COCOVILA – Compiler-Compiler for Visual Languages

Pavel Grigorenko, Ando Saabas and Enn Tyugu [1]

Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21
12618 Tallinn
Estonia

Abstract

A compiler-compiler for visual languages is presented. It has been designed as a framework for building visual programming environments that translate schemas into textual representation as well as into programs representing the deep meaning of schemas. The deep semantics is implemented by applying attribute grammars to schema languages; attribute dependencies are implemented as methods of Java classes. Unlike compiler-compilers of textual languages, a large part of the framework is needed for support of interactive usage of a visual language.

Key words: compiler-compiler, extended attribute grammars, visual languages.

1 Shallow and deep semantics of visual languages

Our idea has been to develop a compiler-compiler for visual languages analogous to the compiler compilers of programming languages, to be used as a tool for rapid development of domain-specific visual languages. To be able to describe both syntax and semantics of languages considered, we have restricted the class of languages to schema languages with well-defined abstract syntax. On the semantic side, we are able to specify precisely shallow semantics that produces a textual representation of schemas without loss of essential information included in a schema. Furthermore, we give a general way to implement deep semantics of schemas, i.e. to generate an executable code from a schema, using in essence an extension of attribute grammars to schema languages.

The attempts to generate visual environments automatically have been made earlier, a good representative is [1]. Our framework differs from the

[1] Email: {pavelg, ando, tyugu}@cs.ioc.ee

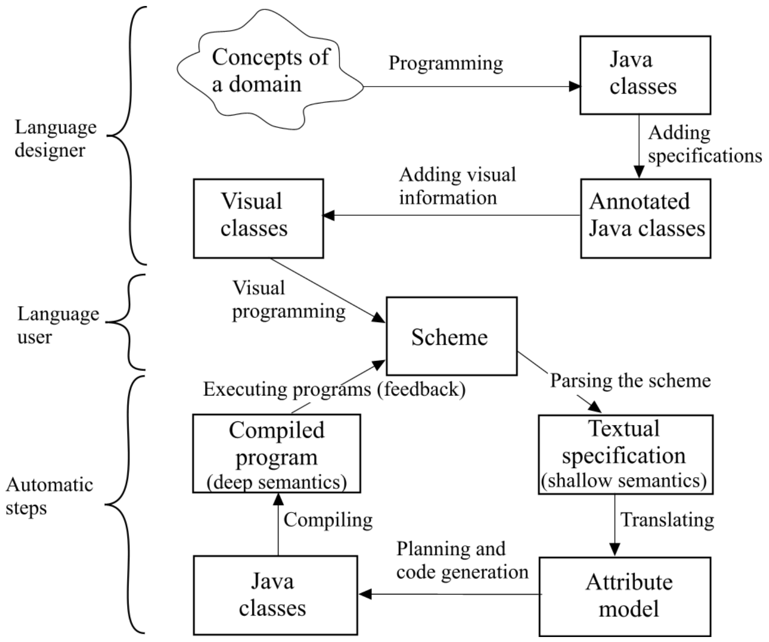


Fig. 1. Visual programming

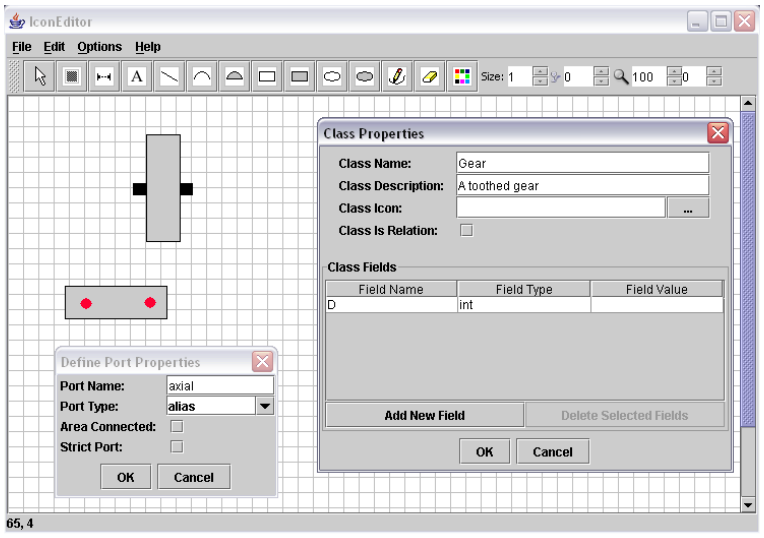
earlier ones by its capability to translate a visual sentence in principle into an arbitrary code using semantic programs run as implementations of attribute dependencies.

Figure 1 describes our approach to development of a visual language and using it for programming. The compiler-compiler COCOVILA supports a language designer in the definition of visual languages, including the specification of graphical objects, syntax and semantics of the language and provides the user with a visual programming environment, which is automatically generated from the visual language definition. When a scheme is composed by the user, the following steps – parsing, planning and code generation – are fully automatic. The compiled program then provides a solution for the problem specified in the scheme, and the results it provides can be fed back into the scheme, thus providing interactive properties.

In several ways, COCOVILA is similar to meta-modelling tools such as MetaEdit[2] or AToM3 [3]. However, our treatment of scheme semantics is quite different. Unlike the latter we are using a technique of dynamic evaluation of attributes of the syntactic graph underlying the scheme.

2 Class Editor

From a user's point of view COCOVILA consists of two components: Class Editor and Scheme Editor. The tool for a visual language developer is Class Editor, which supports the language designer in defining the visual aspects of classes, but also some of their logical and interactive aspects. The functional properties of visual classes are implemented by annotated Java classes. The



Class Properties

Class Name: Gear

Class Description: A toothed gear

Class Icon:

Class Is Relation: ☐

Class Fields

Field Name	Field Type	Field Value
D	int	

Define Port Properties

Port Name: axial

Port Type: alias

Area Connected: ☐

Strict Port: ☐

65, 4

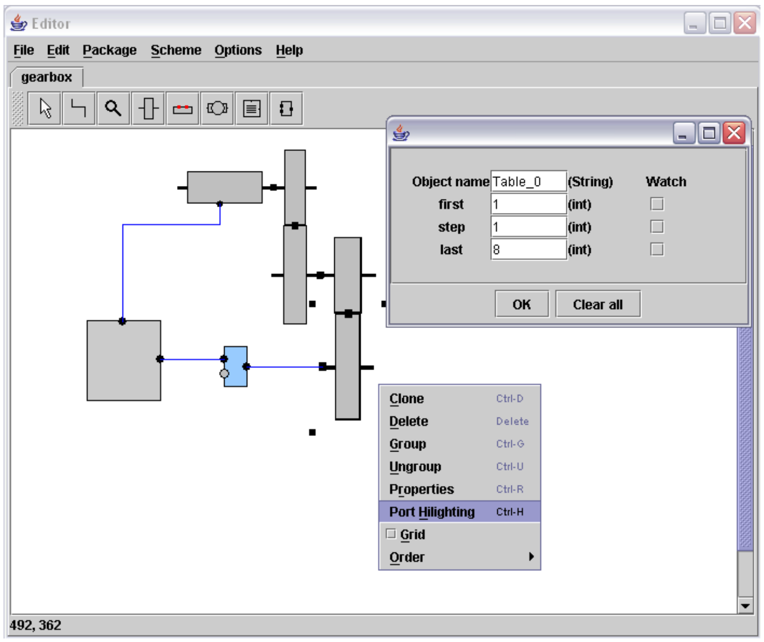
Fig. 2. Class Editor window

class editor is used to map domain concepts to visual classes as described in Figure 1. Its main window is shown in Figure 2. Pop-up windows for defining port properties and for developing an object window of the class are visible there as well. Results of a visual language development are stored in a package that is usable by the Scheme Editor. The user interface for using a visual language in Scheme Editor is automatically generated from the language definition given in Class Editor.

3 Scheme Editor

The Scheme Editor is a tool for the language user. It is intended for developing schemes and for compiling (synthesizing) programs from the schemes according to the specified semantics of a particular domain. The scheme editor is implemented using Java Swing library. It provides an interface for visual programming, which enables one to compose a scheme from shapes of classes. The environment generated for a particular visual language allows the user to draw, edit and compile visual sentences (schemes) through language-specific menus and toolbars. Figure 3 shows the scheme editor in use, when a package for calculating loads and kinematics of a gearbox has been loaded. Gears are connected to each other by arranging them on top or next to each other; lines connect other objects (motor and monitoring device). The toolbar at the top of the scheme is used for adding objects and relations to the scheme. One pop-up window is designed for instantiating object attributes, another pop-up window is designed for manipulating the scheme - deleting and arranging objects etc.

The scheme editor is fully syntax directed in the sense that the correctness of the scheme is forced during editing: drawing syntactically incorrect diagrams is impossible.



Object name	Table_0	(String)	Watch
first	1	(int)	☐
step	1	(int)	☐
last	8	(int)	☐

Fig. 3. Scheme Editor

The way to handle large schemes in the scheme editor is to use hierarchical composition in building the scheme. Any part of a scheme can be encapsulated as a separate class, so a large scheme can consist of a hierarchy of schemes, where each scheme object can contain subschemes. This means that schemes can be viewed in several different levels of abstraction, in order to encapsulate and manipulate parts of the scheme which are relevant to a particular issue.

4 Compiler

The deep semantics of schemes has been implemented in our framework on Java platform in such a way that a synthesized program becomes a method in a new Java class. Functional dependencies between attributes of an object of a schema are given either as methods of a class representing the object, or by equations. Equations and the usage of methods as functional dependencies are specified in a textual specification added to every class that can represent an object in a scheme. An important point of the implementation is that attributes of objects of a scheme are not components of the class of the object. They become components of the synthesized class, and are passed as parameters to methods. The logic of program synthesis needed for the attribute evaluation has been explained thoroughly in the paper [4].

5 Demo Packages

Several packages have been developed in this framework: a package for calculating loads and kinematics in a gearbox (shown in Figure 3), a package for analyzing logical schemes, a package for designing mechanical drives etc. Also, several UML modelling tools have been implemented in the framework, for example class diagram, use case diagram and state chart diagram tools.

As the experiments have shown, rapid prototyping of visual languages can be quite effective in the framework. For example, the implementation of the functional aspects (generating Java class file templates) of the class diagram editor was under 100 lines of Java code, and specification of the graphical and interactive aspects (which were also part of the language definition) was done visually easily in Class Editor (Figure 2).

The prototype implementation together with demo packages for analysis of logical circuits and mechanical drives as well as packages for UML modelling are accessible from `http://www.cs.ioc.ee/~cocovila/`.

Acknowledgements
This work has been supported by the Grant No. 5400 of the Estonian Science Foundation. Participation of Ando Saabas and Pavel Grigorenko in ETAPS 2005 was supported by the EU project eVikings II (IST-2001-37592).

References

[1] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora. Automatic Generation of Visual Programming Environments. IEEE Computer, 28(3):56-66, 1995.

[2] J-P. Tolvanen, M. Rossi. Metaedit+: Defining and Using domain-Specific Modeling Languages and Code Generators. In: OOPSLA 2003 demonstration, 2003.

[3] J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. Journal of Visual Languages and Computing, 15(3 - 4):309-330, 2004.

[4] M. Matskin and E. Tyugu. Strategies of structural synthesis of programs and its extensions. Computing and Informatics, 20:1-25, 2001.