a DSL provides simpler, easier and more reliable programming. It also reduces the amount and complexity of development artifacts produced which translates into a better productivity and lower maintenance costs. They allow validation and optimization at the domain level, conservation and reuse of domain knowledge [8]. In some cases, DSL are considered end-user languages because they are supposed to be used by non programmers.

A DSL is normally declarative and provides a smaller set of notations and abstractions when compared to a general purpose language (GPL). DSLs are also known as micro language, little language, application-oriented language or special-purpose language [8]. These definitions, micro and little, refer to the tiny scope of the language and not to the number of concepts present on the DSL which is usually large. Some well known examples of DSLs are SQL, HTML, BNF, LATEX, GraphViz and the shell scripts of the Unix systems.

2.2 DSL Development Methodology

The DSL development methodology follows three main phases: domain analysis, modeling and implementation, as shown in Fig.1. On the domain analysis the problem domain is identified. Relevant domain knowledge is collected and transformed into
concepts and operations that will be used on the design of the DSL. The main sources of domain knowledge, as mentioned in [9], are technical literature, existing implementations, customer surveys, expert advice, and current and future requirements. The outcome of the domain analysis is the domain model.

On the modeling phase the language is designed. The design process of a DSL includes the definition of the abstract concepts (language’s abstract syntax) used in the target domain. Then the representation of each abstract concept is mapped using a particular editor (language’s concrete syntax). The concrete syntax represents the way of how the abstract concepts are to be symbolically presented to the domain experts. Generally, the concrete syntax is categorized into textual or visual/diagrammatic according to the presentation paradigms present on today’s modeling frameworks. Finally, each abstract concept (or patterns) is mapped with its meaning (language’s semantics) on a computational framework.

On the implementation phase it is built a custom editor for the DSL users to work with, and an automatic artifact generator. A software framework can also provide the base for a DSL implementation.

A fourth phase can be considered for when the DSL is deployed for the users to start to use it. The evaluation of the DSL usability may trigger again a new DSL development cycle. The automatic editor generation of the MLW usually become useless when most of the hand-code customizations break the automatic prototyping assumptions admitted by the MLW generation engine.

The creation process of a DSL is ruled by the same base principles that those of a GPL: helps the user, to have a simple syntax and semantic, to achieve a good level of abstraction and expressiveness capacities, and to be orthogonal. In the case of visual DSLs, special care should be taken with the usability of the editor.

### 2.3 Implementing DSLs with Model Driven Engineering

In Model Driven Engineering (MDE), models are the main development and maintenance artifacts of an application [10]. A model is used to construct abstractions that define the properties and features of a system [11].

Models are first class entities allowing a better comprehension of the domain problem concepts, and ready to run code is automatically generated from them. The usage of models as development entities, bring advantages like: a cleaner architecture presentation, conceptual simplicity, efficient implementations, scalability and
flexibility [11]. In MDE, a system is represented by a model that is in conformity with a metamodel. To achieve the full potential of MDE it is necessary that the modeling concepts refer to domain concepts instead of being mapped into generic technological concepts [12].

2.4 MS DSL Tools

Using a MLW simplifies the development process of a DSL. It supplies all the basic necessary functionalities that are needed to build a DSL solution. It enables and guide the language designer to build the language metamodel, and the elements representation, to setup the rules and validations to be run on the models, to create the graphic editor to be used by the DSL users, and to specify the code generation process.

As mentioned there are other MLWs like MetaEdit+ and EMF/GMF. This one, Microsoft DSL Tools [1], was selected because of its implementation language. Nevertheless all of these MLWs are comparable and there is no difference regarding the usability aspects. The decision factor to use DSL Tools (using the latest version available Visual Studio 2008 SDK 1.1), was that it shared the same implementation language, C#, as the target framework. Which allows an easier code transformation from the DSL model to the framework code.

One of the key features of this tool is a set of Application Programming Interfaces (APIs) called the In-Memory Store, that is automatically specified during the DSL creation process. It provides a set of facilities to support the behavior of a DSL including creation, manipulation and deletion of model elements, links and access to the domain model. The base concepts provided by this tool to build the metamodels are domain classes and relationships. For the presentation connectors and a vast range of visual elements are provided, like shapes, compartments and swim lanes. The models are saved using Extensible Markup Language (XML).

Constraints

DSL Tools uses C# code to define constraints. Constraints are used to extend the correctness checking of the models beyond the abstract and concrete syntax of their modeling language. They allow the specification of complex interrelationships between the model data. Soft constraints are wrapped into a validation method that defines its structure and error/warning messages.

Generation methods for artifact generation

DSL Tools provides three techniques that can be used to generate artifacts [1]: Extensible StyleSheet Language Transformations (XSLT); using a domain-specific API; and using a template based approach. Any of these techniques can convert the XML model built on the editor into implementation artifacts such as C# classes or XML files. The use of XSLT usually results in extremely quick model-to-code transformations. But has the downside of the XSLT being non trivial to create and to understand as there is a lot of string manipulation occurring. The second technique uses the API provided by the In-Memory Store and standard .NET code. Still the code to be generated is hard-coded with individual strings that reduce its legibility. Finally,
the third technique is based on text templates. It uses a specific derived template base class that provides access to the language model In-Memory Store. There is a clear separation of concepts between the elements that are rendered directly on the output file, and the elements that are used to provide structure and dynamic behavior of the generation process. This is achieved using special characters for delimiting the text commands. This can be a gradual process dealing with a specific type of model element at a time. The readability of a text template is still not optimal but from the three techniques, this is the closest to the desired output.

**DSL Tools generated editor**

The main common elements in a DSL Tools generated editor, as seen in Fig.2, are the toolbox, design surface, model explorer, solution explorer, messages window and properties window. The toolbox is where the language elements are placed and divided into categories. The design surface is where the language elements are placed (standard zoom-in and zoom-out features are provided). The solution explorer presents a tree like representation of the files that belongs to the solution. The model explorer displays a tree like presentation of the model elements. The properties window displays the parameters of the current selected model element. Finally, in the messages window, error and warning messages are presented. It is also on this window that the language validation messages are displayed.

As the editor provides all these options from the start, some users may have difficulty identifying where to start and what to do. The generated editor should be customizable to the needs of each domain expert profile, offering different abstract edition models. For less experienced users, the use of a wizard to help and guide would simplify and decrease their learning-time of the editor.

The default common interaction method of the generated editor is based on drag-and-drop. The user selects from the toolbox the element to add to the model, and places it on the correct place on the design surface. It is also possible to write code and define more advanced forms and context menus for each shape.

One important note about the DSL Tools is that it provides a visual editor with a base set of features and functionalities. It is possible to go beyond those base functionalities by constructing new functionalities using code. For even more advanced functionalities it is required an understanding of the Visual Studio internal architecture that hosts the DSL Tools. This is a reason why this solution does not scale so well. For more demanding customers it is necessary to create non trivial code that will become increasingly harder to understand. Therefore, the prototyping capacities of this MLW starts to fade away. As it is essential to identify the points where code will need to be introduced and the specification of the code to place there.

### 2.5 Augmented Reality Language

In our case study we developed a DSL called Augmented Reality Language (ARL), which is dedicated to the particular domain of specifications of Augmented Reality Software applications. The ARL is used to define the application logic’s organization using a workflow-like presentation schema, while providing means to configure it. It is also used to define the virtual reality world objects and their behavior. A full
description of this language can be found in [5]. Implementation details and user evaluation results were presented in [6].

It focuses on improving the productivity during the software development process of Augmented Reality applications. This DSL was built using a combination of meta modeling theory, visual languages and DSLs. A metamodeling language workbench (MS DSL Tools) was used to define the language’s concrete and abstract syntax. Its semantics interpretation was achieved by generating code from our models to a target framework. The evaluation process [6] used a series of tests, and a case study where domain experts validated the DSL's efficiency, effectiveness and expressiveness. Fig. 2 presents the ARL DSL Tools editor.

![Fig. 2. ARL DSL Tools editor.](image)

The Augmented Reality Language DSL is a complex language and its development process followed all the phases of the creation process of a DSL. This includes the creation of a complete editor and code generation to a real C# framework (YVision) from YDreams, and an analysis of the users feedback about the use of the DSL. Furthermore, this is not a purely academic exercise, but a real and concrete medium.

---

1 This DSL is the outcome of a collaboration project between the Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (DI/FCT/UNL) with the Portuguese software house YDreams [13].
level complexity DSL, built to be applied on the software development industry context. This is why it makes a good example for the evaluation of the MLW which was used to implement it.

3 Analysis of DSL Development

3.1 Domain Analysis

On the domain analysis phase, the main language requirements and domain rules were enumerated, and most of these rules became constraints later on. In spite of this being one of the most important phase of the DSL development, the MLW provided no special support for this phase. There is no direct connection between the language requirements specification and the MLW implementation.

3.2 Language Design

The creation process of languages is still in consolidation, it has not reached the point of full maturity. Therefore, the language’s design phase can be classified as an “artistic process” that starts when the domain analysis ends, and its outcome is obtaining a language specification. In spite of the existence of several different and configurable parts, it was decided that the DSL (as a whole) would have a consistent and unifying organization and presentation. The usage of nodes and connections was adopted as the base for the language design. The used MLW, like the others (MetaEdit+ and EMF/GMF), provide no particular support for this phase of the DSL development. Some kind of customizable templates for language design (reusable patterns) could have been useful in order to guide and help the software language engineer.

3.3 Language Implementation

The usual phases of a DSL implementation in a MLW are the definition of the language abstract syntax and concrete syntax. Normally, it is also necessary to specify additional constraints, and details about the graphical editor that will be provided to the DSL users.

Abstract Syntax

The language abstract syntax is defined using a metamodel. Each MLW has its own particularities on the definition of the metamodel. Therefore, as it was the case, it is sometimes necessary to slightly adapt the original DSL metamodel so it can be implemented using a particular MLW. The implemented metamodel is very complex. It has 18 main elements and several secondary elements. As such, it turned out to be difficult to visualize specific elements and their relationships. This is the result of an organization issue of the MLW. Proving that the MLW does not scale so well with a large language. A mechanism to compose the language metamodel, using packages
(like EMF/GMF uses), or another structure as a way to organize the metamodel should be provided. Also a system of user-defined views (7 in our language) would help the language engineer to cope with the visual complexity of the metamodel specification. All this would allow an easier abstract language validation process with the domain experts.

**Concrete Syntax**

The visual representation of the DSL concepts follows the logic of nodes and connections, where the nodes are placed inside the boundaries of other nodes.

The concrete syntax is defined by mapping each language concept to its representation. This is a simple and quick process to execute, but has the side effect of being too tied to the metamodel definition. This is especially true in the case of container elements that are represented as aggregations in the metamodel. Furthermore, the container concept is one that is not part of the standard meta-metamodel definition of DSL Tools, so it is a responsibility of the language engineer to construct it. In spite of providing various connectors and a vast range of visual elements like shapes and swim lanes, more flexible options could have been provided. In fact, it feels like the language representation is bound by the restrictions of the options provided by the MLW. Regarding the DSL editor, its layout is based on the same layout the language engineer uses. This is in most cases good enough, but it should be possible for the language engineer to build a editor from scratch using predefined components (e.g. toolboxes, buttons, message window). Furthermore, it is not possible to define or use other kind of interaction models (wizards, 3D editors, textual-diagrammatic mixed editors, etc).

**Constraints**

Code was used to define the constraints and also to specify when to check them. This solution allows for the definition of very complex constraints but does not have the simplicity of a solution like OCL (Object Constraint Language) present in EMF/GMF. A better mechanism to define constraints and specify their verification points should be provided. A solution could be the use of a new visual language built upon the DSL to define its constraints. This would allow the language engineer to build constraints using a graphic representation within the domain terms.

Given the cases where the same language is used by users with different profiles and access permissions, it should be possible to define different access levels. Each of these access levels would have less or more restrictive constraints. These more restrictive constraints could provide guidance for less experienced domain experts.

Another important aspect of constraints is their feedback to the users: when a constraint is violated, a message and/or visual change is passed to the user. This MLW does not make a distinction between actions resulting from constraint violation from normal edition actions, nor it provides a structured way to define the visual feedback. To do this kind of reactive actions, like color or form change when the introduction of an invalid value, it is again necessary to write code.

**Generic Elements**

One of the features of the ARL is the use of generic elements [5]. Each generic element defines an interface, and those interfaces correspond to the main building
blocks of the language. Basically, a loaded concrete element exists as an attribute of the generic element, and the generic element supplies a communication interface to it. This allows new building blocks to be created and used on the DSL according to their interface. Again, to implement this feature it was necessary to go beyond the normal use of the MLW. It required the use of code, and a lot of “digging” in the DSL Tools API and organization, to make it happen. This is another good example of a specific problem where the software language engineer needed to go beyond the normal use of the MLW and “dig” into its internal code. This situation is unavoidable, but it could be made easier if the MLWs have that in consideration, and also present their internal organization specified as a model.

**Editor complementary features**

The default implementation of the editor already solved some challenges like providing a quick way to edit an element’s name, automatic routing of the connections between language elements, and zoom in/out of the design surface.

Still, it was necessary to implement a set of visual functionalities whose sole purpose is to enhance the user experience. They add no new features to the language definition but are especially useful to manage the scalability of the models and to overcome the shortcoming normally associated to visual programming languages. The types of enhancement they provide are: center view on an element, hide/show of unused elements, instant visual feedback of model changes, and presentation of additional information in selected places. These features are not part of the initial DSL definition, but are important for the user experience when using the editor. We had no way to express these features on the MLW except using code, and they are part of the interaction model of the editor.

### 3.4 Language Semantic

The semantic of the DSL was defined by generating code from the models to the target framework. To do that, we used text based templates. They ended up being a good solution as they make a separation of concepts between the elements that are rendered directly on the output file, and the elements that are used to provide structure and dynamic behavior to the generation process.

In regards to the organization of the generation process, it was lacking a good distinction between implementation independent code generation and dependent code generation. In most situations, the code to be generated depends on the used implementation technology but the meaning of the generated code remains the same, it is only represented in a different way. Therefore, to use another target framework in the generation process it will be necessary to rewrite almost all of the generation process. This happens because the MLW does not provide a way to express the code generation mechanism. One of the requirements for the code generation was the generation of organized model-related code. This means that the identifiers used on the generated code are the same that are present on the model. This allows the use of a traceability framework to identify which part of the model caused the runtime error tracked to a specific line of generated code. This also has the side effect of allowing the generated code to be extended and customizable by developers. To do this, it was
necessary to do more work on the generation rules present on the text templates and to add additional verifications (e.g. valid C# identifier, prevent collisions between identifiers) to the models to ensure correct and valid identifiers for the elements to be generated. These verifications ended up being on the same level (of abstraction) as the model constraints, since it was not possible to distinguish between domain restrictions and code generation related restrictions.

4 Usability report

Fig. 3 presents an overview of the DSL solution that was created.

There are two actors: the software language engineer and the domain expert user. The software language engineer defines the metamodel and constraints used by the editor and the templates used for the code generator.

The editor, code generator and framework are methodology actuators. The model, code and application are artifacts. The user uses the front end (the editor) to help him express his mental model on a new DSL model. The user only contacts with the model representation given by the editor, and with the application itself. The implementation level details are supposed to be hidden to the user. All the transitions represented by the larger arrows are made without human intervention.

User comments

In spite of the measurable gains in productivity [6] there are still some issues that could be improved. From the evaluation process that was conducted to YDreams’s Augmented Reality domain experts, a set of usability problems were identified [6]. The interface details that were suggested to be enhanced are: option to keep a connector tool selected after a connection is made, and when making connections the drop space should be bigger. These details are related to the default generated editor functionalities that are not matured enough to consider this kind of advanced user support. The most important extra functionalities mentioned by the users were
enabling copy / paste of the elements, and ability to select multiple elements of the same type and configure them all at once. Also mentioned by the users was the support for optional forced placement of elements in the design surface. These functionalities make sense, and are important in providing a better productivity for the users. Therefore, they should be standard in every MLW. In DSL Tools, that is not the case, as the implementation of these functionalities requires some profound changes and customizations that need to be made by the language engineer.

Furthermore, the use of the toolbox entries is overwhelming. A solution to this was the creation of element specific context menus. Context menus are not part of the base functionalities provided by the MLW and had to be created using non-trivial code.

**Interaction Model**

The problems detected by the users are not related to our language definition. They are related to the MLW. Some of the minor problems related to Human-Computer Interaction (HCI) are easy to identify but hard to solve using the MLW. The existence of users with different profiles (such as programmers and designers) also makes it harder to define a single interaction schema. The best approach would be to provide different interaction schemas all based on the same interaction model.

When dealing with productivity, a very usable interaction model is required. Even if the DSL has the right concepts with the right representation, it is necessary that it also provides an interaction model for the users. The interaction model encapsulates the DSL syntax (abstract and concrete), and the users interact with the DSL through it. Fig.4 represents this interaction.

![Fig. 4. DSL Interaction Model.](image)

The language designer usually can dedicate more time to learning the language workbench functionalities. But his work could also benefit from a better interaction model. With a proper interaction model definition, it would be possible to automatically measure productivity using that same model (e.g. according to HCI usability guidelines). Therefore justifications for the users productivity while using a specific interface would be directly linked to the quality of the Interaction Model. It would not be necessary to conduct as many usability tests, speeding up the language development process.
5 Conclusion and Future Work

The main contribution of this work is to highlight the typical interaction problems found while using a specific MLW, and for that we used a diagrammatic DSL for the domain of Augmented Reality as a case study. When implementing DSLs with MLW there are problems that are not part of the DSL definition but part of the implementation technology. Given the increased interest in DSLs the current MLWs are reaching their expressiveness limits.

MLWs need to support the definition of the user interaction model, and are still not fully adequate to all the stages of the DSL development methodology. It is still necessary to use code to define important functionalities of the generated graphical editors, and doing that usually means that we lose some prototyping abilities provided by the MLWs.

This work will continue surveying DSL implementations using several MLWs (e.g. MS DSL Tools, MetaEdit+, EMF/GMF and the under development AToM3 [14]) in search of challenges and respective solutions to pave the way to the creation of a HCI-based model specification for the definition of MLWs.

Acknowledgments. A note of appreciation goes for the YDreams company for allowing the use of their framework as the target of the DSL.

References