CoCoViLa

- Declarative model-based software development
- Domain-specific languages
- Visual specification
- Program synthesis
- Java + OSS + GPL
- Portable and extensible
Visual DSLs in CoCoViLa

• Automatic Editor generation from language definition
• Well-defined abstract syntax
• No need to implement custom parsers and code generators

• Precise **semantics:**
  • **Shallow** semantics ⇒ textual representation
  • **Deep** semantics ⇒ executable code
(Pre)History

• Research in Artificial Intelligence

• Structural Synthesis of Programs (E. Tyugu 1979; G. Mints, E. Tyugu 1982)

• 1980s - PRIZ/UTOPIST

• 1990s - NUT

• >= 2003 - CoCoViLa
Structural Synthesis of Programs

• Structural properties of computations

• Based on implicational fragment of intuitionistic propositional logic (uses Curry-Howard correspondence)

• Deductive method:
  (specification $\Rightarrow$ existence theorem $\Rightarrow$ proof $\Rightarrow$ code)

• Complexity: Polynomial-space complete
  (Statman ’78)
Computational problems in SSP

Knowing $M$, compute $y_1, \ldots, y_n$ from $x_1, \ldots, x_m$

Example:

(Triangle; $a, b, c \Rightarrow S$)

Triangle

**var:**
- $a, b, c, p, S$;

**rel:**
- $a, b, c \Rightarrow p \{sp\}$;
- $a, b, c \Rightarrow S \{h\}$;
- $a, b, c \Rightarrow S (\lambda abc. h(a,b,c,sp(a,b,c)))$;

$\vdash a, b, c \Rightarrow p \{sp\}; a, b, c \Leftarrow a, b, c$ ($\Rightarrow$)
$\vdash a, b, c \Rightarrow S \{h\}; a, b, c \Leftarrow a, b, c, p$ ($\Rightarrow$)
$\vdash a, b, c \Leftarrow S (h(a,b,c,sp(a,b,c)))$ ($\Rightarrow$+)
$\vdash a, b, c \Rightarrow S (\lambda abc. h(a,b,c,sp(a,b,c)))$
visual language lifecycle
public class AndGate {
    /*@
    specification AndGate {
        int in1, in2, out;
        in1, in2 -> out {calc};
    }
    @*/

    public int calc( int x, int y ) {
        return Math.min(x, y);
    }
}

Domain Concepts

Visual classes

Scheme

Programming

Java classes

Add spec

Metaclasses

Designer

User

Automatic

Compose

Execute (w/ feedback)

Bytecode

Compile

Java classes

Planning and code generation

Transform

Translate

Metaclass

Computational model
Visual Class

- Metaclass
- Image/Graphics
- Ports
- Fields
- Toolbar icon
CoCoViLa Class Editor

A diagram of a class editor with a schematic of a logic gate and a properties editor showing fields for class name, description, icon, and fields with field names, types, and values.
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE package SYSTEM "package2.dtd">
<package>
  <name>Logic</name>
  <description>Logical circuits builder</description>
  <class type="class">
    <name>And</name>
    <description>And</description>
    <icon>images/and.png</icon>
    <graphics>
      <bounds height="83" width="202" x="0" y="0"/>
      <image fixed="false" height="100"
        path="/images/200px-AND_ANSI.png" width="200" x="3" y="-11"/>
    </graphics>
    <ports>
      <port name="in1" portConnection="" strict="false" type="int"
        x="13" y="19"/>
      <port name="in2" portConnection="" strict="false" type="int"
        x="13" y="59"/>
      <port name="out" portConnection="" strict="false" type="int"
        x="190" y="39"/>
    </ports>
    <fields>
      <field name="in1" type="int"/>
Domain Concepts

Visual classes

Compose

Scheme

Programming

Add spec

Java classes

Metaclasses

Designer

User

Automatic

Java classes

Bytecode

Execute (w/ feedback)

Translate

Metaclass

Transform

Computational model

Planning and code generation

Compile

Planning and code generation
CoCoViLa Scheme Editor

- Instantiate objects
- Connect ports
- Assign values
- Assign a superclass
Scheme to Metaclass

```java
public class Logic extends Process {
    /**
     * @specification Logic super Process {
     * NumGenerator NumGenerator_0;
     * Delayer Delayer_0;
     * time = 20; -
     * static VisualizerM VisualizerM_0;
     * And And_2;
     * Or Or_0;
     * Not Not_0;
     * Or Or_1;
     * And And_3;
     * NumGenerator NumGenerator_1;
     * Delayer Delayer_1;
     * Delayer_0.in = NumGenerator_0.out;
     * Or_0.inl = NumGenerator_0.out;
     * Or_0.in2 = Delayer_0.out;
     * Delayer_1.in = NumGenerator_1.out;
     * Or_1.inl = NumGenerator_1.out;
     * Not_0.in = Or_1.inl;
     * Or_1.in2 = Delayer_1.out;
     * And_3.in2 = Or_1.out;
     * And_2.in1 = Or_0.out;
     * And_2.in2 = Not_0.out;
     * And_3.inl = And_2.out;
     * VisualizerM_0.inputs = { And_3.out };
    */
}
```
Metaclasses to Code
Domain Concepts

Java classes

Metaclasses

Visual classes

Scheme

Add spec

Programming

Draw

Compose

Execute (w/ feedback)

Compile

Java classes

Bytecode

Translate

Transform

Planning and code generation

Computational model

Automatic

Designer

User
Compilation

- Storing generated Java files: in memory or file system
- Eclipse JDT Core compiler
- Compiling Classloader (CCL)
Execution
Specification language

- Variables
  ```
  double a, b;
  AndGate and;
  ```
- Constants
  ```
  const double PI = Math.PI;
  ```
- Bindings (equality)
  ```
  a = b;
  ```
- Value assignments
  ```
  a = 3;
  and.in1 = 1;
  ```
Simple functional dependencies

\[
\text{in1, in2} \rightarrow \text{out } \{\text{calc}\};
\]

Realization in Java:

```java
public int calc( int x, int y ) {
    return Math.min(x, y);
}
```
Higher-order f. D-s (w/ subtasks)

[arg -> val], from, to, step -> done {loop};

Realization in Java:

```java
public void loop(Subtask s, int st, int fn, int inc) {
    for (int i = st; i <= fn; i += inc) {
        s.run(new Object[]{i});
    }
}

public interface Subtask {
    public Object[] run(Object[] args);
}
```
Higher-order dataflow

from, to, step  |  loop  | done

arg  |  val

Diagram showing a loop with input arguments and values, indicating a dataflow process.
Specification language

- Equations (built-in solver)

```plaintext
specification AndGate {
  double in1, in2, out;
  in1*in2 = out;
}

specification Complex {
  double re, im, arg, mod;
  mod^2 = re^2 + im^2;
  mod * sin(arg) = im;
}
```
• Tuples
  alias x = (a, b);
  alias y = (c, d);
  x = y;
  x.0 = y.1;
  x.length -> x {init};

• Wildcards
  alias ts = (*.t);
  x.*.1 = y.*.z;

• Inheritance
• Goals
• Polymorphic types
• Control variables
• ...

Specification language
Applications

- Hydraulic systems
- GrADAR model
- Security Costs Optimization
- Web-service composition
- Functional Constraint Networks
- Differential Equations
- Discrete Event Simulation
- Hybrid Network Simulation
- Logic Circuits
- Electric Circuits
- Gearbox kinematics
- Neural networks
- Petri nets
- Attack Trees
- UML diagrams
Thank you!
Modeling and simulation of an electro-hydraulic servo-valve
Composition of X-Road services
graded IT security Cost optimization
Starting CoCoViLa

• Requires Java 1.6

• How to run:
  • Java Web Start (http://cs.ioc.ee/cocovila -> Web Start)
  • Download zipped version and execute .jar
  • git clone
git://cocovila.git.sourceforge.net/gitroot/cocovila/cocovila -> ant run-se
Predator-Prey model (Lotka-Volterra)

• Prey (rabbits)
  \[ \frac{dR}{dt} = kR - aRW \]

• Predators (wolfs)
  \[ \frac{dW}{dt} = -rW + bRW \]

Parameters: \( k = .08, a = .001, r = .02, b = .00002, R = 1000, W = 30 \)