Generating visual structure editors from high-level specifications

Carsten Schmidt, Bastian Cramer, Uwe Kastens
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
{cschmidt, bcramer, uwe}@uni-paderborn.de

Abstract

The implementation of visual languages requires a wide range of conceptual and technical knowledge from issues of user interface design and graphical implementation to aspects of analysis and transformation for languages in general. We present the DEViL system that incorporates such knowledge. It generates complete language implementations from high level specifications. DEViL is the successor of the VL-Eli system [13] and includes new concepts, that allow to implement even challenging visual languages. We give a general overview of the provided specification concepts and languages and emphasize the underlying principles for generation and reuse.

1. Introduction

Visual structure editors have an important role in modeling systems, specification of software, and in specific application domains. Creating such editors is often a challenging task, that requires a great amount of technical and conceptual knowledge and effort. Visual languages are most effective for domain specific purposes. However, the user communities of many domain specific languages are too small to justify a language implementation by hand. We developed a system called DEViL (Development Environment for Visual Languages), that generates visual structure editors from high-level specifications. Expert knowledge with respect to language design, graphical user interfaces, interaction and visual layout mechanism is encapsulated in DEViL and can be used by non experts.

DEViL is based on concepts and systems of compiler construction derived from the Eli system [12]. It is designed for language developers to help them implement even complex domain specific visual languages (DSVLs) with rather challenging graphical representations. DEViL's focus therefore differs from that of meta-modeling tools, which usually have limited flexibility in respect of the graphical representation.

DEViL generates complete language environments including a visual editor, analysis components and code generators. It is a successor of the VL-Eli System [13], that has been successfully used for many language implementations, even in industial projects [18]. The most important advances from VL-Eli to DEViL are the conceptual distinction between semantic structure, editable structure and representation structure (Section 3.1 and 3.2), a language to specify these structures and their coupling (Section 3.1), as well as new specification languages for specific kinds of representations (Section 3.2).

In this paper we describe the DEViL environment with a special focus on the underlying principles for generation and reuse. A set of generators produce the implementation of the structure models, the attribute computation, and the visual representation. Libraries and frameworks are used to encapsulate GUI components and allow the user to integrate application specific code. The generation process itself is controlled by build tools, that coordinate the generators and compilers and provide effective caching mechanisms as well as error tracing.

In Chapter 2 we present the application domain of DEViL. We characterize general properties of language implementations and introduce an UML editor as running example.

Chapter 3 gives an overview of the DEViL system. We show how the individual generators cooperate with each other and how reusable and extendable specification modules establish a powerful, flexible and easy to use specification mechanism for visual representations.

Chapter 4 discusses the fundamental reuse techniques applied in DEViL. We emphasize the multi-level specification method, the combination of specialized languages and discuss some interesting realisation techniques. These concepts are not restricted to the presented domain but can be applied in other contexts, too.

Chapter 5 presents related work and Chapter 6 concludes this article.

2. Visual structure editors

With CASE Tools, a special variant of visual structure editors, software can be specified on a high abstraction level. These tools allow a rapid prototyping approach in a comparable less expenditure of time. Also in other sectors of development visual structure editors help users to expand their productivity e. g. in hardware design. Hence, many structure editors are needed, each dedicated for its particular language. As methods and techniques for their implementation are quite well understood, it is not acceptable to create each one from scratch. Tools can be used to generate them from high level specifications.

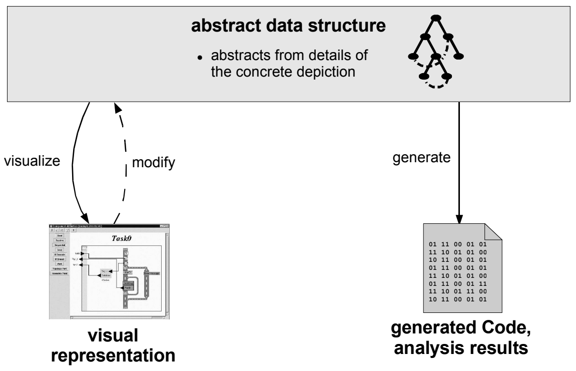

abstract data structure
• abstracts from details of the concrete depiction
visualize
modify
generate
visual representation
generated Code, analysis results

Figure 1. The abstract data structure as central unit in structure editors

Graphical structure editors use an abstract data structure as central program representation (Figure 1). Visual representations are derived from the abstract structure and displayed to the user. The user can interact with the graphical representation and thereby manipulate the abstract structure. In addition to the edit cycle, a language implementation must be able to analyse and transform the constructed programs. Often language implementations include generators that produce source code of a programming language.

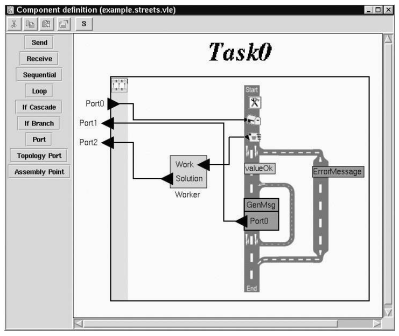
An important implementation aspect is the graphical representation. Visual structure editors use visual properties like spatial placement, line connections or special markups to visualize a domain specific language . The visualization helps the user to understand even complex structures. Figure 2 shows screenshots of visual editors with different graphical styles. Figure 2a shows a program in Streets [11], a visual language for modeling parallel processes. Its central visual paradigm is a road system representing the control flow using nested control structures. Figure 2b shows an electronic circuit. It consists of icons for electronic devices like diodes or resistors, that are composed by orthogonal line connections. Figure 2c is a screenshot of the so-called Derivation Tree Assistant, that aids visual construction of derivation trees and uses automatic tree layout. Figure 2d is a part of a PaderWAVE [20] specification. PaderWAVE is a visual modeling language for web applications.

All these editors have been generated by the DEViL system and more examples can be found on the DEViL website [16]. The styles of the visual representation vary pretty much. Some contain deeply nested structures (Streets), some are based on graph-like structures with arbitrary node positions and orthogonal or direct line connections (Electronic circuits, parts of Streets), some use visual tree structures with automatic layout (Derivation tree assistant), and some languages are even based on table-like representations (parts of PaderWAVE). Some languages introduce complex graphical icons and metaphors (Streets, Electronic circuits), whereas others use geometric primitives (Derivation tree assistant) and some use colors to dintinguish language elements (Streets, Derivation tree assistant, PaderWAVE). Textual representations as part of graphical structures are also very common (e.g. OCL in UML diagrams). Especially the layout of the graphical representations is a challenging task.

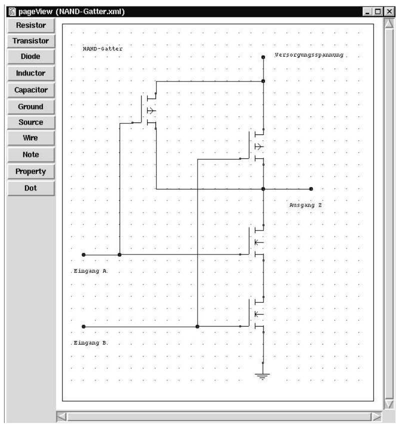
Another important aspect of a structure-editor is the multiple document environment it is integrated in. The environment coordinates different views on a program and provides important features like saving or loading, cut-and-paste operations, undo, printing, searching and navigation, display of error and information messages and so on. A complete set of those functions is vital, because they determine whether a structure editor is usable in practice.

DEViL generates most of the above mentioned features without additional effort of the language developer. Figure 3 shows a screenshot of a generated UML Editor with three different views on a Statechart structure. There is a graphical view, an abstract tree view and an error view. All views show different aspects of the underlying common structure. Multiple documents can be loaded at the same time and the editor supports a rich set of navigation and interaction mechanisms. We use this UML editor as a running example in the remainder of this paper.

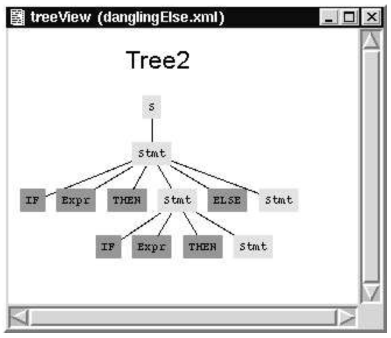
Some languages impose an additional implementation requirement on a structure editor. Sometimes there can be multiple representations of a single semantic object with individual representation properties. An example are classes in UML diagrams. Multiple representations of a specific class can occur in different class diagrams and even in a single class diagram. Thus the language implementation must distinguish between semantic objects and representation objects. Both kinds of objects carry different information: The semantic objects store the name and attributes of a class, the representation objects store the position and size of the individual class representations. To be prepared for this situation, DEViL distinguishes two variants of the



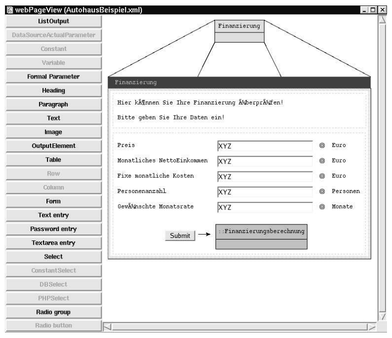
(a) Streets



(b) Electronic circuits



(c) Derivation tree assistant



Finanzierung

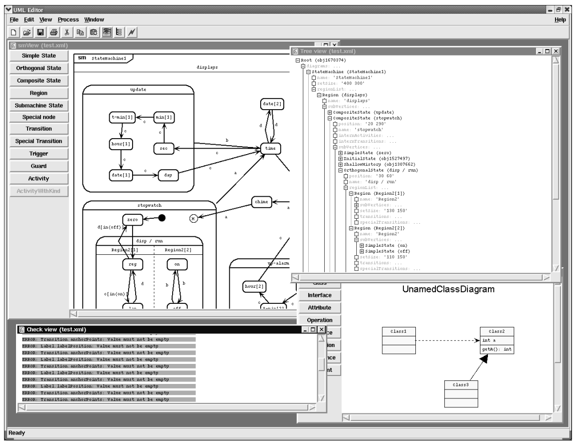
Hier kÃ¶nnen Sie Ihre Finanzierung Ã¼berprÃ¼fen!

Bitte geben Sie Ihre Daten ein!

Preis	XYZ	Euro
Monatliches NettoEinkommen	XYZ	Euro
Fixe monatliche Kosten	XYZ	Euro
Personenanzahl	XYZ	Personen
GewÃ¼nschte Monatsrate	XYZ	Monate

(d) PaderWAVE

Figure 2. Examples for different graphical styles



```
ERROR: Transition.anchorPoints: Value must not be empty
ERROR: Label.labelPosition: Value must not be empty
ERROR: Transition.anchorPoints: Value must not be empty
ERROR: Label.labelPosition: Value must not be empty
ERROR: Transition.anchorPoints: Value must not be empty
ERROR: Label.labelPosition: Value must not be empty
ERROR: Transition.anchorPoints: Value must not be empty
ERROR: Label.labelPosition: Value must not be empty
ERROR: Transition.anchorPoints: Value must not be empty
ERROR: Transition.anchorPoints: Value must not be empty
```

Figure 3. A simple UML editor used as running example

abstract structure: the editable structure and the semantic structure. The editable structure represents the information of the visual representation and the semantic structure represents the semantical information and serves as a basis for language processing.

3 The DEViL System

DEViL combines individual DSLs, frameworks and specification modules, that cooperate with each other to create implementations for graphical structure editors. The structure of the DEViL system is displayed in Figure 4. The basic component is the VL-Generator. It accepts specifications of the language structure, visual representations and code generation and generates a complete language implementation. It is built on top of tools for general language processing (Eli), graphical user interfaces (Tcl/Tk [14]), product derivation (Wodan) and graph layout (dot [8]). The topmost layer of DEViL contains specification modules and languages for specific aspects of the graphical representation. Visual patterns encapsulate implementations of common graphical representations. Generic drawings are a visual language to specify graphical details for certain pattern instances. SLTR is a simple specification language for textual parts in visual languages.

From the language developer's perspective, the tools, generators and libraries are closely integrated. A specification consists of three main aspects: the abstract structure, visual representations and the code generation (Figure 5). Because a language implementation may require multiple view types and multiple code processors, a DEViL specification can contain multiple corresponding specifications. They can be defined completely independent from each other and depend only on the syntax specification.

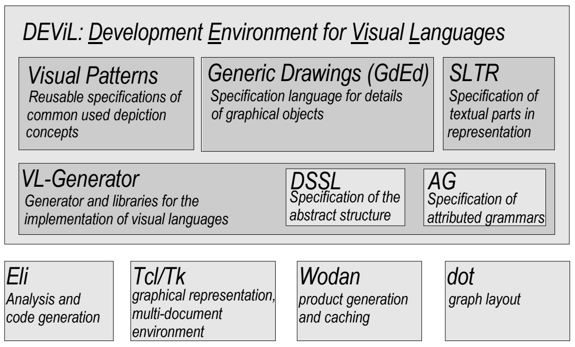

DEViL: Development Environment for Visual Languages
Visual Patterns
Reusable specifications of common used depiction concepts
Generic Drawings (GdEd)
Specification language for details of graphical objects
SLTR
Specification of textual parts in representation
VL-Generator
Generator and libraries for the implementation of visual languages
DSSL
Specification of the abstract structure
AG
Specification of attributed grammars
Eli
Analysis and code generation
Tcl/Tk
graphical representation, multi-document environment
Wodan
product generation and caching
dot
graph layout

Figure 4. Structure of the DEViL Environment

In the remainder of this section, we will discuss the three specification aspects shown in Figure 5 in more detail.

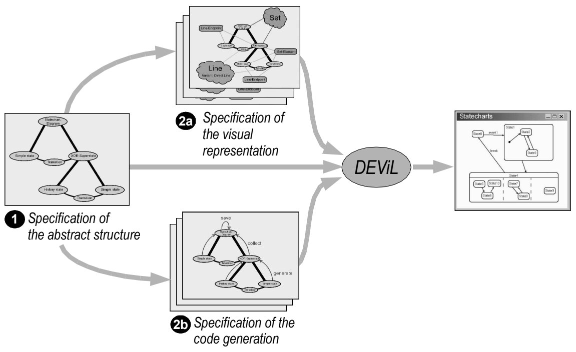

Set
Line
1 Specification of the abstract structure
2a Specification of the visual representation
DEViL
Statecharts
2b Specification of the code generation

Figure 5. Main specification aspects in DEViL

3.1 Structure Specification

In this subsection we show, how the abstract structure is defined. We further give an overview how the semantic and editable structure can be distinguished and how its coupling is defined.

The specification of abstract structures is based on a tailored specification language called DSSL (DEViL Structure Specification Language). Figure 6 shows an abstract syntax for statechart diagrams. The notation is very similar to object oriented programming languages like Java. It is based on the modeling concepts 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 6 has the attributes states and transitions which hold a set of subordinated language objects.

Figure 6 defines the structure of statecharts as follows: A Statechart has a list of *States* and a list of *Transitions*. States are either *SimpleStates* or *XORStates*. *XORStates* contain a

```
CLASS Root {
 diagrams: SUB Statechart*;
}
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 6. Structure specification of for Statechart Diagrams

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 point.

DEViL distinguishes between three types of attributes:

1. VAL-attributes (like *State.name*) store values and have an associated data type like VLInt or VLString.

2. SUB-attributes (like *Statechart.states*) store substructures that are instances of the specified class. The cardinality is specified behind the type. “*” means cardinality 0..*, “?” means cardinal 0..1 and “!” means cardinality 1.

3. REF-attributes store references to other language objects. They model cross relations in the abstract structure tree.

The keyword *INHERITS* specifies the subclass relationship. Just like in object oriented programming languages the subtype relation and the inheritance of attributes is expressed. DSSL supports multiple inheritance.

The handling of the abstract structure is one of the main tasks of a structure editor. To ease this task DEViL supports path expressions similar to XPath [21] that allow accessing an arbitrary part of the structure tree relative to a specific location. Amongst others path expressions can be used to define complex structural consistency checks.

To support the distinction between a semantic object and representations of that object, the abstract structure can be separated into editable and semantic structure (Figure 7). The editable structure represents the information that is needed to provide a visual representation. It stores user-defined layout decisions like positions or sizes of a language object as mentioned in Section 2.

In general, the relation between editable and semantic structure is quite complex. The editable structure can contain layout informations that are not present in the semantic structure. The other way round the editable structure may only represent fractions of the semantic structure. This happens if a view visualizes only a part of the semantic structure, which is very common for graphical structure editors. In case of the UML class diagram editor one view could show the package structure of a model and another one could display a detailed class diagram. Hence there is no functional dependence between editable and semantic structures. Both are built and maintained in parallel.

To define the correspondence between both structures, a tailored specification concept is used. According to it's design, similarity is considered as normal case and differences as exception. Therefore, in the very common case of closely related structures little specification overhead is required. The mechanism to couple the structures is asymmetric: From left to right changes are propagated in the same way, as changes of the visual representation are carried over to the editable structure. In the direction from right to left, structural constraints are checked and corrected if necessary. A violation of a constraint causes a change of the editable structure. A more detailed discussion about the specification concept and it's application range will be published separately.

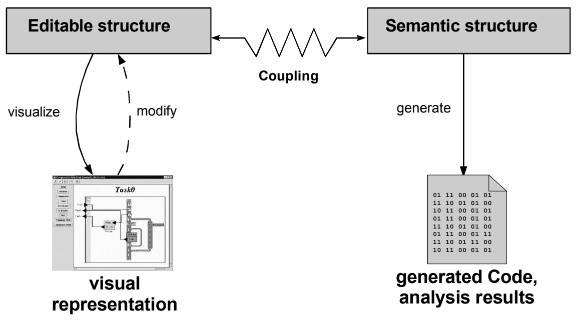


Figure 7. Separation of semantic and editable structure

3.2 Graphical Representation

Generated language implementations can provide multiple views on the editable structure. Each view type is declared by a tailored notation. Figure 8 declares the view type “smView”. Views of this type show a visual representation of trees with the root node *Statechart* which corresponds to the DSSL Syntax in Figure 6. They have the buttons *SimpleState*, *XORState* and *Transition* to insert the corresponding language elements.

The specification of the graphical representation is based on attribute grammars. The underlying tree grammar of the attribute grammar is called layout structure. The grammar as well as the structural mapping from the editable structure is automatically generated. Both can be adjusted by additional specifications which is useful for some visual representations where additional tree contexts are needed in order to specify a visual representation.

Attribute computations are associated to these tree contexts. They compute layout attributes like positions and paint graphical primitives into a canvas. Figure 9 shows the computation of a rectangular frame as part of a statechart object.

```
VIEW smView ROOT Statechart {
  BUTTON "Simple State" INSERTS SimpleState;
  BUTTON "XOR State"    INSERTS XORState;
  BUTTON "Transition"   INSERTS Transition;
}
```

Figure 8. Declaration of a statechart view

Beside the graphical representation, additional invisible context informations are added so that the language user can interact with the graphical representation. For instance insertion points are added to highlight positions in which new language objects can be inserted. When a new language construct is inserted the editable structure is modified and an update of the view is invoked which forces the attribute evaluator to recompute the graphical representation (model-view paradigm).

```
SYMBOL Statechart COMPUTE
 SYNT.pos1 = SELECT(THIS.region,getPosNW());
 SYNT.pos2 = SELECT(THIS.region,getPosSE());
 SYNT.rectangle =
       vlCreateObject(THIS.objId, "rectangle",
            VLPointArray(THIS.pos1, THIS.pos2),
                    INCLUDING Root.lineOptions);
END;
```

Figure 9. Attribute computations

To simplify the specification of graphical views, there are higher level libraries and tools available, that are built on top of the VL-generator (see Figure 4). 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 representation structure. 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.

```
SYMBOL Statechart INHERITS VPRootElement,
  VPForm, VPConnectionArea
COMPUTE
  SYNT.drawing = ADDROF(StatechartDrawing);
END;

SYMBOL Statechart_transitions
  INHERITS VPConnectionList END;

SYMBOL Statechart_name INHERITS
 VPFormElement, VPIdTextPrimitive
COMPUTE
  SYNT.formElementName = "name";
END;

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

SYMBOL State INHERITS
   VPSetElement, VPConnectionEndpoint
END;

SYMBOL SimpleState INHERITS VPForm
COMPUTE
  SYNT.drawing = ADDROF(SimpleStateDrawing);
END;

SYMBOL SimpleState_name
  INHERITS VPFormElement, VPIdTextPrimitive
COMPUTE
  SYNT.formElementName = "body";
END;

SYMBOL XORState INHERITS VPForm
COMPUTE
  SYNT.drawing = ADDROF(XORStateDrawing);
END;

SYMBOL XORState_name NHERITS VPFormElement,
  VPIdTextPrimitive
COMPUTE
  SYNT.formElementName = "name";
END;

SYMBOL XORState_subStates
  INHERITS VPFormElement, VPSet
COMPUTE
  SYNT.formElementName = "subStates";
END;

SYMBOL Transition INHERITS
   VPPolyConnection
COMPUTE
  SYNT.lineWidth = 2;
END;

SYMBOL Transition_from INHERITS
   VPConnectionFrom END;

SYMBOL Transition_to INHERITS
   VPConnectionTo END;
```

Figure 10. Specification of a statechart view

In figure 10, the role *VPSet* is associated to the symbol *Statechart_states* and the role *VPSetElement* is associated to the class *State*, which means that this structure node can be positioned at any position inside of a set.

Figure 10 contains a complete attribute grammar which computes the concrete representation of the view type "*smView*" (see Figure 3). By assigning roles like "*VP-RootElement*", "*VPForm*" or "*VPTextPrimitive*" to certain grammar symbols the whole depiction is specified. The definition of these roles are formulated by attribute computations like that shown in Figure 9.

Inherited attribute computations can be overriden or existing computations can be extended to influence details of the graphical representation. For instance in the context of *Transition* the inherited *lineWidth* attribute of the *VPPoly-Connection* role is overridden. The meaning of roles can easily be understood. *Statechart* inherits from *VPRootElement*, because this symbol is the root of the depiction. Furthermore it inherits from *VPForm*, because the depiction corresponds to the form pattern.

The appearance of this form is specified in another specification language. The so called generic drawing is shown in Figure 11. The subelements *Statechart_name* resp. *Statechart_states* inherit the *VPFormElement* role to denote that it belongs to the above formular. By the use of the control attribute *formElementName* the correct container of the generic drawing is chosen. The role *VPConnectionArea* at the symbol *Statechart* is needed to express that connection lines can be inserted inside the *Statechart*.

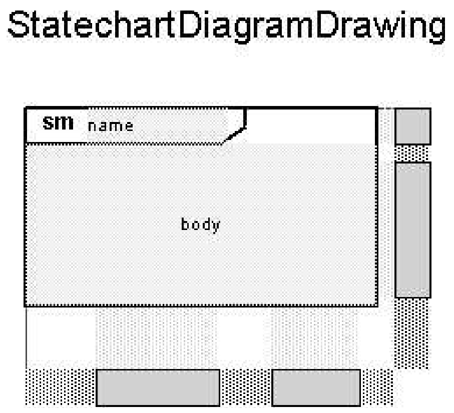


Figure 11. Generic drawing for a Statechart Diagram

Let's have a closer look on the specification language for generic drawings. It allows to define the representation for language objects presented by the form pattern. Figure 11 shows the generic drawing for a Statechart Diagram. It consists of several line, text and rectangle primitives and two containers in which sub elements can be nested. The decorating primitives like lines, rectangles, images or text can be specified in a WYSIWYG style. Containers act as placeholders for other language objects, in this case for the *name* and the content (*body*) of the statechart.

The so-called stretch intervals at the bottom and the right side of the diagram define how the representation can be transformed to fit into the layout of the complete reprenseta-tion. If the size of a subelement exceeds the actual container size, the area covered by one or more stretch intervals is linearly stretched to enlarge the corresponding container. All details of this layout process are encapsulated by a built-in size optimization strategy.

With the library of visual patterns and user-defined generic drawings a wide spectrum of visual languages can be specified. Nested visual languages like Nassi-Shneiderman diagrams, matrix based languages that are used in mathematics, languages with sets and connections like UML diagrams can be modeled as well as graph structures and many more. 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. It is not restricted to a HTML like representation but other graphical concepts like trees, graphs or connections can also be displayed. Additionally, the patterns encapsulate mechanisms to edit the representation. Due to the fact that the implementation and the application of visual patterns are specified on the level of attribute grammars, the concrete representation can easily be adapted. A more detailed discussion of attribute grammars and visual patterns in the context of view specification can be found in [17].

3.3 Code generation

Code generators are specified by attribute grammars, too. In this case the attributes store code fragments and analysis results. The Eli subsystem [12] provied a lot of useful libraries and tools for this processing phase.

Often, a language implementation generates source code of a textual programming or specification language, e. g. C-Code or SQL statements. In DEViL the tool IPTG is used to implement the generation of source code.

IPTG is a variant of PTG (pattern based text generator), that is part of the Eli system. It can be used standalone in ad-hoc programs or in interaction with other generators and evaluators to produce intermediate code. In DEViL IPTG interacts with the LIDO attribute evaluator that is responsible for semantic analysis and code generation.

The specification style of IPTG is comparable to the *printf* statement in C in which parameter substitution is applied. Nevertheless PTG reaches the output efficiency of a stream based implementation.

Figure 12 shows an example for an IPTG text pattern.

```
WhileStmt(condition, body):
  while ([condition]) {
    [body]
  }
```

Figure 12. A simple IPTG pattern

The user defines text that is interspersed with placeholders in which the values of the parameters are substituted. The parameter values can be primitive values or results of other PTG functions. IPTG automatically generates C functions out of the provided definitions which can be called for example in LIDO (see figure 13).

```
PTGWhileStmt(THIS.pers_condition,
 CONSTITUENTS StmtList.code WITH
  (PTGNode, PTGNewLine, IDENTICAL, PTGNull));
```

Figure 13. Application of a generated PTG function in the attribute grammar

IPTG supports pretty printing to beautify the target code. The indentations of the lines in figure 12 specify the indentation of the generated code. More informations about PTG can be found in [2] and a description of IPTG is available at [16].

4 Selected design and realisation concepts

In this Section, we discuss some interesting technical and conceptional design aspects. None of these aspects are limited to the domain of visual languages, but they are useful in any domain.

4.1 Automated product derivation

The presented application domain is rather challenging and the generation task is far too complex to be solved by a single generator. To modularize the generation process, DEViL consists of about 16 individual generators. Each focuses on a very specific, manageable subproblem. The most important generators are:

- *modelGen*: Generates data structures from DSSL specifications. These data structures are used to store the internal program representation. The generated code and the associated framework additionally supports saving/loading of structures in XML format, the evaluation of path expressions, the registration of event listeners and a multi-level undo mechanism. The processor also generates a syntax documentation in docbook format [19].

- *viewGen*: Generates the core implementations for visual views. The generator and the associated framework implements basic properties of the the view window like the associated toolbar buttons. Further, the processor generates a grammar for the representation structure as well as mapping functions, that build the representation structure from the editable structure.

- *Liga* [9]: This attribute evaluator generator yields tree walk implementations from attribute grammar specifications. It is part of the Eli system. Attribute grammars are used in DEViL to specify visual views as well as code generators.

- *Generic drawings*: This language processor generates implementations for graphical objects based on the form pattern. The implementations support dynamic adjustment of the objects according to its graphical context (see Section 3.2).

- *docbook2html* and *docbook2pdf*: This family of processors transform docbook documents into specific target formats.

- *PTG* [2]: Generates constructor functions for text fragments. The generated code and associated framework provides a very convenient and efficient way to construct structured target text.

- *IPTG*: Implements an alternative syntax for PTG patterns. In this notation, the indentation of generated texts is specified by the indentation of the corresponding patterns.

- *cc*: Compiles and links the generated as well as the hand-written C/C++ files into binary code.

- *installerGenerator*: Creates self-extracting distributions of the language implementation, that can be installed elsewhere.

- *FunnelWeb* [1]: This literate programming system allows the construction of specifications containing both documentation and code fragments. The processor can extract the specifications and produce documentation files in different formats, including html and texinfo.

The implementation of generic drawings has been generated by DEViL itself. Most of these textual processors (*modelGen*, *viewGen*, *Liga*, *PTG*, and *IPTG*) have been generated by the Eli system. Other tools (*cc*, *docbook2html*, *docbook2pdf* and the *installerGenerator*) are external tools, that are maintained elsewhere. Beside of the above mentioned generators, DEViL and Eli contain a number of “smaller” generators, that process configuration files, simplify the specification of textual sub-representations (SLTR) or serve special purposes in the Eli system (PDL etc.).

The coordination of the generators is a challenging task: They must be executed in the right order with the right inputs and intermediate results must be organized and cached. Caching is important, because the build process would otherwise require an unacceptable amount of time. The caching mechanism must further be able to manage different variants of the products (e. g. code with or without debug informations) at the same time. Last but not least, generator errors must be processed in a user-friendly way. For example, some specification languages allow the integration of c-code fragments. Errors in such a fragment are found by the c-compiler, but they should be reported in the context of the specification file, that was originally written by the language developer. To summarize all these requirements, the system user should have the impression of one big generator. The intermediate generation steps should be completely hidden from the user.

To implement all these requirements, DEViL uses the tool control system Wodan. Its features are comparable to those of the Odin system [4], which is used as tool control system in Eli. Wodan is a new development to circumvent some technical shortcomings of Odin. In the remainder of this subsection we will describe the use of Wodan in the DEViL System.

The product dependences of a concrete project can be considered as derivation graph. The high-level structure of DEViL’s derivation graph is shown in figure 14. Nodes model processors and incident edges model inputs of the corresponding processor. Some of the graph nodes model atomic processors, but more often nodes contain a subordinated derivation graph. For example, the node “binVlEnv” can be interpreted as a single generator, whereas the nested derivation graph reveals its internal processing structure. Note, that none of the nodes shown in Figure 14 is an atomic processor at this level, but all have a substructure, that is realized using the Wodan system, too.

From another perspective, the graph nodes represent derivable products, that can be requested by the user or that can be used as input for other processors. For example, the DEViL user can request the product “modelDocPdf”. To derive this product, Wodan executes all paths from the input to the requested node. Note, that most of the nodes represent sets of files. Often, the processors extract a certain subset of the input files and generate new files from these inputs. In general, the output of a processor is composed by generated files, library files, as well as input files, that are simply passed through.

“drawingsGen” is a typical generator node. It processes files of certain types in the input (in this case files containing generic drawings), generates new files from them (in this case c-implementations of the drawings) and inserts them together with some library files into the pipe. An other kind of node is “visualPatterns”. It doesn’t contain a generator

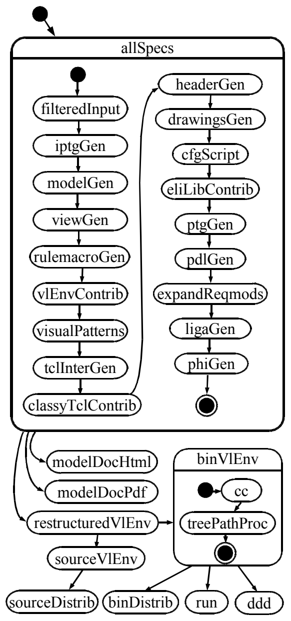


Figure 14. The derivation graph

but just contributes some library files (in this case formulated in terms of attribute computations). They and other attribute grammar fragments will then be processed later on in the “ligaGen” processing step.

At first glance the derivation graph seems to be static, but in fact the nested derivation graphs partially depend on the concrete input. As mentioned before, most of the nodes denote sets of files. Let’s consider the “cc” derivation step inside of the “binVlEnv” node: It extracts a set of c- an h- files, compiles them (if necessary) into o-files and links them into a single binary file. So on a lower level there are nodes representing individual c- and h-files and even the dependences between them (caused by include-directives) depend on the input.

Odin and Wodan have different mechanisms to specify construction plans for derivation graphs. Odin uses derivation rules, that are specified in a special language. In Wodan, the construction plan for derivation graphs looks like a functional program. Functions describe, how a processor is executed (in the case of an atomic processor) or how a subordinated derivation graph is established, that realizes the desired processing. A function execution corresponds to an instantiation of a concrete derivation graph node and the parameters of the function determine the inputs of the corresponding processing step. The functions does not operate on real files. Instead, they operate on virtual files in a virtual file space hidden from the system user. The organization of the virtual files, the distinction of product variants and the caching is done transparently by Wodan. The individual processors are only executed, if the cached products are not up-to-date.

The construction of an input-specific derivation graph, the error tracing and the support of product variants is quite challenging. Elaborated versions of the unix tool *make* have some support to incorporate automatic dependence analysis, but they can’t handle product variants in a scalable way and there is no built-in support for error tracing. Additionally, our approach is much more modular and scalable.

4.2 General strategies for system design

The design of the DEViL system (as well as the design of Eli) is based on some general principles, that are usable in other application domains, too. To achieve both, a powerful reuse mechanism as well as high flexibility, the following techniques have been successfully applied in both systems:

- multiple specification levels,
- optional concepts and
- cooperating specification languages, each of which focusing on a very specific subproblem

The concept of multiple specification levels helps to reach a high specification level without limiting the flexibility of the approach. Visual patterns are an instance of this approach. In general, the user applies and parametrizes visual patterns to specify a graphical representation as described in Section 3.2. If this level is not sufficient for a specific instance, the user can switch to a lower specification level and write attribute computations by hand. If for some reasons even hand-coded attribute computations are not sufficient, the user can implement auxiliary C or C++ functions. In this case, the lowest specification level has been reached.

Optional concepts of DEViL include the distinction between semantic structure, editable structure and layout structure, the definition and generation of documentation, as well as the application of high-level languages like generic drawings or SLTR. None of these concepts or corresponding specification languages has to be considered, if the current project does not require them. Due to this principle, implementations of small languages can be generated from specifications, that contain only few hundred lines of code [16].

The last key concept, cooperating generators that focus on specific subtasks, is very advantageous to promote a modular system design. From the DEViL developer’s point of view, it is very easy to restructure or to extend the system in oder to enhance it or address a new problem domain. In our case we reused and restructured the Eli system to address the domain of visual languages. From the tool set user’s point of view, the well-defined interfaces between the generators offer a great flexibility, because one can easily replace some generated parts by hand-coded implementations, whereas other generators can still be used.

5 Related work

Many generators and systems have been developed to aid implementation of visual languages. They can be distinguished into two groups: systems for implementation of structure editors and systems for language implementations, that apply visual parsing techniques. We will not go into detail of the second group, even though it is very interesting, too.

Some well known systems for implementation of visual structure editors are PROGRES [15] and GenGEd [3]. In both approaches, the abstract structure is modeled as a graph. In contrast to these systems, the distinction between aggregated and referenced structures (SUB vs. REF attributes) play an important role for structure modeling and view definition in DEViL. The GIGAS System [7] has inspired the transition from editable structure to representation structure. In this system the representation structure is called layout structure and can be derived from the editable structure by a comparable specification mechanism.

The Eclipse Graphical Modeling Framework (GMF) has a number of strong similarites to our approach. It is based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). EMF is used for structure specification and is roughly comparable to DSSL in DEViL. GEF offers a library of common graphical constructs, whose counterpart in DEViL are the visual patterns. GMF distinguishes between semantic structure (domain model) and editable structure (graphical definition model) like DEViL does. The fundamental difference is, that DEViL introduces more specialized languages, to make the integration of different specification aspects as seamless as possible. For example, the coupling concept between the distinct structures causes only little overhead due to a tailored mapping language. To specify the graphical representation, we use attribute grammars as basic calculus. In this way, the encapsulated graphical concepts can be applied, combined and adapted much easier than on source code level.

Another variant of systems for construction of visual structure editors are meta modeling tools like MetaEdit+ [5]. They provide similar concepts to model the language structure, but they have limited flexibility in respect to the graphical representation. Often meta modeling tools provide a graphical user interface for language definition, but often they do not have a multi-level specification mechanism as described in Section 4.2.

We further stressed the applied design concepts and mechanisms on a general level. A related, more detailed discussion about reuse in domain specific environments in the context of the Eli system can be found in [10]. We further expressed our experiences with elaborated build tools like Wodan and Odin and emphasized their necessity. Even though the underlying concepts are not new, popular systems like Gnu Make or Apache Ant [6] don't recognize them.

6 Conclusion

In this paper we described the DEViL system with special focus on the underlying generator- and reuse principles. The generator itself is based on an extensive domain study, which is an essential prerequisite for a successful domain specific tool set. New abstractions and concepts comprise the distinction of semantic, editable and layout structures, as well as specialized languages for specific kinds of representations (generic drawings and SLTR). The combination of these concepts allows for generation of even challenging visual languages. The system design discussed in Section 4.2 enables both, a high level specification approach allowing very concise specifications, as well as high flexibility.

An important realization prerequisite is a method to coordinate the individual generators. We discussed the basic concepts of the applied build tools. The most important aspect of these tools is the input-specific construction of a derivation graph. The knowledge about the generator dependences can be completely encapsulated and the internal derivation structure is therefore completely transparent for the tool set user. From the user's perspective, DEViL looks like a single big generator, that accepts a file set as input and generates the requested product, say a self-extracting distribution of the generated language implementation.

References

[1] *Eli Online Documentation: FunnelWeb User's Manual*, 2005. `http://ag-kastens.uni-paderborn.de/elionline/fw_toc.html`.

[2] *Eli Online Documentation: Pattern-based Text Generator*, 2005. `http://www.upb.de/cs/ag-kastens/elionline/ptg_toc.html`.

[3] R. Bardohl. GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In *1998 IEEE Symp. on Visual Lang.*, pages 48–55, Sept. 1998.

[4] G. M. Clemm. The odin specification language. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 144–158, January 1988.

[5] M. Consulting. *MetaEdit+ User's Guide*, 2002. `http://www.metacase.com/fs.asp?vasen=vasen.html&paa=products.html`.

[6] A. S. Foundation. Apache Ant Homepage, 2006. `http://ant.apache.org`.

[7] P. Franchi-Zannettacci. Attribute specifications for graphical interface generation. In G. X. Ritter, editor, *Inform. Proc. '89*, pages 149–155. North-Holland, 1989.

[8] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[9] U. Kastens. LIGA: A language independent generator for attribute evaluators. Technischer Bericht, Reihe Informatik tr-ri-89-63, Universität Paderborn Fachbereich Mathematik-Informatik, 1989.

[10] U. Kastens. Reuse methods in a domain specific software environment. Technischer Bericht, Reihe Informatik tr-ri-02-232, Universität Paderborn Fachbereich Mathematik-Informatik, 2002.

[11] U. Kastens and M. Jung. Streets Abschlußbericht. Technical report, Universität Paderborn, 1998. `http://www.upb.de/cs/ag-kastens/paper/streets.ps.gz`.

[12] U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. 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, Mar. 1998.

[13] U. Kastens and C. Schmidt. VL-Eli: A generator for visual languages. In *Proceedings of Second Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, number 2027 in Electronic Notes in Theoretical Computer Science, Grenoble, France, 2002. Band 65, Elsevier Science Publishers.

[14] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[15] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *1996 IEEE Symp. on Visual Lang.*, pages 148–155. IEEE Comp. Soc. Press, 1996.

[16] C. Schmidt. DEViL Homepage, 2006. `http://ag-kastens.uni-paderborn.de/forschung/devil/index_en.php`.

[17] C. Schmidt and U. Kastens. Implementation of visual languages using pattern-based specifications. *Software - Practice and Experience*, 33:1471–1505, Dec. 2003.

[18] C. Schmidt, P. Pfahler, U. Kastens, and C. 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.

[19] N. Walsh. *DocBook: The Definitive Guide*. O'Reilly and Associates, Inc., 2003.

[20] S. Winter. Generierung von dynamischen Web-Anwendungen aus visuellen Spezifikationen. In *Fachwissenschaftlicher Informatikkongress - Informatiktage 2005*, Schloss Birlinghoven - Sankt Augustin, Apr. 2005.

[21] XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. `http://www.w3c.org/TR/xpath`.