Experiences on DSL Tools for Visual Studio*

Tomaž Kosar[1], Marjan Mernik[1], Pablo E. Martínez López[2]

[1] *University of Maribor, Faculty of Electrical Engineering and Computer Science*
Smetanova ulica 17, 2000 Maribor, Slovenia
{tomaz.kosar, marjan.mernik}@uni-mb.si

[2] *Universidad Nacional de La Plata, Facultad de Informática, LIFIA*
CC. 11, Correo Central
(1900) La Plata, Buenos Aires, Argentina
fidel@lifia.info.unlp.edu.ar

Abstract. *Within their application domains, domain-specific languages offer substantial gains in expressiveness, productivity, and ease of use, compared with general-purpose programming languages. Despite the many advantages of domain-specific languages, their use has been unduly limited, by a lack of support in developmental environments. Recently, Microsoft introduced some support by constructing domain-specific languages with a plug-in 'DSL Tools for Visual Studio'. This paper gives language designers tips on developing a domain-specific language using this tool and describes the experiences of an end-user of constructing a language. Another contribution of this paper is a comparison of tools with the traditional approach by the implementation of a domain-specific language, done on the same representative language.*

Keywords. domain-specific visual language, domain-specific language, language workbenches, source-to-source transformation

1 Introduction

A domain-specific language (DSL) is a language designed in such a way that it provides a notation tailored towards an application domain, and is based only on the relevant concepts and features of that domain [8]. As such, a DSL is a means of describing and generating the members of a family of programs in the given domain, without the need for knowledge about general programming.

However, language design and its implementation have always been considered as a very difficult task – this is perhaps one of the reasons why the popularization of DSLs has not reached the expectations of many DSL researchers. Various implementation approaches also exist when developing domain-specific language. It is hard to decide which approach to take, and only a few studies on their comparisons are found in the literature [5]. Another reason for this poor popularization is the lack of language support in the developmental environments.

The traditional development of domain-specific languages is still based on the code world, which often leads to poor domain understanding. Domain-specific visual languages (DSVL) has been introduced to overcome this, where graphical representations are used to develop a system performing task similar, but much more expressive, than programming by code [9]. The easiest way to imagine DSVLs is by comparing it to UML, that enables users to model/program in a visual manner. Unlike UML, DSVLs enable end-users to program with concepts relevant for specific domain. DSL Tools for Visual Studio [1] is a representative example of those tools.

In this paper, the stress is put in developing a domain specific visual language using DSL Tool for Visual Studio [1]. However, traditional DSL implementation approaches

*This work is sponsored by bilateral project "Implementation of DSLs: Evaluation of Approaches" (code ES/PA02/E01, BI-BA/03-05-002) between Slovenia and Argentina, part of the program SECyT-MESS.

```
Class FDL_Menu {
  begin_spec(Menu, strResult)
    begin_feature
      Menu        :all (opt(Appetizer), MainCourse, opt(Salad), Dessert)
      Appetizer   :all (cheese, smokedHam, olives)
      MainCourse:one-of (DishA, DishB, DishC, DishD)
      DishA       :all (backedPotatoes, roastBeef)
      DishB       :all (golaž[1], polenta)
      DishC       :all (mashedPotatoes, stewedSteak)
      DishD       :all (asado[2], chorizo)
      Salad       :more-of (lettuce, tomato, cucumber)
      Dessert     :one-of (gibanica[3], flanConDulceDeLeche[4])
    end_feature

    begin_constraints
      exclude smokedHam
      include asado
      asado requires tomato
      exclude flanConDulceDeLeche
    end_constraints
  end_spec

  static void Main(){
    Console.WriteLine(strResult);
  }
}
```

Figure 1: FDL description of Menu

```
one-of(
      all(asado, chorizo, lettuce, tomato, cucumber, gibanica ),
      all(asado, chorizo, lettuce, tomato,           gibanica ),
      all(asado, chorizo,          tomato, cucumber, gibanica ),
      all(asado, chorizo,          tomato,           gibanica )
      )
```

Figure 2: The meaning of program Menu

should not be underestimated, so a very simple comparison of this tool with those approaches is also presented.

The organization of the paper is as follows. A small, representative language for case study is discussed in 2. The characteristics of DSL Tools for Visual Studio are described in Section 3. Related work is discussed in Section 4. Finally, a synthesis and concluding remarks are presented in Section 5.

2 Case study

In order to show the language construction with DSL Tools for Visual Studio, we need to choose the appropriate domain to solve. The Feature Description Language (FDL) [4] has served this purpose well in the past [5, 6, 7], and is also used as a representative language in this paper.

The language FDL describes features and their hierarchical composition. A feature can be `atomic`, `optional`, `one-of`, `more-of`, or `all`. Atomic features are the basic construct; optional features can be present or not; `one-of` means that exactly one of its child features can be present in the configuration; `more-of` is similar to `one-of`, except that an arbitrary number of child features can be chosen from the set; finally, `all` means that all its children have to be present.

The meaning of the FDL program Menu, in Figure 1, is the set of all possible configurations. There are 256 various compositions of the menu in this example. When we add constraints to it (see the part begin_constraints in Figure 1), the original 256 possibilities are reduced to just 4. In Figure 2, every all feature represents a possible solution to our problem. Interested readers can find more details about FDL in the literature [4, 5].

3 DSL Tools for Visual Studio

The current version of DSL Tools is 1.0 and can be added to Visual Studio 2005 as a plug-in. The toolkit contains support for the metamodeling environment, the modeling environment, and the generation of code. An overview of the toolkit is given in this section.

3.1 Metamodeling Environment

A metamodeling environment is defined as a set of rules, notations, and constructs, which are further used by an end-user in modeling environment. These rules and constructs should be strictly defined, without any ambiguities or errors. A metamodel consists of the following definitions: domain model notation, constructs, and relationships between model notation and the constructs.

Definition of Domain Model Notation
Domain model notation represents the structure of a domain and is composed in a visual manner. It shows the inner structure and relationship between classes, which are defined in a document with *.dsldm* extension (DSL Domain Model). Constructs which can be used in visual mode are class, property, embedding, reference, and inheritance.

In a similar way as in object-oriented programming, a class represents a set of objects sharing the same structure, behaviour, and

[1] Traditional Slovenian food, stewed beef made with paprika and onions.

[2] Slow cooked barbecued cow meat, national speciality of Argentina.

[3] A cake with poppy seeds, curd cheese, walnuts and apples, national speciality of Slovenia.

[4] Traditional Argentinian dessert, custard made of milk and egg yolks with a jam made of long boiled sugared milk.

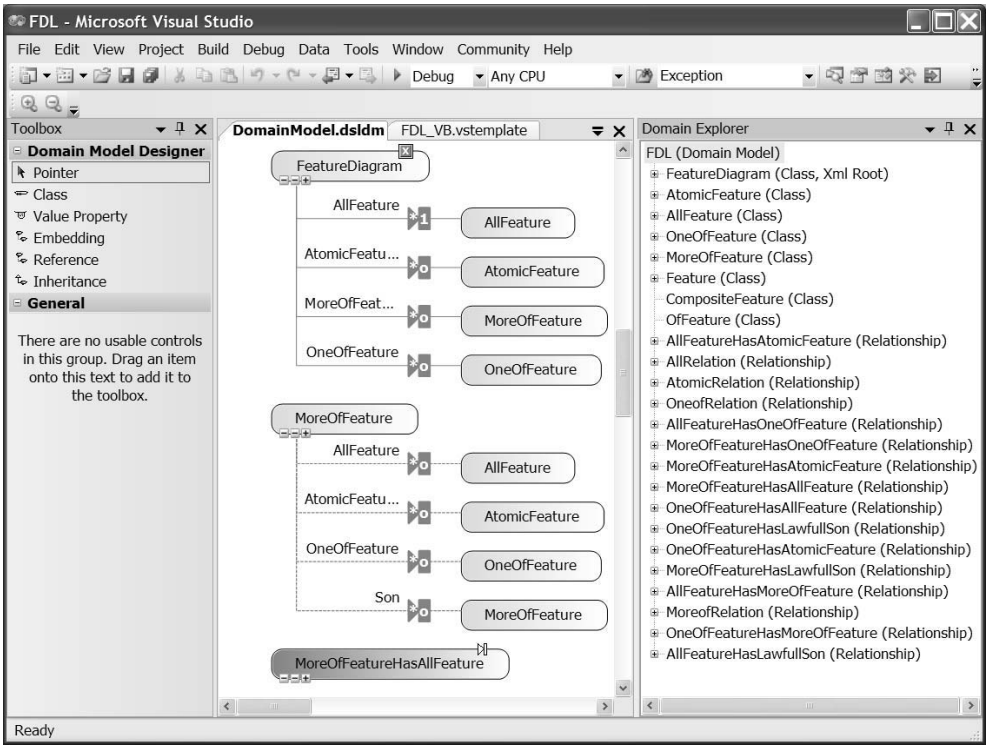

Figure 3: Metamodel environment in DSL Tools

relationships. Classes can have sets of attributes, which define the properties of a construct and can be used by end-users to customize their own needs for the language under construction.

The basic functions of 'embedding' and 'reference' are to connect classes of DSL. The difference between these two links is in semantics. If we have classes A and B, 'embedding', means that class A includes B (B is part of A), while 'reference' means that class A uses B. There are some rules as to how classes can be connected. Links between classes are graphically represented by triangles and rectangles, which represent the source and type in the relationship. Both source and type are defined by a cardinal number (0,1,+ or *). Another link between classes is inheritance. Again, graphical notation and meaning is analog to the one from the object-oriented paradigm.

The notation of a FDL domain model is given using the constructs explained above (see Figure 3). A domain model represents the syntax of a language. If the reader is familiar with EBNF, she can draw some correlations using domain model: the meaning of classes and non-terminals is similar, attributes corresponds to terminal symbols, and links pointing from individual classes can be seen as productions in EBNF notation.

A semantically valid language is obtained by a *validation model*, where attribute values and link names (they must be unique) are checked. Validation process checks also if the links produce cycles. For validation purposes a new document must be defined in a project, which validates a specific class or attribute. For instance, the name of FDL's atomicFeature must start with a lowercase letter. Validation of the class representing atomicFeature is part of the validation model.

Definition of Domain Constructs

Constructs definition is defined in a *.dsldd* document (DSL Designer Definition). Unfortunately, this task is quite demanding since everything must be done in XML notation and no supporting visual tool exists for this task.

Elements *notation* and *objectModels* are concerned with defining a domain model. Furthermore, element *notation* consists of elements *diagrams* and *diagramMaps*. The element *diagram* defines the types of diagram. Also, *diagram* defines symbols, which will be present in the modeling environment as a toolbar, modeling shapes, and connectors.

Relationships between domain notation and constructs

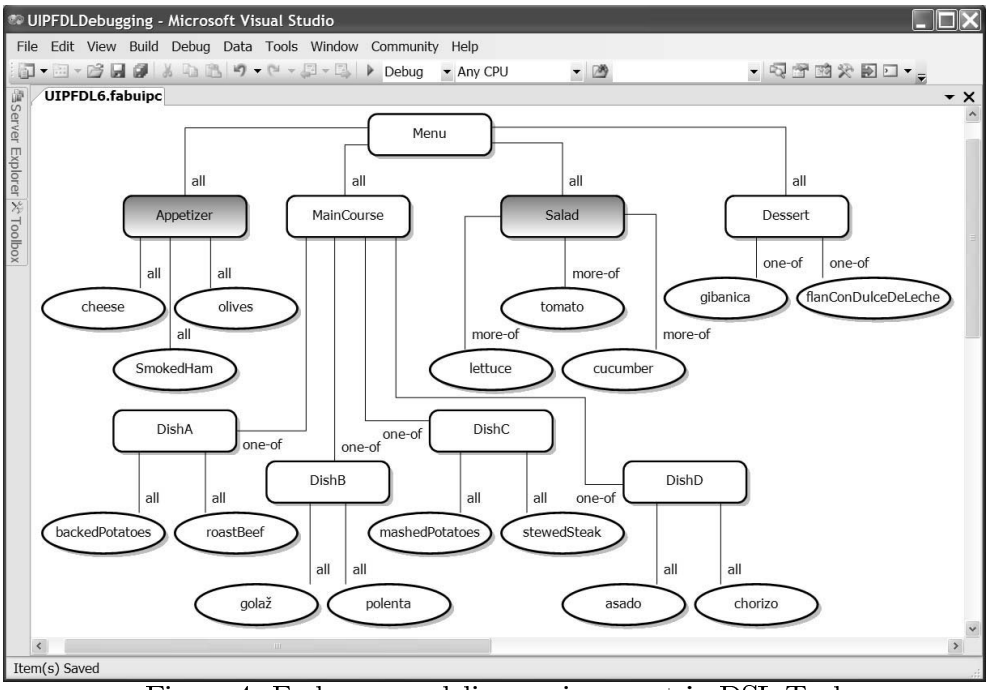

Figure 4: End-user modeling environment in DSL Tools

Shapes and connectors from the toolbar need to contain a reference to constructs, otherwise the modeling environment will stay empty in spite of a user selection of the toolbar construct. This definitions are placed in the same document as constructs and toolbar. They can be found under section *diagramMaps*. In the element diagramMaps shapemaps and connectormaps definitions can be found.

3.2 Modeling Environment

Before starting with a modeling environment, there is a need to transform metamodeling language into source code. This procedure transforms a metamodel into code, representing an executable end-user modeling environment. In visual mode, all constructs (shapes and connectors) are present. The end-user can drag-and-drop a construct from the toolbar to the working place, where a DSL program is then constructed. The end-user can change the initial values of the construct properties.

An example of FDL visual program is depicted in Figure 4 (whose text version appeared in Figure 1). The FDL end-user visual program expresses that a Menu is composed of several parts (indicated by the tree-like structure in the diagram). Some of these parts are optional (indicated by the grey rounded rectangle in the diagram), and others are mandatory (empty rounded rectangle in the diagram). Some features are atomic (indicated by ellipse leaves of the tree in the diagram) and some others are composed of subfeatures – this is indicated by a tree-like structure and their names start with capital letters. The type of tree-like structure is defined by link names. The meanings of all, one-of, and more-of were given in Section 2.

After specifying a visual program, the end-user can start a transformation tool in order to run a DSL program.

3.3 Generation of code

The toolkit DSL Tools for Microsoft Visual Studio 2005 is based on transformations of templates. In the literature, we can often find this kind of implementation marked as assimilation [2] or source-to-source transformation [6, 8]. In DSL tools, source code is generated from text templates (file extension *.cstemplate*), which take DSL visual program and produce a source code in the chosen language (in FDL case, C# has been produced). A tool for generating source code includes:

- Text Template Transformation Engine,
- Host (interface between engine and user-environment), and
- Directive Processors (represents a link between directives and text templates).

Text templates are documents which includes a mixture of text, control, and directive

IDE	ToolKit	URL
Eclipse	Eclipse Modeling Framework	http://www.eclipse.org/emf/
NetBeans	Project Ace	http://research.sun.com/features/ace/
IntelliJ	Meta Programming System	http://www.jetbrains.com/mps/
Visual Studio.Net	DSL Tools For Visual Studio	http://msdn.microsoft.com/vstudio/DSLTools/

Table 1: DSL support in integrated development environments (IDEs)

blocks. The latter two, are included in the sorrounding marks <# and #>. All transformations occur between these two marks.

Directives give instructions to the transformation engine. On the other hand, control blocks are programming code blocks in defined general-purpose language. Text blocks are the only blocks that do not compile and are directly written to the output.

4 Related work

In the previous section it has been shown how to obtain a DSVL with a Microsoft's toolkit. DSVL opens up the possibility for language designers to develop a system where end-users can program using graphical representations relevant for a specific domain. Therefore, a DSVL can be much more expressive as DSL. But the question might arise – how much more development effort is needed to construct a DSVL?

A rough measure of development effort can be obtained in terms of the number of effective lines of code (eLOC), that is, all lines that are not blanks, standalone-braces or parentheses. The definition of domain constructs is written in XML code and for FDL consists of 750 eLOC. Templates for code generation contain an additional 200 eLOC. A DSL domain model consists of 1000 XML eLOC, but visual notation is developed graphically. Therefore, eLOC is unsuitable for measuring effort on building the domain model of a DSL.

Various techniques exist for implementing a DSL: preprocessing, embedding, compiler/interpreter, compiler generator, extensible compiler/interpreter, and commercial off-the-shelf [8]. One of the most demanding, but also very rewarding, implementation approach of DSLs is by the extensible compiler/interpreter approach [6]. Here, the DSL developer requires access to the definition of the base language notation in order to incorporate the DSL syntax definitions. Compilers, that allow incorporation of new features to the base language, are called open compilers. Mono C# compiler is one of them. The extension of Mono C# compiler with representative domain-specific language FDL has been carried out in [7]. Using extensible compiler Mono 1.0.1, it takes 136 eLOC added to the specification of the compiler and the hand-coded lexical analyzer in order to incorporate FDL into it. After extending the compiler using FDL, we are able to write the C# program as presented in Figure 1. Class FDL_Menu is a C# class, but it contains domain-specific constructs to define a Menu program. For further details on how to incorporate DSL constructs to Mono, we refer the reader to the literature [6, 7].

The tool MetaBorg [2] helps to include a particular DSL notation back into a GPL by assimilating (not extending) it into the language, i.e. domain-specific notation is translated into some existing APIs operations. GPL code is obtained, as a result of assimilation. Similarly, DSL tools for Visual Studio contains a transformation step that generates source code in the base language.

Recently, many IDEs received a support for language construction and some authors name these IDE tools as Language Workbenches. Examples of these tools [3] are JetBrains's Meta Programming System, and the tool described in this paper – Microsoft's DSL Tools for Visual Studio. These tools share the same goal – the construction of a domain-specific language. On the other hand, other workbenches have diverse methodologies on how to obtain domain-specific language. The explanation of these techniques is outside the scope of this paper. Suitable references can be found in Table 1.

5 Conclusion

In this paper we have described the toolkit DSL Tools for Visual Studio that enables a developer to design a visual language. The

toolkit supports metamodeling, generation of code, and visual programming for the end-user. Guidelines are given to practitioners on the construction of a visual language.

Our long-lasting interest is in DSL implementation approaches. Recently, visual construction of DSLs became highly popular. In the future we would like to extend the comparison on DSL implementation approaches with DSVLs. This paper is the first step towards this goal. A simple comparison with the traditional DSL implementation approach has also been carried out. This comparison must be extended with a measurement model similar to previous experiments [6] comparising DSL's implementation approaches from the perspective of implementation and end-user effort.

However, research shows that a language designer can develop a DSL much faster than a DSVL. Using DSL Tools for Visual Studio is quite demanding, since managing XML document takes a lot of development time, and no support tools exist. On the other hand, end-user time and perception should not be underestimated. Visual programming allows end-users to master the environment and usage of domain concepts more quickly.

Of course, developing a DSL is not a solution for every computable problem. In combination with a proper application programming library, any general purpose language can act as a DSL. The programmer must decide which approach to use bearing in mind, not only the characteristics of application programming library or DSVLs, but the capabilities of the languages he masters and his knowledge and history, in order to obtain better results in terms of efficiency, effort, development time, and/or any other dimension where he wants to obtain good results.

Shortcomings of application libraries such as limited domain-specific notations and the inability of domain-specific analysis, verification, optimization and transformation restrict their usefulness. With our work we would like to promote usage of DSLs to the practitioners and help them find alternative solutions for their problems. Limitations of common solutions can be overcame with a toolkit such as DSL Tools for Visual Studio.

References

[1] DSL Tools for Visual Studio, available at `http://msdn.microsoft.com/vstudio/DSLTools/`.

[2] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of OOPSLA'04*, pages 365–383. ACM Press, October 2004.

[3] G. Caprio. Domain-specific languages & dsl workbench. *Dr. Dobb's Journal*, 31(5):38–41, May 2006.

[4] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[5] T. Kosar, P. E. Martínez López, P. A. Barrientos, and M. Mernik. Experiencing diverse implementation approaches for domain-specific languages. Technical report, 2005, `http://marvin.uni-mb.si/technical-report/RAJ-T0501.pdf`.

[6] T. Kosar, P. E. Martínez López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *To apear in Information and Software Technology*, 2007.

[7] D. Krmpotić, T. Kosar, M. Mernik, and V. Žumer. Extending open compilers. In *Proceedings of the 27th Internation Conference on Information Technology Interface (ITI 2005)*, pages 645–650, 2005.

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

[9] J.P. Tolvanen and S. Kelly. Modelling languages for product families: A method engineering approach. In *Proceedings of OOPSLA workshop on Domain-Specific Visual Languages*, 2001.