

CoCoViLa - Compiler-Compiler for Visual Languages

Pavel Grigorenko Mait Harf

Institute of Cybernetics
Tallinn University of Technology

Final Workshop of CDC 2002-2007
22 January 2008

CoCoViLa¹ is a visual programming framework for rapid design and implementation of domain specific visual languages.

It is being developed in Software Department of Institute of Cybernetics since 2003.

Implemented techniques (visual programming, automatic program synthesis) have previously been used in NUT, Priz, etc.

CoCoViLa is Open Source, Java based, platform independent and extendable system that is able to perform large-scale simulation tasks.

¹<http://cs.ioc.ee/~cocovila>

- A. Saabas “*A Framework for Design and Implementation of Visual Languages*” (master thesis) – 2004
- P. Grigorenko “*Program Synthesis in Java Environment*” (bachelor thesis) – 2004
- P. Grigorenko “*Attribute Semantics of Visual Languages*” (master thesis) – 2006
- Andres Ojamaa “*Modulaarne simuleerimisplatvorm*” (master thesis) – 2007
- Riina Maigre “*Visuaalse kasutaja-liidesega veebiteenuste tarkvara*” (master thesis) – 2007

Supervisor: Enn Tyugu

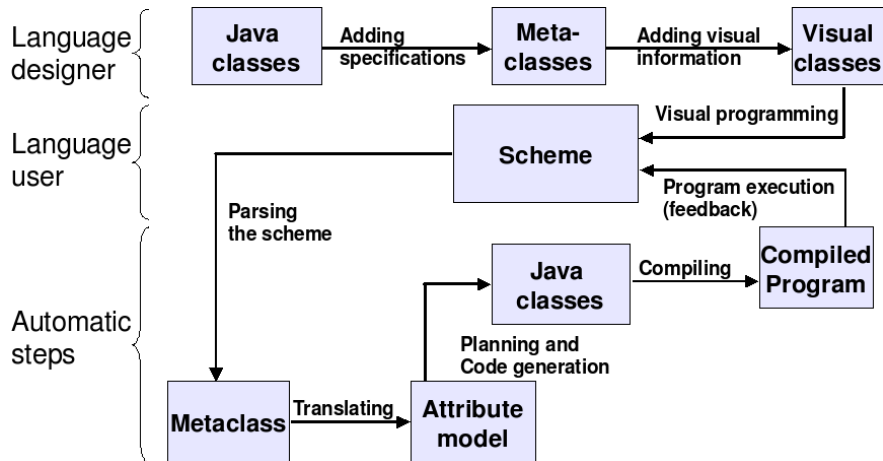
1 Programming Technology

2 Components

- Metaclasses and Metainterfaces
- Visual Classes and Schemes
- Specification Language
- Automatic Program Synthesis

3 Runnables

- Class Editor
- Scheme Editor



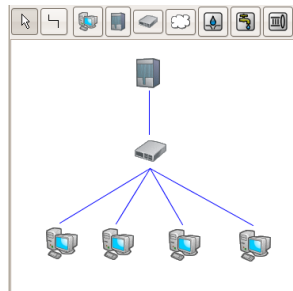
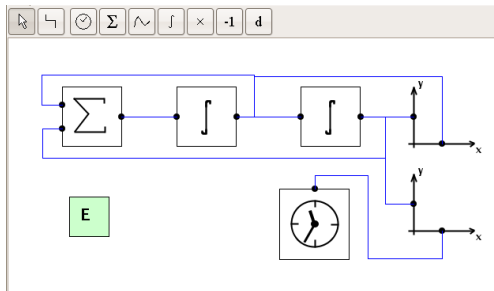
Metaclasses and Metainterfaces

- *Metaclass* is a Java class supplied with *metainterface*
- *Metainterface* is a specification of an object representing its usability as a component in terms of a problem domain

```
public class Resistor {  
    /*@ specification Resistor {  
        double r, u, i;  
        u = r*i;  
    } @*/  
    ...  
}
```

Visual Classes and Schemes

- Metaclasses have visual representations – *visual classes*.
- Visual class contains:
 - Icon (Toolbar)
 - Image (Scheme)
- Visual classes can be connected using ports.
- Schemes are visual specifications of problems to solve.



Specification Language

- Variable declarations

`<type> <identifier>`

- Bindings

`<var> = <var>`

- Valuations

`<var> = <constant>`

- Inheritance

`super <name>`

- Axioms

`precondition -> postcondition{impl}`

- Equations

`<exp> = <exp>`

- Aliases

`alias [<type>] <name> = (<var>, <var>, ...)`

Example

```
class Factorial {
    /*@ specification Factorial {
        double n, f, arg, val;
        n = 10;
        [ arg -> val ], n -> f, (java.lang.Exception) { fact };
        val = arg - 1;
        n -> f;
    } @*/

    double fact( Subtask s, double n ) throws Exception {
        if( n == 0 ) return 1;
        Object[] in = new Object[]{ n };
        Object[] out = s.run(in);
        return n * fact( s, ((Double)out[0]).doubleValue() );
    }
}
```

Automatic synthesis of programs is a technique for the automatic construction of programs from the knowledge available in specifications of classes. Having a specification of a class, we are, in general, interested in solving the following problem: find an algorithm for computing the values of components y_1, \dots, y_n from the values of components x_1, \dots, x_m . The automatic synthesis of programs is based on Structural Synthesis of Programs (SSP).

Specification

- Description of problem area (model, scheme, specification)
- Known values x_1, \dots, x_m (constants, initial values of components)
- Unknown values y_1, \dots, y_n (components to be computed)

Computational tasks

- Compute required values of components
- Compute all the values of components that is possible to compute


- The *Class Editor* is used for developing visual languages for different problem domains. This is done by defining metainterfaces of language components as well as their visual and interactive aspects.
- The *Scheme Editor* is a tool for creating schemes, compiling and running programs defined by a scheme and a goal. Algorithms of automatic program synthesis are embedded into the Scheme Editor and are not visible to the user.

Class Editor

CoCoViLa - Class Editor

File Edit Options Help

Size: 1 255 100 0



Edit Port Properties

Port Name: in2
Port Type: double
Area Connected:
Strict Port:
Multi Port:
OK Cancel

Class Properties

Class Name: Adder
Class Descripti...: adder
Class Icon: adder.gif
Class Is Relation:

Class Fields

Field Name	Field Type	Field Value
k1	double	
k2	double	
in1	double	
in2	double	
out	double	

Add New Field Delete Selected Fields
OK Cancel

732, 266

- The *Class Editor* is used for developing visual languages for different problem domains. This is done by defining metainterfaces of language components as well as their visual and interactive aspects.
- The *Scheme Editor* is a tool for creating schemes, compiling and running programs defined by a scheme and a goal. Algorithms of automatic program synthesis are embedded into the Scheme Editor and are not visible to the user.

Scheme Editor

The image displays the Scheme Editor interface for a project named "oscillator_2gr - CoCoViLa". The main window shows a block diagram of an oscillator circuit. The circuit consists of a summing junction (Σ) that receives an input from a block labeled "E" and a feedback signal from a graph. The output of the summing junction is integrated by two integrator blocks (∫) in series. The output of the second integrator is fed back to the summing junction and also plotted on a graph. The graph shows a sinusoidal signal. A configuration dialog box titled "Euler_0 - Dif" is open, showing the following parameters:

Object name	Euler_0	(String)	Input	Goal	Watch
time	450	(int)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
T	5	(double)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> Static					

The dialog has "OK", "Close", "Apply", and "Clear all" buttons. To the right, a "Dif - Specification" window shows the following Scheme code:

```
1 public class Dif extends Euler {
2   /*@ specification Dif super Euler {
3   Adder Adder;
4     Adder.k1 = -0.001;
5     Adder.k2 = -0.1;
6   Integrator Integrator_1;
7     Integrator_1.instate = 10;
8   Integrator Integrator_2;
9     Integrator_2.instate = 0;
10  Clock Clock_4;
11    Clock_4.instate = 0;
12    time = 450;
13    T = 5;
14  Graph Graph_1;
15    Graph_1.delay = 10;
16  Graph Graph_0;
17    Graph_0.delay = 10;
18  Integrator_2.in = Integrator_1.out;
19  Integrator_2.in = Adder.in1;
20  Graph_1.y = Integrator_2.out;
21  Integrator_1.in = Adder.out;
22  Graph_0.y = Integrator_2.out;
23  Graph_0.x = Clock_4.out;
24  Integrator_2.out = Adder.in2;
25  Graph_1.x = Integrator_1.out;
26  }*/
27 }
```

Below the code is a plot of the signal, showing a sinusoidal wave with an amplitude of approximately 40 and a period of 5 units. The x-axis ranges from 0 to 90, and the y-axis ranges from -70 to 40.

Scheme Editor

The image shows two windows from a software tool. The left window, titled "Algorithm Visualizer", contains a specification for a system with a clock and several components. The right window, titled "Dif - Specification", shows the corresponding Java code for the Dif-Specification.

```
Algorithm Visualizer
DF [X]
Clock_4 : instate = 0
spec : instate_name = "instate"
spec : finalstate_name = "finalstate"
spec : Adder.cocovilaSpecObjectName = "Adder"
spec : Adder.k1 = -0.001
spec : Adder.k2 = -0.1
spec : Integrator_1.cocovilaSpecObjectName = "Integrator_1"
spec : Integrator_1.instate = 10
spec : Integrator_2.cocovilaSpecObjectName = "Integrator_2"
spec : Integrator_2.instate = 0
spec : Clock_4.cocovilaSpecObjectName = "Clock_4"
spec : Clock_4.instate = 0
spec : time = 450
spec : T = 5
spec : Graph_1.cocovilaSpecObjectName = "Graph_1"
spec : Graph_1.delay = 10
spec : Graph_0.cocovilaSpecObjectName = "Graph_0"
spec : Graph_0.delay = 10
Adder : k1 = k1s
Adder : k2 = k2s
spec : alias (double) instate = (^ .instate)
spec : allT.length, T -> allT[setT]
spec : instate_name, instate -> done_print_instate (print_state)
spec : [ state -> nextstate, draw, instate, time -> finalstate (proc_run)
  (Subtask [ state -> nextstate, draw ] :
    Integrator_1 : out = state
    Integrator_2 : out = state
    Clock_4 : nextstate = state + 1/T
    Clock_4 : out = state
    spec : Integrator_2.in = Integrator_1.out
    spec : Graph_1.x = Integrator_1.out
    spec : Graph_1.y = Integrator_2.out
    spec : Graph_0.y = Integrator_2.out
    spec : Integrator_1.out = Adder.in2
    spec : Graph_0.x = Clock_4.out
    Integrator_2 : nextstate = state + in / T
    spec : Integrator_2.in = Adder.in1
    Graph_1 : cocovilaSpecObjectName, x, y -> delay_s(getDelay)
    Graph_0 : cocovilaSpecObjectName, x, y -> delay_s(getDelay)
    Adder : out = k1*s + in1 + k2*s + in2
    Graph_1 : x, y, delay_s -> drawing_ready(draw)
    Graph_0 : x, y, delay_s -> drawing_ready(draw)
    spec : Integrator_1.in = Adder.out
    spec : alias draw = (^ .drawing_ready)
    Integrator_1 : nextstate = state + in / T
    spec : alias (double) nextstate = (^ .nextstate)
  )
end of spec : [ state -> nextstate, draw, instate, time -> finalstate (proc_run)
spec : finalstate_name, finalstate -> done_print_finalstate (print_state)

Dif - Specification
Specification Program Run results
Compile & Run Propagate Font Size: 12 Bold
89 Clock_4.nextstate = Dif.this.Clock_4.nextstate;
90 Clock_4.finalstate = Dif.this.Clock_4.finalstate;
91 Graph_0.cocovilaSpecObjectName = Dif.this.Graph_0.cocovilaSpecObjectName;
92 Graph_0.x = Dif.this.Graph_0.x;
93 Graph_1.delay_s = Dif.this.Graph_1.delay_s;
94 }
95
96 public Object[] run(Object[] in) throws Exception {
97 //Subtask: [this state] -> [this.nextstate, this draw]
98 double[] alias_state_0 = {Double.valueOf(
99 Integrator_1.state = ((java.lang.Double)alias_state_0[0]).doubleValue();
100 Integrator_2.state = ((java.lang.Double)alias_state_0[1]).doubleValue();
101 Clock_4.state = ((java.lang.Double)alias_state_0[2]).doubleValue();
102
103 Integrator_1.out = Integrator_1.state;
104 Integrator_2.out = Integrator_2.state;
105 Clock_4.nextstate = (Clock_4.state + (1 / Clock_4.T));
106 Clock_4.out = Clock_4.state;
107 Integrator_2.in = Integrator_1.out;
108 Graph_1.x = Integrator_1.out;
109 Graph_1.y = Integrator_2.out;
110 Graph_0.y = Integrator_2.out;
111 Adder.in2 = Integrator_2.out;
112 Graph_0.x = Clock_4.out;
113 Integrator_2.nextstate = Integrator_2.state + (Integrator_2.in / Integrator_2.T);
114 Adder.in1 = Integrator_2.in;
115 Graph_1.delay_s = Graph_1.getDelay(Graph_1.cocovilaSpecObjectName, Graph_1.x, Graph_1.y);
116 Graph_0.delay_s = Graph_0.getDelay(Graph_0.cocovilaSpecObjectName, Graph_0.x, Graph_0.y);
117 Adder.out = ((Adder.k1*s + Adder.in1) + (Adder.k2*s + Adder.in2));
118 Graph_1.draw(Graph_1.x, Graph_1.y, Graph_1.delay_s);
119 Graph_0.draw(Graph_0.x, Graph_0.y, Graph_0.delay_s);
120 Integrator_1.in = Adder.out;
121 Integrator_1.nextstate = Integrator_1.state + (Integrator_1.in / Integrator_1.T);
122
123 double[] alias_nextstate_1 = new double[] { Integrator_1.nextstate, Integrator_2.nextstate, Clock_4.nextstate };
124 Object[] alias_draw_2 = new Object[] {
125 return new Object[] { alias_nextstate_1, alias_draw_2 };
126 }
127 } //End of subtask: [this state] -> [this.nextstate, this draw]
128 Subtask_0.subtask_0 = new Subtask_0();
129
130 double[] alias_instate_97 = new double[3];
131 alias_instate_97[0] = Integrator_1.instate;
132 alias_instate_97[1] = Integrator_2.instate;
133 alias_instate_97[2] = Clock_4.instate;
134 double[] alias_finalstate_97 = proc_run.runSubtask_0, alias_instate_97, time);
135 Integrator_1.finalstate = ((java.lang.Double)alias_finalstate_97[0]).doubleValue();
136 Integrator_2.finalstate = ((java.lang.Double)alias_finalstate_97[1]).doubleValue();
137 Clock_4.finalstate = ((java.lang.Double)alias_finalstate_97[2]).doubleValue();
138
139 double[] alias_finalstate_100 = new double[3];
140 alias_finalstate_100[0] = Integrator_1.finalstate;
141 alias_finalstate_100[1] = Integrator_2.finalstate;
142 alias_finalstate_100[2] = Clock_4.finalstate;
143 print_state(finalstate_name, alias_finalstate_100);
144 }
```