Visual Tool for Generative Programming

Pavel Grigorenko
Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21
12618 Tallinn, Estonia
+372 6204212
pavelg@cs.ioc.ee

Ando Saabas
Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21
12618 Tallinn, Estonia
+372 6204212
ando@cs.ioc.ee

Enn Tyugu
Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21
12618 Tallinn, Estonia
+372 6204212
tyugu@cs.ioc.ee

Abstract. A way of combining object-oriented and structural paradigms of software composition is demonstrated in a tool for generative programming. Metaclasses are introduced that are components with specifications called metainterfaces. Automatic code generation is used that is based on structural synthesis of programs. This guarantees that problems of handling data dependencies, order of application of components, usage of higher-order control structures etc are handled automatically. Specifications can be written either in a specification language or given visually on an architectural level. The tool is Java-based and portable.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming – *program synthesis*; D.2.11 **[Software Architectures]**: Description and interconnection languages, domain-specific architectures; D.2.6 [**Programming Environments**]: Graphical Environments

General Terms

Design, Languages

Keywords: generative programming, compositional software engineering, model-based software engineering

1. INTRODUCTION

One can distinguish between two paradigms of software composition: object-oriented and structural. The object orientation is widely used and supported by UML as a standard.

The structural paradigm is older, but has been used mainly in specific application domains – first of all in simulation. Now it is becoming popular in composition of web services and in model-based software development. There is a principal difficulty in combining these two paradigms. Object-oriented approach uses widely encapsulation and hiding, and relies on parameter passing. Structural composition uses ports for binding components, it can provide considerable flexibility, and in some cases it is more fitted to represent ontology of a problem domain. We present a way to combine these two almost orthogonal dimensions of software architecture. This can be achieved by keeping implementations object-oriented and abstract specifications structural. We use components that are metaclasses. These components, together with universal binding rules, constitute a composition language. Architecture of a software system can be in many cases completely specified in this composition language. Components of real applications can be quite large, but due to hierarchical composition, each metaclass can be made not too big and well observable.

This presentation is written as follows: first we introduce metaclasses and metainterfaces, thereafter we describe software technology for using our tool and outline the composition language, and finally we describe the implementation.

2. METACLASSES AND METAINTERFACES

In our setting, components are *metaclasses* – Java classes supplied with *metainterfaces*. The latter are specifications written in the composition language. The concept of metaclass has had various meanings in object-oriented programming languages. It has been used mainly for representing class properties in a systematic way (eg class of classes). Our meaning of this concept is "*metaclass is a source of classes that can be synthesized from its metainterface and its methods*". Logically, the main part of a component is its metainterface. A class of a component is just an implementation of functionality specified in the metainterface. In particular, a metaclass may consist of a metainterface only (having an empty class), and then the whole implementation should be synthesized automatically from a metainterface. However, this is possible only when a metainterface refers to other metaclasses with nonempty implementations.

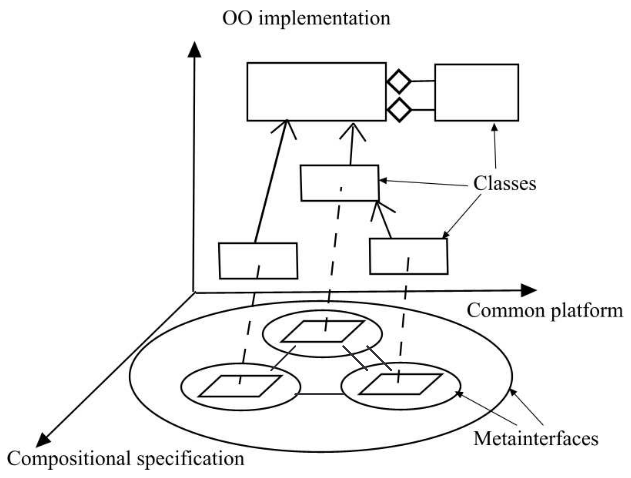


Figure 1. Classes, metaclasses and metainterfaces

Figure 1 shows three dimensions of software composition from metaclasses: compositional specification, object-oriented implementation, and common platform supporting both paradigms. On the compositional specification level (shown in horizontal plane), instances of metaclasses (ie classes with metainterfaces) are bound by equalities between their components. One metainterface (an oval shape) can wrap one class (a rectangular shape) or several metaclasses that can be bound with each other. We see four metaclasses there. Three of them have implementations as Java classes. The fourth includes them as components and has no own implementation. This supports the structural description. In another dimension, conventional object-oriented inheritance and aggregation are supported as shown by arrows between the classes in Figure 1. A new class for performing computations can be composed from its specification given as a metainterface and a goal. We can say that *metainterfaces are wrappers that provide flexibility to classes and contain information about their usability*.

3. PROGRAMMING TECHNOLOGY

We have found the combination of object-oriented and structural composition useful in rapid development of visual languages for modeling and simulation. Figure 2 describes the general approach we take for developing a visual language and using it for programming. The first steps described in the figure are taken by the developer of the language who is responsible for writing the necessary Java classes and metainterfaces, merging them into metaclasses and then adding visual extensions. It should be noted that these three steps do not have to be in that chronological order. In practice, there can be a large degree of parallelism and iteration between these steps.

When the visual classes have been built by software developers who must understand the problem domain as well, the language user need not be a software expert, but can work on the level of visual programming, arranging and connecting objects to create a scheme. Manipulating the scheme – a visual representation of a problem, is the central part of the user's activities. After a scheme has been built by the user, the remaining steps: parsing, planning and compiling are automatically taken by the computer.

In a way, the described approach is similar to those taken in meta-modeling tools such as MetaEdit+ [2] or AToM^3 [5]. The main difference from the latter lies in our use of metaclasses for specifying concepts in problem domains. The use of metaclasses allows us to use program synthesis for software composition, and integration with Java gives us an advantage of seamlessly integrating the Java API into the visual language.

4. COMPOSITION LANGUAGE

The composition language used for describing metainterfaces has to include at least the following constructs.

1. Specifications of interface variables

   type id, [*id,* ..]

2. Bindings

   var1 = var2

3. Axioms

   precondition -> postcondition{*implementation*}

Types can be primitive Java types (`int, double` etc), Java classes or metaclasses. *Bindings* are used for structural composition, to specify the equality of interface variables *var1* and *var2* (these can be components of other interface variables declared in the specification).

Axioms are written in a logical language, the choice of which depends on the availability of a prover to be used in the synthesizer. At present we use a logic with power of intuitionistic propositional calculus [6]. The *preconditions* of axioms can be conjunctions of propositional variables and implications of conjunctions of propositional variables. The *postconditions* are

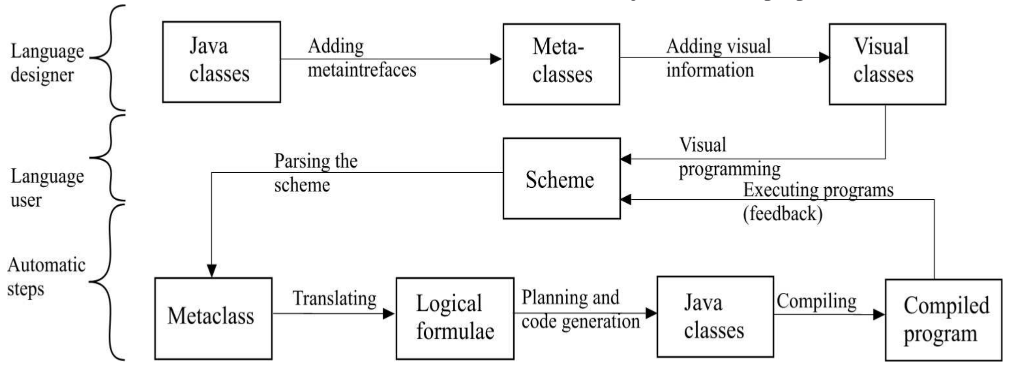


Figure 2. Visual generative programming

conjunctions of propositional variables. Name of any interface variable can be used as a propositional variable, denoting that value of the variable can be computed. An implication in a precondition denotes a new goal (see section 5.2) – a new computational problem whose algorithm has to be synthesized before the method with the precondition can be applied. This logic has been tested in several practically applied synthesizers, in particular, in the NUT system [7].

Implementation is a name of a method of the class being specified, or it is a keyword `spec` telling that the axiom states a new subgoal, i.e. a computational problem. In the latter case the axiom can have only conjunctions of propositional variables as pre- and postconditions.

Besides variable declarations, bindings and axioms, one can introduce more constructs in the specification language for convenience of writing specifications. We are using equations and aliases:

4. Equations
   exp1 = exp2

5. Aliases
   `alias` *name* = (*name*$_1$, ..., *name*$_n$)

Equations are given by two arithmetic expressions *exp1* and *exp2*. We have found equations especially useful for gluing components together and adjusting their data representation (units etc.). Which expressions are acceptable in equations depends on the equation solver that has to be a part of the supporting software.

Alias describes an interface variable whose type is a structure. This is needed for specifying data structures associated with ports (standard Java style would require an inner class for this purpose).

5. IMPLEMENTATION

5.1 Implementing metainterfaces

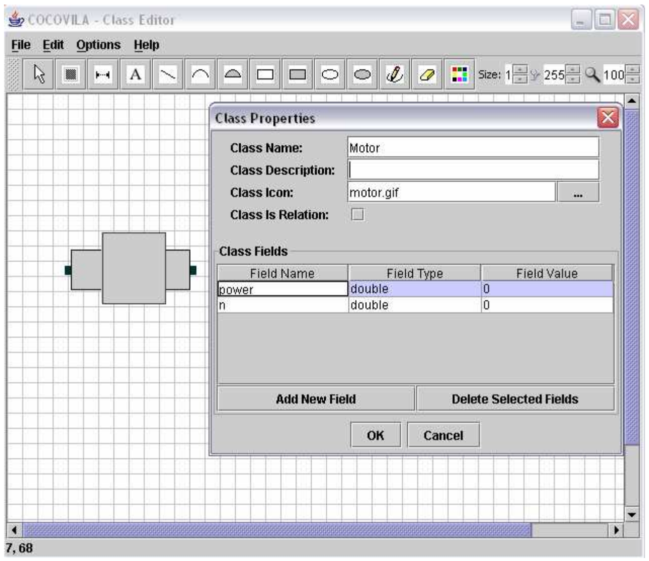
Metainterfaces are included as comments in Java files, between /*@ and @*/, so that our planner can use this information, but the Java compiler can safely ignore them in the process of compilation. In this sense it is similar to JML, a behavioral interface specification language for Java [3]. However, the latter is used for verifying the correctness of the code, while the purpose of a meta-interface is to enable automated composition of software.

5.2 Software generation

Having a metainterface, one gives a top-level goal in the form of an implication *P* -> *R* where *P* is a precondition and *R* is a postcondition (written in a suitable logical language) of the main method of the new class to be synthesized. The goal is to find a realization for the implication *P* -> *R* in the form of a new method of the new class *C*. This is achieved by a conventional scheme of deductive program synthesis: derive *P* -> *R* using logical formulae of specification as specific axioms, and extract the realization of the goal (a program) from its derivation. New subgoals may appear during this derivation, hence, more methods of the new class *C* may be synthesized completely automatically. Also instance variables of the new class may be introduced automatically, if needed. The synthesis method used is SSP – Structural Synthesis of Programs [1], [6].

6. Class Editor

The visual generative programming technology has been implemented in a tool CoCoViLa [8]. From a user's point of view the tool 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 their logical and interactive aspects (Figure 3). Functional properties of visual classes are implemented by means of metaclasses. The class editor is used to map domain concepts to visual classes as described in Figure 2. The user interface (including toolbars and menus) is automatically generated from the language definition. Results of a visual language development are stored in a package that is usable by the Scheme Editor.



Class Name: Motor
Class Description:
Class Icon: motor.gif
Class Is Relation: ☐

Class Fields

Field Name	Field Type	Field Value
power	double	0
n	double	0

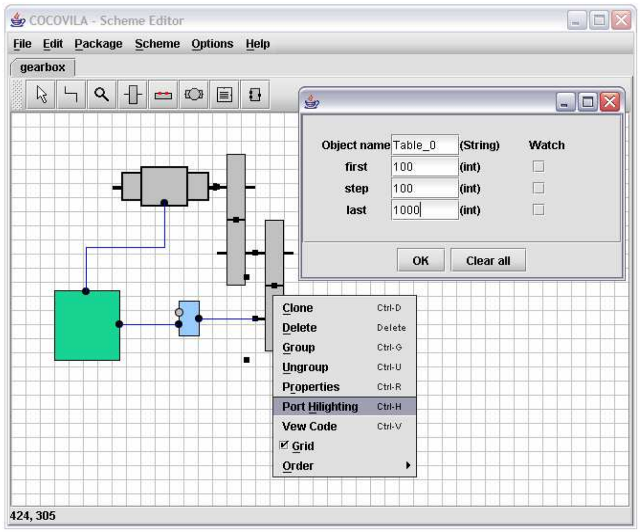
Figure 3. Class Editor window

6.1 Scheme Editor

The *Scheme Editor* is a tool for the language user. It is intended for developing schemes, compiling and running programs. It is used for generating (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 – putting together a scheme from visual images 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 4 shows the scheme editor in use, when a package for calculating the 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 for adding objects and relations to the scheme. One pop-up window is for instantiating object attributes, another pop-up window is 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	100	(int)	☐
step	100	(int)	☐
last	1000	(int)	☐

Figure 4. Scheme editor window.

7. RELATED WORK

There are several approaches to software composition in the context of visual domain-specific languages.

Vilpert is an object oriented framework for implementing visual languages in the Java language [4]. Implementing a visual language with Vilpert means generating a language analyzer based on a formal syntactic specification and implementing a graphical editor for manipulating the programs. The semantics of a language can be given by a set of methods to be invoked on the parse tree after parsing a scheme.

MetaEdit+ is a framework for creating CASE tools that comprise a domain specific visual language [2]. The heart of the approach is the creation of a metamodel that specifies the domain specific languages. Code generation can be enabled by defining a set of generators in the metamodel (specified in a custom scripting language), which transform models into some external format.

$AToM^3$ is a tool for meta-modeling and model transformations [5]. Operations that can be performed on models are defined through graph grammars

Having based our composition technique on object-oriented paradigm, and having taken Java environment as a platform, we should compare our components with Java beans. Beans are very successfully applied in development of graphic user interfaces, and less successfully in other areas, although at their introduction time the expectations were for wider usage of beans. The reason is obviously in weak support for scripting. Another kind of components is enterprise Java beans (EJB) - better scripting support is needed also here. We are quite satisfied with the design decision of using one and the same language for specifying components and specifying compositions. This decision has been justified by usage of common logic for both purposes. It works well in general, and supports hierarchical composition and extensibility as well.

Acknowledgements.

This work has been supported by grant no. 5400 of the Estonian Science Foundation and EU FP5 IST project eVikings II.

References

[1] S. Lämmerman, E. Tyugu. A Specification Logic for Dynamic Composition of Services. In: *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems* Workshops, Mesa, Arizona, 16-19 April, 2001.

[2] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: Defining and using domain-specific modeling languages and code generators. In OOPSLA 2003 demonstration, 2003.

[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR #98-06v, Department of Computer Science, Iowa State University, 2003.

[4] Antti-Pekka Tuovinen, VILPERT - visual language expert. In: Proceedings of the Sixth Fenno-Ugric Symposium on Software Technology FUSST'99, 1999.

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

[6] M. Matskin and E. Tyugu. Strategies of structural synthesis of programs and its extensions. Computing and Informatics, Vol. 20 (2001) 1 – 25.

[7] E. Tyugu and R. Valt, Visual programming in NUT. Journal of visual languages and programming, Vol. 8 (1997) 523 - 544.

[8] P. Grigorenko, A. Saabas, E. Tyugu. COCOVILA - Compiler-Compiler for Visual Languages. In: Proceedings of the Fifth Workshop on Language Descriptions, Tools and Applications LDTA'05, pages 101 - 105.