CoCoViLa at a glance

CoCoViLa is a software tool for *model-based* software development with a *visual language* support that performs *automatic synthesis* of programs from logical specifications. It is tightly integrated with Java: it is written in Java, uses advanced features of Java, and it supports programming of new software components in Java almost without restrictions.

Software technologies supported by CoCoViLa enable the developers to separate domain analysis, application design and implementation. It is expected that these processes are performed by people with different expertise, and it provides simple visual development tools for their communication. One can distinguish two different ways of software development in it: 1) design and implementation of large applications and 2) development of domain-oriented software packages with visual languages. They must be supported by different software technologies. Development of large software systems requires support of a heavyweight technology. A domain-oriented package with a visual language can easily be developed in an agile way. Some examples of applications of this kind are a package for simulation of hydraulic systems and a package for synth
esis of services on large service models.

CoCoViLa consists of two runnables: *Class Editor* and *Scheme Editor*. The Class Editor is used for implementing visual languages for different problem domains. This is done by defining models of language components as well as their visual and interactive aspects. The Scheme Editor is a tool for drawing schemes, compiling and running programs defined by a scheme and a goal.

Model-based software development

Model-based software development is a way to overcome the increasing complexity of software products and their changeability. It is based on dividing the software development into two separated processes: *domain engineering* and *application engineering*. Both include software development as a part. (Precise names for these processes should be domain software engineering and application software engineering.) The domain engineering provides software assets for the use in the application engineering. *Software assets* are the reusable resources used in application engineering. Examples of software assets include domain models, software architectures, code components and application generators with visual languages. This facilitates software development by raising the conceptual level of application programming. Application engineering results in the development of an *application* – a program that performs the required tasks, see Fig. 1.

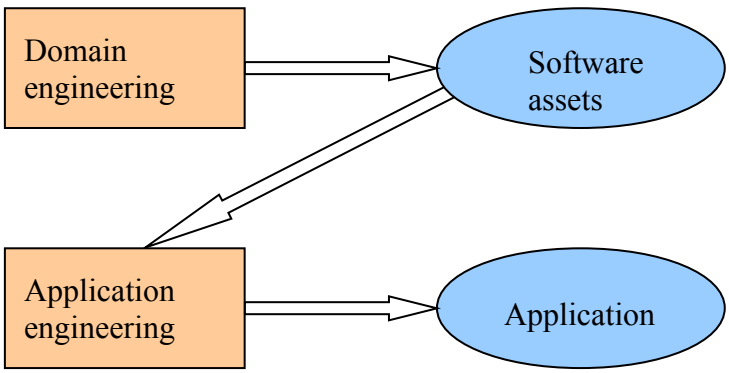

Figure 1. A general view of model-based software engineering

Software assets in CoCoViLa are organized around the visual language – they are concepts of domain implemented as components and represented by visual objects. The assets of one application domain constitute a *package*.

Visual language

A *visual language* developed in CoCoViLa is intended, first of all, for specifying models by drawing their *schemes* and adding data to components of schemes. A scheme is always translated into a text that is a *specification* of an application. A scheme can be used also as an environment of communication with an application – for controlling its execution and displaying results. Schemes are handled by the scheme editor.

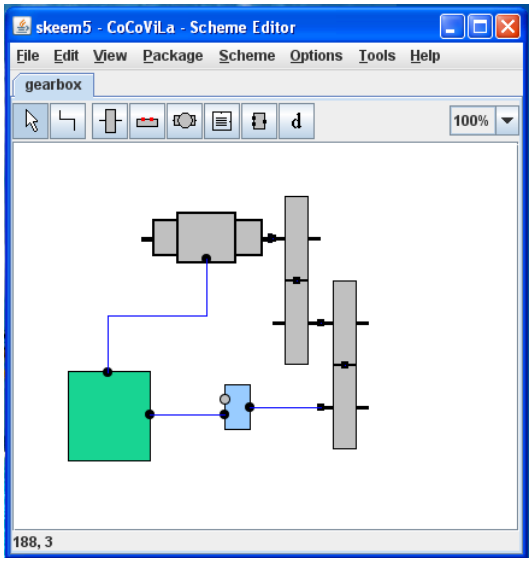

Figure 2. An example of scheme in a scheme editor window

Fig. 2 shows a scheme editor window that includes a scheme, and includes menus with all commands needed for development and usage of a scheme. The lower menu bar includes buttons for components available for developing a scheme in the selected context. These buttons represent the concepts of a particular domain oriented visual language.

Automatic synthesis of programs

A model presented as a scheme is automatically translated into a specification of an application in CoCoViLa. A specification always contains some information about the computability of variables (components) included in the specification. This information may be implicitly given in a specification of a data structure, e.g. if *a* is a component of *b*, then a value of *a* is always computable from a value of *b*. This information can be represented by simple equations, and it can be explicitly represented by axioms specifying the applicability of methods of a Java class. CoCoViLa uses this information automatically for synthesizing, if possible, a Java program that performs a task given by a *goal*, see Fig. 3. A goal specifies a task of computing values of output variables listed in the goal from given values of input variables listed in the goal. The program synthesis method is based on formal logic and it is called *structural synthesis of programs*.

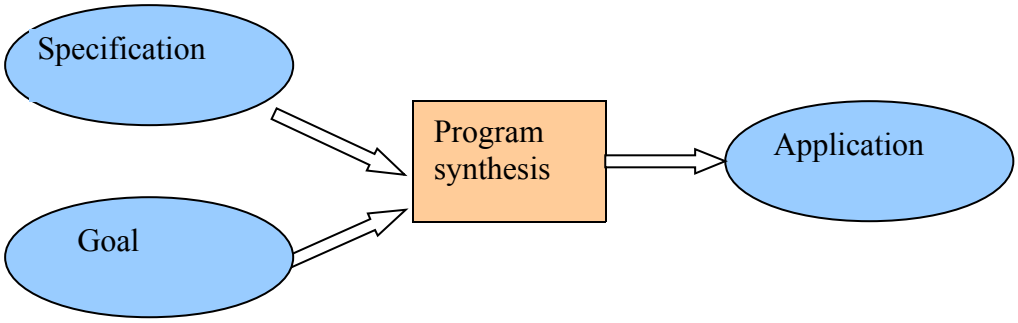

Figure 3. Program synthesis from a specification and a goal

Essential concepts

Model. Model is a representation of some entity (object, process, system etc.) by means of abstraction in terms of an application domain. A model is specified by defining its components, evaluating some parameters of them, and defining bindings between the components. A model can be unfolded into a flat form that contains only variables and functional dependencies for computing the variables. Model is a meaning of a specification for the user.

Rich component. Rich component is a description of a domain-specific concept used for describing models. It is a class, extended with information needed for automatic usage of the class, and also for visual handling of instances of the class. It is therefore also called visual class. A rich component may have four parts:
• visual part – its image, pop-up window etc.
• specification (a logical part)
• program component
• daemon.

The *visual par*t is for the interaction with a user. The *specification* and the *program component* constitute a *metaclass*. This is a Java class, where a specification (i.e. the logical part) is included as a text. This specification is called also *metainterface,* because it determines which new methods can be synthesized on the metaclass. The *daemon* is another Java class related to the rich component that describes a thread started by a user, if needed. Using daemons enables one to develop flexible interfaces to programs. Not all rich components have to include daemons. Example of a rich component `Boiler` is presented in Fig. 4. We see two Java classes `Boiler` and `BoilerDaemon` as well as visual image of the rich class there. The `Boiler` class written in Java and shown on the right side of Fig. 4 is a metaclass – it includes two parts of the rich class: logical part and program component. The left side of the Fig. 4 shows the boilers image and the daemon.

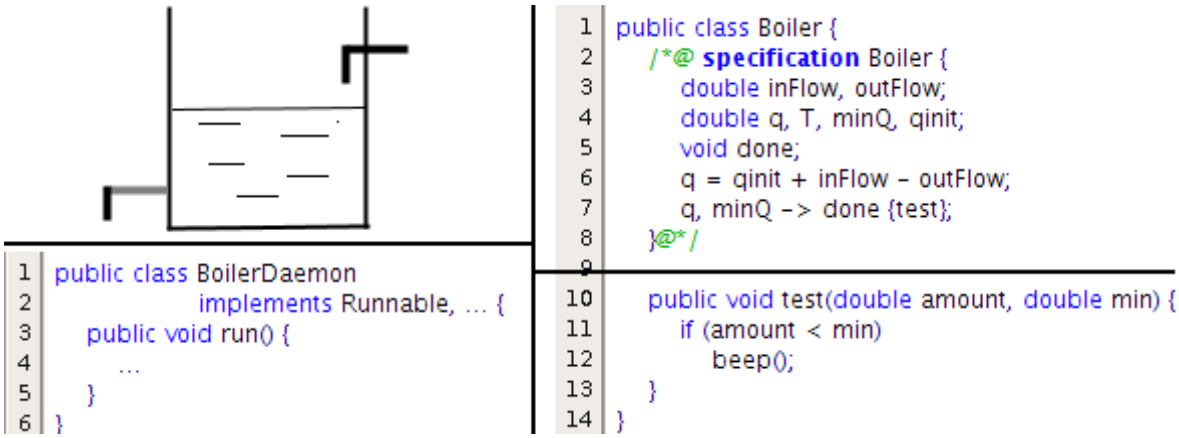

Figure 4. Rich component

Specification.
Specification is a text describing a model. It represents information about computability of variables occurring in the model. It includes variable declarations and information about the functions that can be used for computing values of the variables.

Core of the specification language has three types of statements in it:

1) Declaration of variables
type id[,id,...]

The type is a primitive type, a class, or a metaclass.

2) Binding
a = b

Binding is an equality, where *a, b* are variables.

3) Axiom
precondition -> variable{implementation}

Axiom specifies a function that can be used for computing a value of the *variable* on the right side of the arrow by using the *implementation*. Precondition is a list of inputs of the function: variables and subtasks. Subtasks will be explained separately, and here we restrict the explanation to axioms with only variables on the left side. In this case an axiom specifies a function that computes a value of the variable on the right side of the arrow from values of its *input variables* written on the left side. *Implementation* is name of a method that performs the computations. Its parameters correspond positionally to the input variables. For example, an axiom

x,y -> z{P}

can specify the method

```
int P(int a, int b) {…}
```

Axioms have a precise logical meaning, where commas denote conjunction and arrows denote implication symbols. The variable symbols are translated into logic as propositions about the computability. The example axiom is as follows in logic:

"x is computable" & "y is computable" ⊃ "z is computable".

There are extensions of the core of the specification language: equations, aliases, simple inheritance that are reducible to the core by simple transformations. These extensions increase the convenience of usage of the language.

Subtask. Subtask is a goal for synthesis of a value of a functional variable which is an input variable of a function described by an axiom where the subtask occurs. This description has the form

[precondition -> postcondition]

where pre- and postcondition are respectively lists of input and output variables of the multifunction (a function with several outputs) that is the value of the functional variable. A method described by an axiom with subtasks can be used in computations only if all its subtasks are solvable and all its input variables have values. The solvability of a subtask means that a program can be synthesized for solving it. An example of an axiom with a subtask is as follows:

[x -> y], a –> b {P}

where *x -> y* is a subtask*, a is* an input variable and *b* is an output variable of a function described by the axiom that specifies the computability of a method *P*.

Metaclass.
Metaclass is a Java class supplied with a specification. The specification is included in the source of the class as a comment between /*@ and @*/. The specification, called also metainterface, describes how the methods of the class can be used in computations. The following is an example of a metaclass Or where the logical truth values are represented by integers *0* and *1*:

```
class Or {
     /*@ specification Or {
               int in1, in2, out;
               in1, in2 -> out{calc};
          }
     @*/
     int calc (int in1, int in2) {
          return Math.max(in1, in2);
     }
}
```

Scheme.
Scheme is a visual representation of a model. It can always be translated into a specification. A scheme is used also as an environment of communication with an application – for controlling its execution and displaying results.

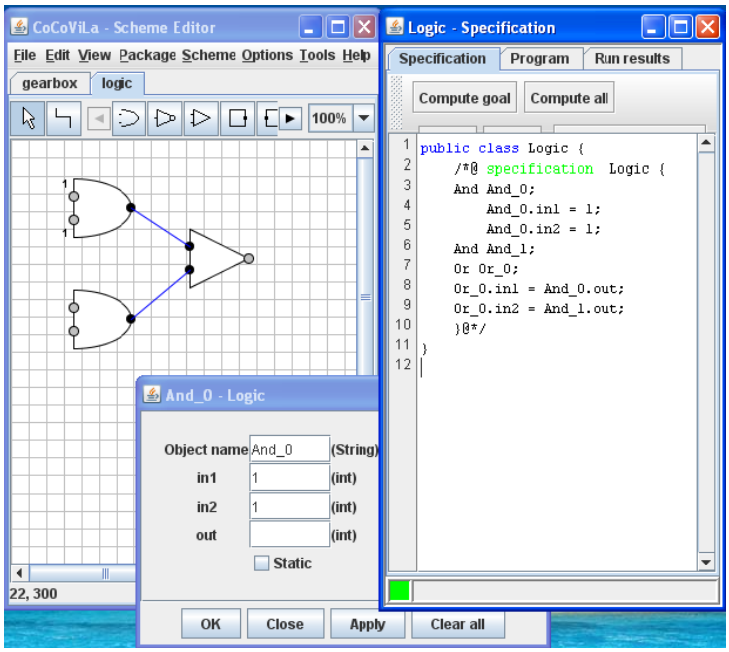

Figure 5. Scheme and specification

A scheme is built by composing it visually from images of components and binding the images by means of connection lines. A connection line binds two ports of images and specifies an equality between the variables represented by the ports. Fig. 5 shows a scheme and a specification derived from it. It also shows a pop-up window of the component `And_0` where two values of two variables are given:

```
in1=1
in2=1
```

These values are visible in the scheme and they are represented in the specification as

```
And_0.in1=1
And_0.in2=1
```

Package

Package is a collection of rich components and schemes related to an application domain, collected in a package folder and supplied with a package description file in xml format.