Using DEViL for Implementation of Domain-Specific Visual Languages

Carsten Schmidt, Uwe Kastens, Bastian Cramer
University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
{cschmidt, uwe, bcramer}@upb.de

Abstract

The implementation of visual languages requires a wide range of conceptual and technical knowledge. A hand-implementation of a visual DSL is often too expensive for a domain with a limited user comunity. We present the DEViL system, that generates complete language implementations from high level specifications. DEViL allows to implement visual languages with limited effort and limited expert knowledge. We give a general overview of DEViL's specification mechanisms and discuss our experiences as well as open issues regarding visual language implementation.

1 Introduction

Domain specific languages are well-suited to encapsulate domain-specific knowledge and to support domain-specific program development. We believe, that *visual* languages become more and more important in this respect. Visual languages are advantageous for high-level structures or models. A well-known example is UML. Visual languages can further adopt established notations in an application domain, so that the gap between the user's thoughts and the provided language constructs is reduced.

However, the implementation of visual languages is often a challenging task, that requires a great amount of technical and conceptual knowledge and effort. The user community of domain specific languages is often too small to justify a language implementation by hand. We developed the DEViL system (Development Environment for Visual Languages), that generates language implementations from high-level specifications. Expert knowledge with respect to language design, graphical user interfaces, interaction and visual layout mechanisms is encapsulated in DEViL, so that visual languages can be implemented with limited effort and without needing specific knowledge concerning implementation issues.

DEViL generates complete language environments including a visual structure editor, analysis components and code generators as can be seen in figure 1. The specification concept is based on attribute grammars. They are used to specify the graphical representation, as well as analysis and code generation components. Since DEViL is based on the well-known Eli system [2], all Eli tools can be used for language processing.

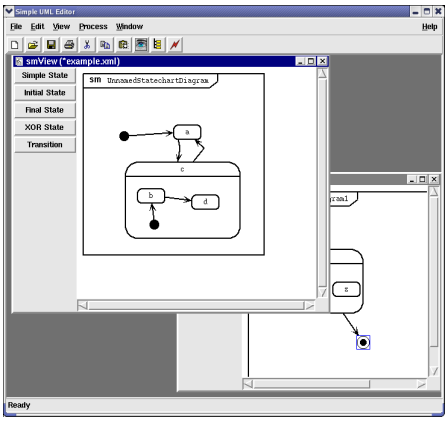

Figure 1. A generated language implementation with the multiple-document environment

DEViL is the successor of the VL-Eli System [3], that has been successfully used for many language implementations, even in industrial projects [4]. The most important advances from VL-Eli to DEViL are the conceptual distinction between semantical structure, editable structure and representation structure, a language to specify these structures and their coupling, as well as new specification languages for specific kinds of representations. Chapter 2 gives an overview about the general specification concepts. Chapter 3 discusses our experiences with this approach and Chapter 4 mentions open issues.

2 The DEViL System

DEViL distinguishes three specification aspects: the abstract syntax, visual representations and the code generation modules (Figure 2). The abstract syntax describes the lan-

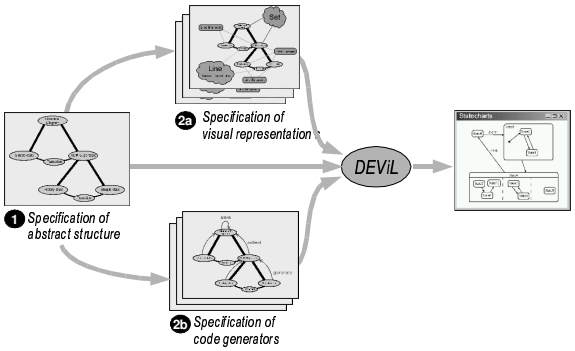

Figure 2. Main specification aspects in DEViL

```
CLASS Statechart {
  states: SUB State*;
  transitions: SUB Transition*;
}
ABSTRACT CLASS State {
  name: VAL VLString;
}
CLASS SimpleState INHERITS State {
}
CLASS XORState INHERITS State {
  subStates: SUB State*;
}
CLASS Transition {
  from: REF State;
  to: REF State;
}
```

Figure 3. Structure specification for Statechart Diagrams

guage constructs and its combinability without setting up a concrete representation. The visual representation part defines a set of views on the abstract structure. Each view defines a concrete representation as well as interactive mechanisms to modifiy the corresponding structure part. Often, there are multiple views on the abstract structure. They can be overlapping and so highlight different aspects of the structure or they can be organized as overview and detail views. Code generation modules are used to implement the analysis and transformation of the visual program. Analogous to visual views, there can be multiple code generation modules for different purposes. Views and code generation modules are specified by attribute grammars. The underlying tree grammars for these specifications are derived from the abstract syntax, but can be adapted to the special needs of the corresponding view or code generation modules. The individual modules are completely independent from each other, so that they can be maintained separately. In the following, we explain the three specification parts in more detail.

2.1 Abstract Syntax

The specification of the abstract syntax is formulated in a tailored specification language called DSSL (DEViL Structure Specification Language). Figure 3 shows an abstract syntax for statechart diagrams. It models a structure tree containing values and cross-references. The notation is very similar to object oriented programming languages like Java. It bases on modeling concepts like classes, attributes and inheritance. Every non abstract class specifies a language construct. Attributes are written inside the body of a class declaration. For instance the class *Statechart* in figure 3 has the attributes *states* and *transitions* which hold a set of subordinated language objects.

DEViL distinguishes between three types of attributes: VAL-Attributes (like *State.name*) store values and have an associated data type like *VLInt* or *VLString*. SUB-Attributes (like *Statechart.states*) store substructures that are instances of the specified class. A cardinality can be specified after the type, for example "*" means zero or more elements. REF-Attributes store references to other language objects. They model cross relations in the abstract structure tree.

Figure 3 defines the structure of statecharts as follows: A *Statechart* has a list of *states* and a list of *transitions*. *States* are either of type *SimpleState* or of type *XORState*. *XORStates* contain a list of *States* again. The *Transitions* are part of the *Statechart* language object. *Transitions* itself have two reference attributes (*from, to*) to store start and ending points.

2.2 Visual views

Generated language implementations can provide multiple views on the abstract structure. Each view defines a concrete representation for a part of the abstract structure. A view specification consists of (1) declarations, that determine which part of the structure is visualized and define the mapping to a tailored tree grammar as basis for layout computations, (2) an attribute grammar, that defines the graphical representation and it's layout, and (3) a definition of toolbar buttons for graphical interaction.

The attribute grammar is the most important part of a view definition. The attribute computations compute layout attributes like positions and dimensions and paint graphical primitives into a canvas. They further integrate invisible active elements into the representation, so that the language user can edit the structure. For example insertion points are added to highlight positions where new language objects can be inserted. When the user interacts with the graphical representation, a corresponding modification is applied to the abstract structure. This modification triggers an update of the involved views, which forces the attribute evaluator to recompute the graphical representation (according to the model-view paradigm).

To raise the specification of graphical views to a higher level, there are specification modules and DSLs, that are

```
SYMBOL Statechart_states
   INHERITS VPFormElement, VPSet
COMPUTE
  SYNT.formElementName = "body";
END;

SYMBOL SimpleState INHERITS
  VPSetElement, VPForm, VPConnectionEndpoint
COMPUTE
  SYNT.drawing = ADDROF(SimpleStateDrawing);
END;
```

Figure 4. Pattern-based view specification

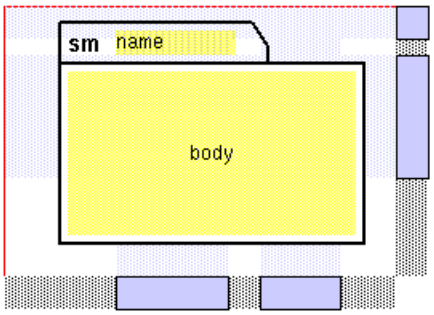

Figure 5. Generic drawing for a Statechart Diagram

built on top of the attribute grammar method. The most important concept is the library of so called visual patterns. Visual patterns are reusable implementations of common representation concepts like lists, sets, tables, trees, line connections or forms. They are defined in terms of visual roles, that can be associated to symbols of the tree grammar. For example, the *set* pattern consists of two roles: *VPSet* represents a graphical set as a whole and *VPSetElement* represents members of such a set.

Figure 4 shows a part of a pattern-based view specification. By assigning roles like "*VPFormElement*", "*VPSet*" or "*VPSetElement*" to certain grammar symbols computations are applied, that describe the whole depiction. The meaning of the roles can easily be understood. *Statechart_states* inherits from *VPSet*, because this attribute represents a visual set of states, which can be placed at arbitrary positions inside the set region. The symbol *SimpleState* inherits from the role *VPSetElement*, because instances represent members of the mentioned set. The representation of *SimpleState*-nodes itself is defined by the Form-pattern, so it additionally inherits from *VPForm*. It further inherits from the role *VPConnectionEndpoint*, because states can be endpoints of line connections.

Inherited attribute computations can be overwritten and existing computations can be extended to influence details of the graphical representation. For instance in the context of *SimpleState* the inherited *drawing* attribute of the role *VPForm* is overwritten. The value of this attribute influences the details of the graphical representation.

The application of visual patterns to language elements is very similar to the mechanism of CSS in combination with HTML/XML, but the DEViL variant is more powerful and more flexible. A more detailed discussion of attribute grammars and visual patterns in the context of view specification can be found in [3].

There are some DSLs, that supplement the specification mechamism of visual patterns. So called "generic drawings" are used to define representation details for language objects based on the form pattern in a WYSIWYG style. Figure 5 shows the generic drawing for a Statechart Diagram. It consists of two containers ("name" and "body") in which sub elements of the language construct are displayed. In this case they act as placeholders for the name and the state-set of statecharts. Miscellaneous line, text and rectangle primitives are used to define the concrete representation of this construct. So-called "expansion intervals" (rectangular bands at the right and lower borders) are used to define the expansion behavior of the representation. Graphical regions covered by expansion intervals are linearly expanded, if more space is needed for subordinated representations.

Another DSL, called SLTR ("Specification Language for Textual Representations") can be used to specify textual sub-representations in visual views. Basically, "unparsing rules" are associated to the classes of the abstract syntax. This specification part is sometimes called "format syntax". Internally, SLTR specifications are translated into applications of visual patterns. In this way, visual and textual representation parts can be combined in a well-defined manner.

2.3 Transformation

Code generators, that transform the visual program into a target representation are specified by attribute grammars, too. The attributes store analysis results (e.g. deduced types of language constructs) or code fragments, that are stepwise combined in order to construct the complete target tree. To implement a code generation module, all libraries and DSLs of the well-known Eli system [2] can be used.

A typical example for a DSL used in the context fo code generation is PTG (pattern based text generator). It is used, to generate source code of a textual target language, e. g. C-Code or SQL statements. The user defines output text fragments, that are interspersed with insertion positions in which parameters or instances of PTG Patterns are substituted. PTG generates C functions out of these text patterns, which can be called in attribute computations in order to build up the target code. With PTG, attribute computations can be formulated independently of the textual output order without loosing efficiency. Additionally, PTG supports powerful methods for pretty printing to beautify the target code.

3 Experiences

DEViL and its predecessor VL-Eli have been successfully used for many language implementations, even in industrial projects [4]. Additionally we have systematically evaluated the DEViL system in respect to learnability, user acceptance, language implementation effort and applicability in large projects. The results are quite encouraging. Users need a considerable amount of time to get used to DEViL, but after that they are satisfied and exhibit a greatly improved development speed. The representation of small visual languages (about 10 language constructs) can be completely implemented within one day.

The visual pattern-based specification allows to construct sophisticated DSLs without getting in touch with implementation details of graphical user interfaces or interaction mechanisms.

We evaluated the scalability of the approach in the context of a student project. Eight students designed and developed a quite large visual language (about 90 language constructs) in a period of one year. The developed language is used to specify dynamic web applications using domain-specific concepts like web-pages, persistent data tables and actions. Specifications are translated into a combination of PHP, HTML and SQL code.

The evaluation shows, that the specification concept is very well suited for large projects. Individual specification aspects (view- and transformation modules) can be implemented quite independently from each other. The underlying abstract syntax serves as interface for implementation as well as foundation for design discussions. Another insight from this project was, that the students are not able to overlook all reqirements of complex languages in the initial design phase. That's why DEViL's support for incremental development is especially important: When new requirements are recognized the abstract syntax and the graphical representation can be changed or extended with limited effort.

A last remarkable benefit of automatically generated language environments is the automatic generation of "standard functionality" like finding usages in the context of definitions, searching for character sequences, cut-and-paste, printing, zooming etc. A complete set of standard functions is mandatory for realistic use of the implementation. Ad-hoc implementations are often poorly usable because they lack such features. Even though object-oriented frameworks can provide some of the mentioned features, it is hard for them to cover features, that are closely related to the language, for example finding usages of definitions.

4 Open Issues

According to our expericences, automatic generation of visual language implementations works quite well, even for complex application domains. However, we are often faced with the fact, that visual DSLs (as well as textual ones) contain domain-independent sub-languages. Examples are mathematical expressions or full-grown general purpose languages like Java or C. The latter are often used as "escape mechanism", to provide sufficient flexibility.

At present, sub-languages like mathematical expressions are manually implemented just like other DSL constructs. Fragments of full-grown general purpose languages are currently stored as simple character sequences and are blindly inserted into the generated target code. Even though this pragmatic approach works, it has considerable drawbacks. Either domain-independent sub-languages must be re-implemented in the context of the DSL, or they are completely neglected by the implementation, which leads to poor integration and awkward error messages. One would therefore dream of a reusable language kit, that allows for easy combination of different DSLs and general purpose languages. This would be an important step towards the era of language oriented programming [1]. However it is difficult to realize such a concept, because the involved languages are interweaved in a quite complex way.

References

[1] Sergey Dmitriev. Language oriented programming: The next programming paradigm. JetBrains 'onBoard' electronic monthly magazine, 2004. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/.

[2] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli system. In Kai Koskimies, editor, *Proceedings of 7th International Conference on Compiler Construction CC'98*, number 1383 in Lecture Notes in Computer Science, pages 294–297. Springer Verlag, March 1998.

[3] Carsten Schmidt and Uwe Kastens. Implementation of visual languages using pattern-based specifications. *Software - Practice and Experience*, 33:1471–1505, December 2003.

[4] Carsten Schmidt, Peter Pfahler, Uwe Kastens, and Carsten Fischer. SIMtelligence Designer/J: A visual language to specify SIM toolkit applications. In *Proceedings of Second Workshop on Domain Specific Visual Languages (OOPSLA 2002)*, Seattle, WA, USA 2002, 2002.