SIMtelligence Designer/J: A Visual Language to Specify SIM Toolkit Applications

Carsten Schmidt[1] Peter Pfahler[1] Uwe Kastens[1] Carsten Fischer[2]

{cschmidt,peter,uwe}@uni-paderborn.de
cfischer@orga.com

[1]University of Paderborn
Department of Mathematics
and Computer Science
Fürstenallee 11
33102 Paderborn
Germany

[2]ORGA Kartensysteme GmbH
Am Hoppenhof 33
33104 Paderborn
Germany

Abstract

The SIM Application Toolkit is a standardized interface that provides mechanisms allowing applications, existing in the SIM, to interact and operate with any mobile equipment which supports the specific mechanisms required by the application. In case of Java Cards such software is implemented in Java. To simplify software development in this rather complex application domain, we developed a visual domain specific language. *SIMtelligence Designer/J* is used to create programs in that language and to translate them into Java. It is generated by the VL-Eli system, a tool for the implementation of visual languages.

This paper gives a short introduction to the application area, presents the *SIMtelligence Designer/J* tool and discusses the VL-Eli system.

1 Introduction

Domain specific languages (DSLs) are developed for specific application domains that tend to be rather narrow. The language constructs are usually chosen from the particular concepts of that domain and their notation adapts the way how domain experts express their designs. In this sense a DSL is much closer to the solution of application tasks than general purpose programming languages are, and they allow designs on a higher level of abstraction.

This strategy of DSLs is especially supported by visual languages: Domain experts often prefer graphical rather than textual descriptions: high level constructs and complex relations between them are usually better captured by iconic representations, graphic connections, two-dimensional layout; visual notations are easier to learn than textual ones.

In this paper we describe the development of a language for the design of Javacard based SIM Toolkit applications (STK applications), e.g. additional software on SIM cards in mobile phones. This domain is very well qualified for being supported by a visual DSL: It is a rather narrow domain; the programs have a common structure and are composed from a small set of high level constructs according to a few design paradigms (menu selection, event driven reaction). The large distance from the DSL constructs to their implementation in the target language Java is bridged by the translation of the DSL. The intended users of the DSL are experts in the STK domain and are not expected to be familiar with programming in textual languages like Java.

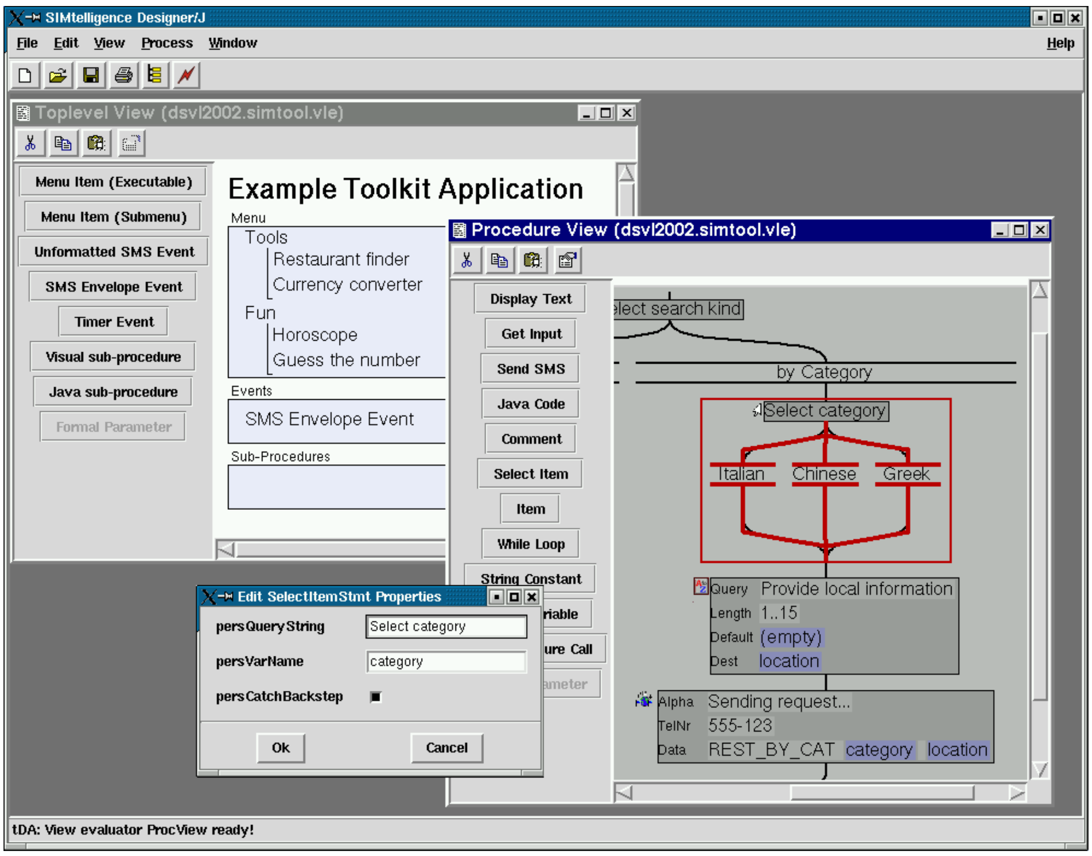
The design and implementation of visual DSLs requires a wide range of know-how: The designer has to understand the application domain and needs techniques for drawing, layout, and manipulation of graphical objects, in addition to most of the analysis and translation tasks needed for textual languages. As the market for a DSL is usually small, such a large design and implementation task should be supported by tools to balance the invested efforts.

We used our VL-Eli system [11, 12, 14] to implement our DSL for STK applications. VL-Eli is a result of our recent research and development in visual languages. Its central paradigm is incorporated in a set of patterns which encapsulate precoined adaptable and combinable solutions for graphical language constructs. The language designer selects patterns from that set and associates them with elements of the abstract syntax.

In VL-Eli conventional tasks of language implementation are supported by the huge set of dedicated tools in the well-established Eli system [13].

The design and implementation of the STK DSL is performed as part of larger cooperation between the University of Paderborn and the company Orga Kartensysteme GmbH. A rather elaborated version of the language has been developed. A structure editor with a prototype translator is generated using VL-Eli. This state has been reached with the investment of only few person months. In the next step the language system will be evaluated and completed.

In the following sections of this paper we first introduce the domain of STK applications. Section 3 gives an overview over the language and presents a selection of its constructs. Central aspects of the implementation using VL-Eli are presented in Section 4.



persQueryString	Select category
persVarName	category
persCatchBackstep	■

tDA: View evaluator ProcView ready!

Figure 1: The Generated Language Environment

2 Application Domain

The owner of a mobile device, such as a cell phone, is identified by a smart card called the subscriber identity module (SIM). It enables the access to a network and can store user specific data such as telephone numbers. Telephone companies and handset manufacturers have agreed upon the extension of the mobile equipment (ME) functionality in a manufacturer and provider independent fashion by allowing to add applications to the SIM card. They defined the SIM Application Toolkit (STK) on top of the standard SIM technology. The STK specifies the interface between the ME and the SIM. It uses Short Message Service (SMS) to transfer information between the handset and the service provider. Here are some examples of typical STK applications: For a horoscope service the user is asked for his zodiac sign. This information is sent to a service center. The service center returns the appropriate horoscope text via SMS. An e-mail application asks the user to enter an e-mail address and a message. The application sends this data to the providers gateway where the e-mail is generated and sent. Or, as a last example, the user may be offered a list of restaurants in his immediate vicinity. This list is received from the service provider, after the restaurant application on the SIM card has transmitted the users current position.

The SIM Application Toolkit has been a great success and has been standardized as part of the GSM standard [1]. It defines how the application program can register menu elements and listen to events such as timer or SMS events. When an event occurs, a procedure on the card is executed. The procedure can invoke other functions of the ME, for example it can display a message, ask for input or dial a telephone number.

In the last years it has become popular to use Java cards [4] as SIMs. Java cards include a virtual Java machine which can execute byte code instructions for a subset of the Java language. Thus Java cards can be programed in the high level object-oriented language Java. There is also a standardized Java package implementing the STK functionality.

Although Java is a high level language it is not very convenient to implement a SIM Toolkit application in Java. There are several reasons for this:

To implement Java card applications, the allowed language constructs are restricted: only a certain part of the Standard API is available and some data types, e.g. `String` and `int`, are not supported. Byte arrays and short-values have to be used instead. Furthermore, the size of the available memory is quite small on Java cards and garbage collection is not supported by all Java cards. So the memory consumption of programs has to be strictly limited. All these restrictions lead to complex, sometimes “ugly”, and difficult to write (and read) Java programs.

There is another reason, why Java is not ideal for SIM Toolkit applications: The user should be able to jump back to a previous menu action at any point in time. For example, in an e-mail application the user is first asked for the e-mail address and then for the message text. At this point, the user may want to jump back to the first question. In general this feature requires goto statements and a rollback functionality to undo state changes, which is not supported directly by Java. A Java approximation consisting of labeled nested loops is quite complex and error-prone to be implemented manually.

We decided to implement a visual structure editor for the construction of STK applications. Its name is *SIMtelligence Designer/J*. The underlying domain specific language (DSL) is designed for non-programmers to be able to construct simple SIM Toolkit programs. We implemented a **structure editor** because users who are not familar with the new language need guidance in the construction process. SIM Toolkit commands have a large set of options and parameters. Showing them in an interactive dialog is a great help for the user. We designed a **visual language** because visual expressions are more intuitive and can improve the perception of specifications. Additionally, users tend to be more motivated to learn and use a visual language. The visual program is translated to a Java Card STK application by the code generation phase of *SIMtelligence Designer/J*.

3 Design of the Visual Language

There are two main concepts in SIM Toolkit applications. (1) At the beginning, all relevant events and the menu structure are registered by the application. (2) Each event triggers a procedure, which has a characteristic control flow with interactive SIM-Toolkit commands and backward move functionality. The separate procedures are independent from each other. The whole program is obtained by associating them to events. Thus, the language is decomposed in two types of views: (1) The top-level view specifies the relevant events and associates each with the appropriate procedure object. (2) The Procedure view shows a procedure in full detail. In the following, we will describe the structure of these views separately.

3.1 The Top-level view

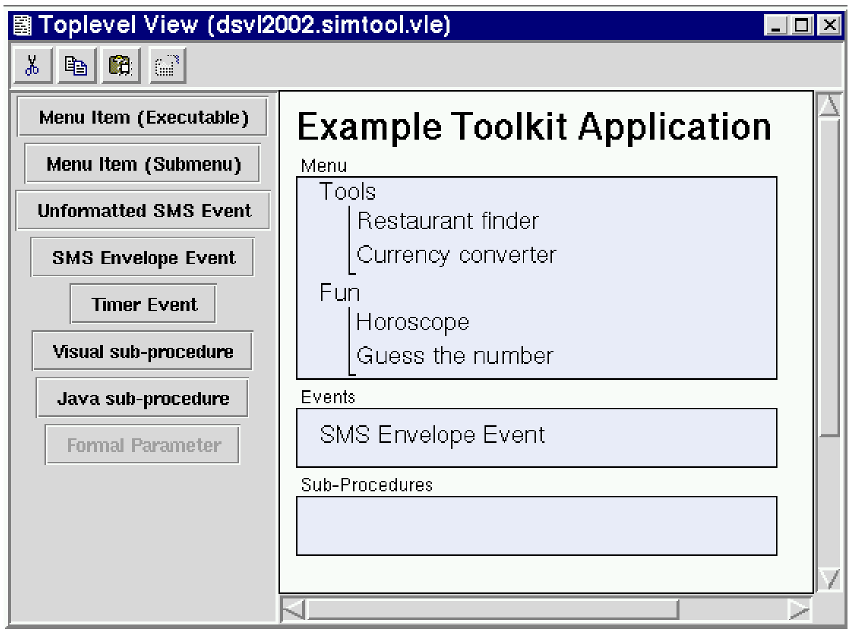


Figure 2: The top-level view

The top-level view (Figure 2) specifies the events, which the application is interested in. At the same time it defines all procedures of the application. (The procedure view can be opened via context-menues of the individual objects in the top-level view.)

The top-level view consists of three parts, which are enclosed by blue boxes:

- The topmost box shows the menu structure of the application. This structure is registered, when the application is installed. It consists of the language elements “Menu item (Executable)” and “Menu item (Submenu)”. When the user of the generated program selects a menu item, that was specified by a “Menu item (Submenu)” construct, the corresponding sub-menu is shown. When he selects a menu item specified by “Menu item (Executable)”, the corresponding procedure is executed.

- The box in the middle specifies additional event handlers. Currently, language elements for timer events and SMS events are available. Each element type has its own properties to customize the event conditions and can trigger a procedure.

- The box at the bottom defines additional procedures. They are not bound to external events but can be called by other procedures. There are two kinds of sub-procedures: The contents of “visual sub-procedures” is specified in the same way as other procedures by Procedure views. “Java sub-procedures” are defined in plain Java. In this way experienced users can achieve any functionality that can’t be specified visually.

3.2 The Procedure view

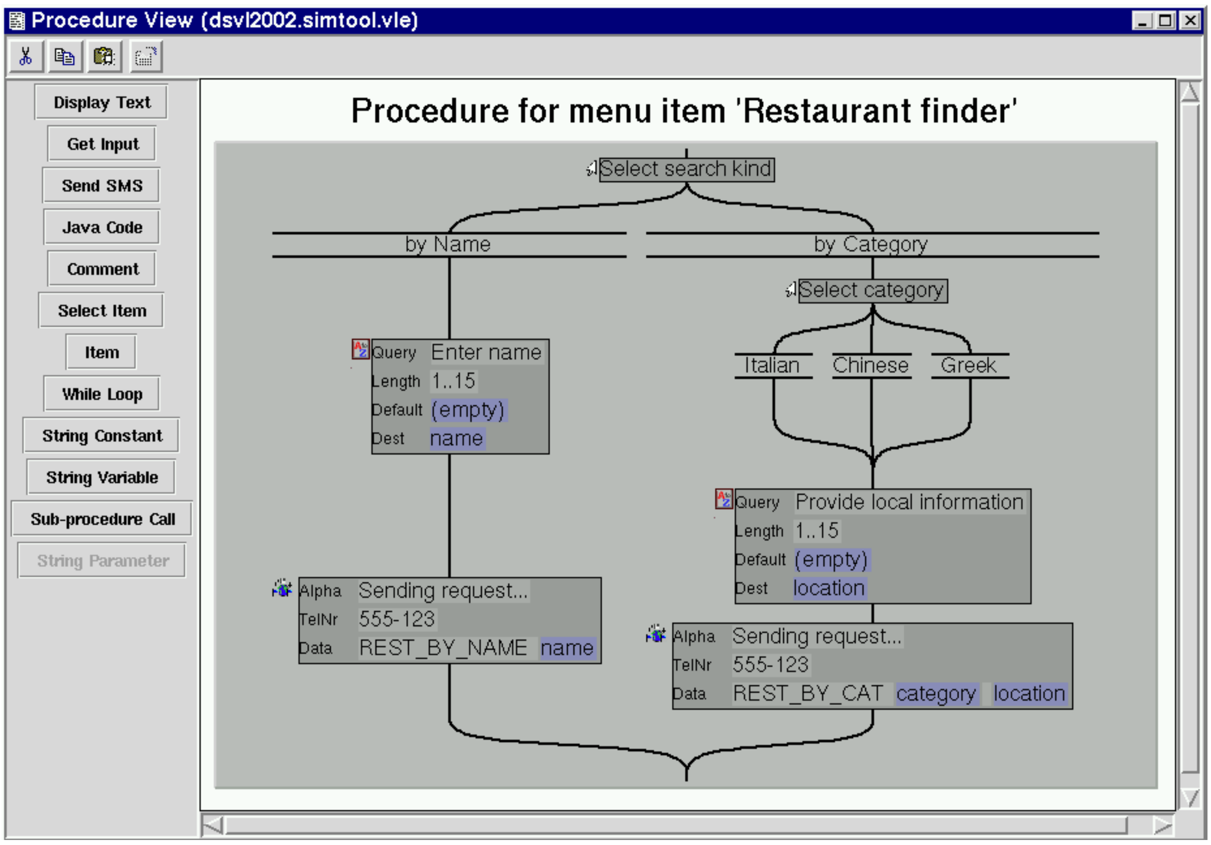
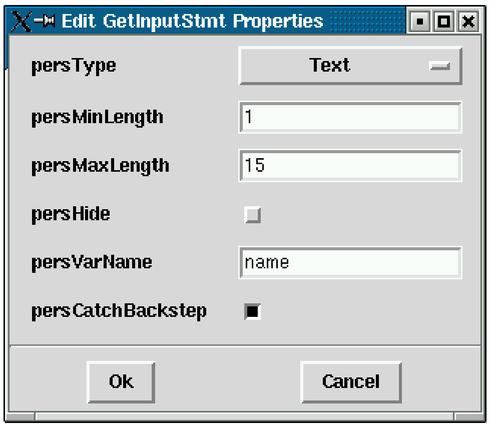


Figure 3: The “Restaurant finder” procedure

The Procedure view (Figure 3) shows the control flow of procedures. The nodes of the control flow graph are SIM Toolkit function calls or sub-procedure calls. They are created by the toolbar buttons on the left side of the window. The type of the nodes is shown by a small icon on the left side. Since the editor is still a prototype, we only support the most common toolkit functions, which are “display text”, “get input”, “send sms” and “select item”. This set will be extended, when the language is fully evaluated.

In contrast to Java, we don’t support general control flow structures but only a specialized subset. The subset is designed to make the specification of common SIM Toolkit applications as easy as possible. For example, the “select item” command and the switch over the function result is combined in a single statement.

Figure 3 shows the specification of the “restaurant finder” procedure. The user is first asked whether he wants to search by name or by category. If he selects “by category”, he is asked for the kind of desired restaurant (Italian, Chinese or Greek). The result is stored in the variable “category”. After that, the SIM Toolkit command “provide local information” is called, which stores the current position (derived from the active network cell) into the variable “location”. These informations are sent to the service provider, that returns the address of the nearest restaurant of the selected category via SMS. If the user selects the branch “by name”, a “get input” statement asks for the name of the restaurant, which is then forwarded to the service provider.



persType	Text
persMinLength	1
persMaxLength	15
persHide	☐
persVarName	name
persCatchBackstep	■

Figure 4: A property dialog for a “get input” object

The properties of the “get input” statement can be configured in a dialog as shown in Figure 4, which is accessible via context menu. The statement requests a text string (the alternative would be a number) with a length between 1 and 15 characters. The input should be visible on the display (hiding it would be reasonable if it were a password). The final value is stored in the variable “name”. As you can see, the representation of the “get input” statement gives an overview about this configuration, too. Additionally it contains a “query” string (“Enter name”) and a “default” string (“(empty)”). These values are not defined in the dialog, because strings are individual objects. They can be placed into the statements visually (Figure 3). There are two types of string objects: String constants (with gray background) and string variables (with blue background). They can be concatenated to build up more complex values.

One explanation is missing up to now: The “catch backstep” option. As mentioned before, when the user initiates a backward move, the execution should jump back to the preceding interactive statement. For example, if a backward move is initiated at the left “send sms” statement, the program should jump back to the “enter name” request. In some cases, the application designer wants to be able to influence the target of a backward move, which is possible by disabling the “catch backstep” option. When a backward move occurs, the program jumps to the last executed statement with *enabled* “catch backstep” option.

3.3 The Development Process

The design and implementation was mainly done by a single person, who had no previous domain knowledge. The first step was to investigate the application area. The requirements were discussed with the domain experts. Further, the language developer analyzed material like STK application descriptions and hand-written implementations as well as the predecessor "SIMtelligence designer". This phase took one month.

In the next step, the language developer designed the core language. The structure and the visual representation were designed at the same time and discussed with the domain experts.

After both sides agreed on the core language, the language designer generated a prototype editor for the language and presented it to the domain experts. This step took another month.

In the last step (so far) the language developer implemented a simple code generation. Simple means, that we have not yet focused on code optimization issues. Some new language constructs were introduced and the result was given to the domain experts to evaluate it. This took about 1.5 months.

4 Implementation

We used the VL-Eli system [11, 12, 14] to implement the visual environment. VL-Eli generates visual structure editors together with a multiple window environment from high level specifications. A visual language is specified by identifying certain patterns in the language structure and selecting a visual representation from a set of precoined solutions. For example, statements in procedures are arranged according to the *List* pattern: They are an ordered sequence of elements, which are placed along a certain axis in a two-dimensional region. The language developer selects a precoined specification module for *List* representations and parameterizes the details of the representation by a variety of options. Visual programs are represented by attributed abstract trees. Therefore, further phases of processing can be generated by state-of-the-art tools for language implementation [13].

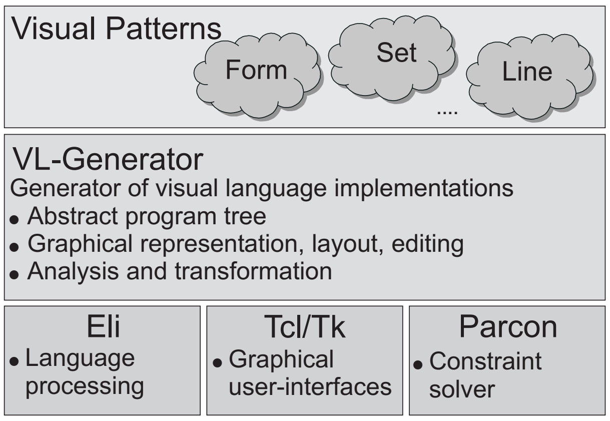

Visual Patterns
Form
Set
Line
....
VL-Generator
Generator of visual language implementations
• Abstract program tree
• Graphical representation, layout, editing
• Analysis and transformation
Eli
• Language processing
Tcl/Tk
• Graphical user-interfaces
Parcon
• Constraint solver

Figure 5: The VL-Eli System

The main concepts of VL-Eli are described by the three layers shown in Fig. 5: The VL-Generator generates visual structure editors from attribute grammar specifications. It is built on top of tools for graphics (Tcl/Tk), layout (the constraint solver Parcon) and for language implementation in general (Eli). The topmost layer contains variants of visual patterns, each of which encapsulates the implementation of visual language elements in terms of specifications for the VL-Generator. The visual patterns are applied by associating computational roles with certain contexts of the attribute grammar.

In the following we describe the implementation of (1) the visual structure editor and (2) interesting aspects of the translation to Java in more detail.

4.1 Specification of the Structure Editor

The visual representation is mainly specified by associating visual patterns to the language grammar. A visual pattern represents an abstract concept like a *list* visualization, which displays the language elements side by side in a row. For each abstract visual pattern concrete implementation variants are defined in terms of composable specification modules. Such a pattern variant encapsulates operations that are needed for a visual structure editor to implement a certain graphical representation of the structural abstraction. These are operations which

- draw graphical components, e.g. the oval surrounding a set,
- layout components of the structure, e.g. the elements of a set within its oval,
- provide facilities for user interactions, e.g. insert and delete elements.

We have chosen to provide automatic layout for all language constructs: The user only has to select new language elements and insert them into the desired place. The layout of the new structure is computed automatically. Often visual languages suffer from editors with a too general layout concept: After a small change, the user has to spend much time to achieve a clear layout again.

The *SIMtelligence Designer/J* language is a typical example for a language with a rich, list-based tree structure, which is ideal for automatic layout. Because the language is completely implemented by applying visual patterns, the automatic layout comes almost for free. Of course, automatic layout may have drawbacks, too: The user is not able to modify the layout to introduce secondary notations [9] or to optimize the layout for printing.

The language is mainly implemented by using the *List* and *Form* pattern. The *List* pattern is used to specify the menu tree (which are nested lists), the statement sequence, the branches of the select statement, and the concatenation of string values. A *Form* is a compact visual object with a constant number of sub-elements, which have a fixed relative position. The *Form* pattern is used as root for both views and to specify the representation of the SIM Toolkit function calls. The sub-elements of the latter form are the parameters of the function call.

The applied patterns support direct manipulation [19]: The user can drag new language elements from toolbars on the left side and drop them where appropriate. The elements can be selected, and moved or deleted as required. Editing is assisted by highlighting the nearest location for inserting a moved language element.

4.2 Code Generation

We used attribute grammars and other tools of the Eli system [13] to specify the code generation. The visual environment generates Java source code that can be compiled, converted into cap-files (a standard format for Java card packages), and loaded onto the SIM card.

We decided to implement Java source code rather than Java bytecode even though the generation of bytecode would prevent a serious problem: The lack of goto-statements in Java. However, the generation of Java source code is easier and the result can be checked with less expense during the development. Further, application developers like to have the source code, so that it is possible to make changes by hand if necessary.

One of the most serious problems is the lack of goto-statements: Whenever a backward move is triggered, the execution has to proceed at the preceding interactive statement. We solved this problem by introducing a state variable for interactive statements. The frame of a procedure is a loop with a nested switch statement over the state variable. Thus, to jump to another interactive statement the state variable is changed and a new loop iteration is initiated.

4.3 Related Work

There are a lot of tools to support the implementation of visual languages. Most approaches agree, that there should be an underlying abstract structure. There are two ways to implement the relation between the visual expressions and the underlying structure: Some systems [5, 6, 15] support a free, almost generic visual editor and use parsing techniques to derive the abstract structure. Other systems [2, 3] use structure-editors to modify the underlying structure directly. Often, the visual representation is derived from the abstract structure by unparsing and pretty-printing techniques. Some approaches support free editing *and* structure editing [15, 18]. Since VL-Eli belongs to the systems that generate structure-editors, we will focus on this area.

The abstract structure of a visual language can be specified in many different ways. Two important methods are graph-grammars [15, 18] and model-based specifications [16, 7, 10]. In model-based specifications object types and relations between them are specified in a declarative way. A popular language for such specifications is UML structure diagrams [17]. We count the tree-grammar based approach of VL-Eli to this group, because tree grammars also specify certain object types and (hierarchical) relations. The main advantage of tree grammars in this respect is, that they allow for efficient attribute evaluator generators, which can be used to specify the visual representation and the code generation.

Again, there are many ways to specify the graphical representation of a visual language. Metacase tools provide a high level, but fixed visual specification mechanism. The specification concept is usually restricted to graph-like visual languages, i.e. there is no support for deep nesting of language constructs. Other approaches use a more general specification concept. In [3, 18] graphical primitives and layout constraints are associated to the language structure. The layout is computed by a constraint-solver. Constraint-solver based specifications are very convenient for certain kinds of visual representations, but in some cases they are too restricted and sometimes there are efficiency problems. Other systems [2, 8] and Vl-Eli use attribute computations to specify the representation. Those specifications are more complex, but they allow a wider range of visual representations. VL-Eli addresses this by an additional specification level: A predefined set of visual patterns. In this way, VL-Eli achieves both, a wide application area and a high specification level.

Not all tools have a sufficient support for code generation. Metacase tools have specialized "report generator" languages. In MetaEdit+ [16] the object structure can be traversed and object attributes can be sent to the output stream. This is suitable for documentation- and simple code generation, but not suitable in the presence of more complicated dependencies. VL-Eli as well as other tools [15] use attribute computations to specify the translation. Since VL-Eli is based on the Eli tool-set [13] for translator construction, it offers many tools which solve a wide range of general language implementation tasks.

5 Conclusion

SIMtelligence Designer/J is a tool that has been created to assist the design and implementation of SIM application toolkit software. Such software extends the functionality of mobile phone SIMs in a manufacturer and provider independent fashion. In the case of Java Card SIMs, STK software is written in the Java card subset of the Java language. STK software turns out to be rather complex to write manually due to memory restrictions, missing language features and the necessity to provide a back-step function.

SIMtelligence Designer/J is based on a visual structure editor for a domain specific language and is generated from specifications by the VL-Eli generator. The language is tailored to the design of SIM Toolkit software. It provides constructs to specify the top level menu structure as well as language elements to describe the control flow of individual functions. It does not try to support mechanisms of general programming languages. For example, there is no support for arithmetic expressions or string processing. We think, that it is not reasonable to transfer these features to the visual level, because there is no gain of abstraction and the visual language would be more complex. Instead, we provide mechanisms to integrate Java code for these aspects.

We consider the current state as core language. The development is not finished yet. Future evaluation will surely disclose features that are missing in the language. In the future we will elaborate the support for other events in more detail. Additionally, we will develop concepts for a more general control flow structure. The difficulty is, that the support of the backward move has to be considered for each new control flow construct. For example it is not obvious how a backward move should behave in the case of a loop.

Although the language was designed for non-programmers, developers of the Orga company evaluated the language, too. Their first comments were very encouraging. To comply with their needs, we plan to integrate a more flexible method to extend the visual specification by pieces of java code.

References

[1] *3rd Generation Partnership Project; Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equiplent (SIM - ME) interface*. ftp://ftp.3gpp.org/specs/archive/11_series/11.14/.

[2] B. Backlund, O. Hagsand, and B. Pherson. Generation of visual language-oriented design environments. *J. of Visual Lang. and Comp.*, 1(4):333–354, 1990.

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

[4] Zhiqun Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison-Wesley, Reading, MA, USA, 2000.

[5] Sit Sen Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proc. of the 11th Annual Symp. on User Interface Software and Technology*, pages 185–194, 1998.

[6] G. Costagliola et al. Supporting hybrid and hierarchical visual language definition. In *1999 IEEE Symp. on Visual Lang.*, pages 236–243. IEEE Comp. Soc. Press, 1999.

[7] Robert Esser and Jörn W. Janneck. Moses: A tool suite for visual modeling of discrete-event systems. In *Symposia on Human-Centric Computing*, pages 272–279. IEEE Computer Society, September 2001.

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

[9] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. of Visual Lang. and Comp.*, 7(2):131–174, 1996.

[10] Honeywell, Inc. Dome guide, 1999. http://www.htc.honeywell.com/dome/DOMEGuide.pdf.

[11] Matthias Jung, Uwe Kastens, Christian Schindler, and Carsten Schmidt. A Pattern-Based Generator for Implementation of Visual Languages. In *Proceedings 2000 IEEE International Symposium on Visual Languages*, pages 71–72, Seattle, Washington, September 2000. IEEE Computer Society Press.

[12] Matthias Jung, Uwe Kastens, Christian Schindler, and Carsten Schmidt. Visual languages: Generating structure-editors from pattern-based specifications. Technischer Bericht, Reihe Informatik tr-ri-00-214, Universität Paderborn Fachbereich Mathematik-Informatik, October 2000. http://www.uni-paderborn.de/fachbereich/AG/agkastens/paper/vleli-tr-ri-00-214.ps.gz.

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

[14] Uwe Kastens and Carsten Schmidt. VL-Eli: A generator for visual languages. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[15] O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000)*, 2000.

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

[17] Object Management Group. *OMG Unified Modelling Language Specification*, 2001. http://www.omg.org/cgi-bin/doc?formal/01-09-67.

[18] 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.

[19] B. Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.