Tallinn University of Technology
Institute of Cybernetics

Attribute Semantics of Visual Languages

Master thesis

Pavel Grigorenko

Supervisor:
Professor Enn Tõugu

A thesis submitted in conformity with the requirements for the degree of
Master of Science.
Tallinn 2006

Abstract

Visual specification of software is gaining popularity, however its usage is restricted by the lack of precise semantics of visual languages. In this thesis a method for representing the semantics of visual languages by means of attribute models is introduced. Attribute models of a wide class of visual scheme languages are defined, higher-order attribute models that allow synthesizing recursive, branching or cyclic programs are explained. Dynamic evaluation of attributes is used. Three kinds of deep semantics of schemes are presented as different ways of usage of attribute models of schemes. The implementation of attribute semantics of visual languages is done in the system CoCoViLa. The features of specification language of attribute models of schemes and visual classes, as well as the realization of attribute evaluation technique are presented.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and goal

Visual specification of software is gaining popularity, but its usage is restricted by the lack of precise semantics of visual languages. In the domain of programming languages, implementation of semantics is supported by attribute grammars that enable one to program a stepwise transformation of a source text into the code. Attribute grammars have also been used for the representation of semantics of visual specifications in some specific applications [2], [20]. Our aim is to give a possibly general way to implement *deep semantics* of visual languages, i.e. to give tools that enable one to program in a systematic and sufficiently simple way transformations that automatically produce the meaning of a visually represented artifact. *For this purpose we generalize attribute semantics and apply it to a well-defined class of visual languages.*

We consider here visual languages that are not restricted to any specific domain, but still have a well-defined syntactic structure. Actually, we have to restrict us to the languages where a visual specification has a definite structure that can be represented as a graph. One could call such languages *scheme languages*. Then a sentence in a language of this kind is a *scheme*. Languages of schemes are often used in engineering domains, e.g. schemes of electrical, logical, mechanical etc. devices. Generally, considering a structure of a visual description of some artifact, one comes always to a scheme representing components and their relations (connections), although this scheme is not always explicit. Also considering the process of composing a picture on a screen of the computer, we always see that the picture is composed of elements of various types related to each other in some definite ways, often positionally. Even an unordered collection of icons on a desktop is a scheme in our meaning.

1.2 Context of the work

The present work is a part of the research project on visual languages that has resulted in the programming environment - CoCoViLa [9], [10]. This project has been supported by the grant no. 5800 of the Estonian Science Foundation. The first version of CoCoViLa has been implemented by Ando Saabas [30]. The present work relies on the program synthesis technique called *structural synthesis of programs* (*SSP*) that has been the subject of research in the Institute of Cybernetics for many years.

1.3 Organization of the thesis

The thesis is organized in the following way: in Chapter 2 we show different representations of schemes and their basic features. Chapter 3 gives an overview of conventional attribute grammars and introduces the notion of attribute models as well as attribute evaluation techniques on such models. In Chapter 4 we define syntax, shallow and deep semantics of schemes and in Chapter 5 we present the realization of semantics in CoCoViLa. Experiments with the visual languages implemented in CoCoViLa are described in Chapter 6. In Chapter 7 we bring an overview of the related work and finally make conclusions in Chapter 8.

Chapter 2

Schemes

2.1 What is a scheme?

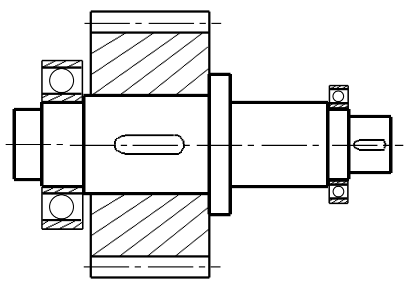
Scheme is visually a set of objects connected with each other either directly or using special connection lines. We allow implicit (invisible) relations in a scheme.

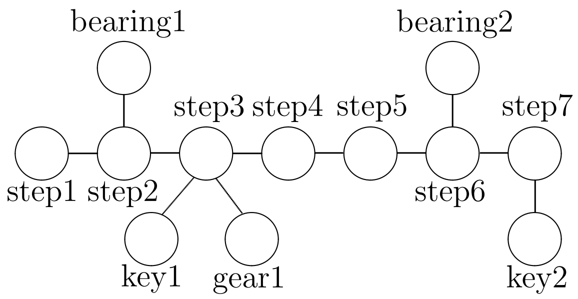
Scheme is a visual representation of an artifact of a particular domain. The meaning of a scheme for different people can be vary depending on their knowledge and belonging to the particular domain to which the scheme relates. On the one hand an ordinary user may consider a scheme as a plain drawing or just a set of objects with some relations between, on the other hand an expert may see some deeper implicit meaning denoting a computational problem in a scheme. Figure 2.1(a) shows an artifact – assembly of a shaft. Mathematically, a scheme can be represented as a graph structure where nodes are objects and edges are connections between objects. Figure 2.1(b) presents a graph that corresponds to the assembly of a shaft.

2.2 Basic features of schemes

Figure 2.2 shows basic features of a scheme language:

- visual identification of type of a component by shape of image,


(a) Drawing



(b) Graph representation

Figure 2.1: Assembly of a shaft

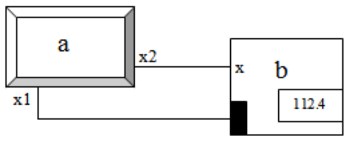


Figure 2.2: A scheme

- connecting particular ports or variables $(x1, x2, x)$ of components,
- introducing names of components (a, b),
- assigning value (112.4) to a field of a component.

Some names are implicit, for example, the variable v behind the given value 112.4 and the name z of the port denoted by the dark element of the component b. The specification of this scheme in a textual language can be, for instance, as follows:

```
ClassP a;
ClassQ b;
a.x1 = b.z;
a.x2 = b.x;
b.v = 112.4;
```

This specification includes declarations of components, equalities that bind variables belonging to components and an assignment of value to a variable. (Values can appear also implicitly, for instance, as coordinates of components in a scheme.)

Here we have briefly introduced schemes as well as their features that we are intrested in.

Chapter 3

Attribute Semantics

Attribute grammars (originally introduced by Knuth [15] as an extension to context-free grammars) have become very popular formal notation for expressing the static semantics of programming languages. In attribute grammars, attribute dependencies (semantic rules) are bound to productions of the syntactic part of a grammar. In our approach we have no productions, and in this chapter we are going to define attribute models that can be directly bound to objects and schemes in order to represent their semantics.

In order to give a better background for a reader not familiar with attribute techniques, in the beginning of this chapter we give basic definitions related to attribute grammars.

3.1 Attribute grammars

A canonical attribute grammar (*AG*) consists of a context-free grammar (*CFG*) G extended with a finite set of attributes A for the non-terminals and a finite set of semantic rules R for the productions:

$$AG = \langle G, A, R \rangle.$$

A context-free grammar

$$G = \langle V, N, P, S \rangle$$

consists of an finite alphabet V of terminal and nonterminal symbols represented by sets Σ and N respectively ($V = \Sigma \cup N$, $\Sigma \cap N = \emptyset$), a set P of productions and $S \in N$ as the starting symbol of grammar G. The productions in P are denoted by $X_0 \rightarrow X_1, ..., X_n$, where $X_0 \in N$ and $X_i \in V,\ i = 1, ..., n$, i.e. left-hand side X_0 is a nonterminal and the right-hand side $X_1, ..., X_n$ is a string of nonterminal and terminal symbols. In the case where $n = 0$, the right-hand side is empty and is represented by the symbol ϵ. The *language* generated by G, denoted by $L(G)$, is the set of sequences of terminal symbols that can be derived by rewriting the start symbol S.

A finite set of attribute $A(X)$ is associated with each symbol $X \in V$. The attributes are characterized as synthesized $S(X)$ or inherited $I(X)$, depending on if they are used to pass the information upward or downward in the syntax tree. $S(X)$ and $I(X)$ are two disjoint sets, $A(X) = S(X) \cup I(X)$. The total set of attributes in grammar is $A = \bigcup A(X)$.

Given a production $p \in P,\ p : X_0 \rightarrow X_1, ..., X_n,$ a semantic rule is written $\alpha_0 = f(\alpha_1, ...\alpha_m)$ and defines α_0 as the value of applying the semantic function f to the attributes $\alpha_1, ...\alpha_m$ of $X_0, ..., X_n$. R_p is a finite set of semantic rules associated with the production p. The attribute α_0 must be either a synthesized attribute of X_0 or an inherited attribute of $X_i,\ i = 1, ..., n$. I.e., a semantic rule defines either a synthesized attribute of the left-hand symbol of the production, or an inherited attribute of one of the symbols on the right hand side of the production. A semantic rule is local and depends only on information available in the attributes of the symbols of the production.

A context-free grammar assigns an abstract syntax tree to every sentence of the language. The semantic rules of an attribute grammar assign attributes to the nodes of the syntax tree. The resulting tree is called attributed tree or undecorated tree. An attributed tree T is defined as follows: to every node N of T that is an instance of nonterminal symbol X, attribute instances are assigned. These instances correspond to the attributes of X. For each attribute $a \in A(X)$ the corresponding instance is denoted by $N.a$.

Attribute evaluation, also called tree decoration, is the process that computes values of attribute instances within an attributed tree T according to the semantic rules of the underlying AG. A program that performs attribute evaluation is called *attribute evaluator*. A decorated tree is an attributed tree in which all attribute instances have a value that was computed according to the attribution rules of the grammar. The meaning of a sentence s of the language generated by G consists of the values of the synthesized attribute instances associated with the root node of the decorated attributed tree, assigned to s. In other words, the meaning is given by the function from inherited to synthesized attributes at the root of the tree.

For further details on attribute grammars and attribute evaluation methods, refer to [26], [29], [31], [45].

There are many works on evaluation of attributes of AGs that use the fact that syntax of a language gives some structure to a set of semantic rules associated with any attributed tree. In the case of scheme languages this structure is weak and practically difficult to use. That is why we are using the notion of attribute models and dynamic attribute evaluation that we will discuss in the following sections.

3.2 Attribute models

In this section we give definitions for attributes, attribute dependencies and attribute models.

Definition 3.1. *Attribute is a variable with a type. It is denoted by a pair* $\langle n, t\rangle$ *where* n *is the name of an attribute and* t *is its type.*

Definition 3.2. *Attribute dependency is a relation between attributes that is represented by one or several functional dependencies whose inputs and outputs are attributes bound by this relation.*

Let us introduce two notations for functional dependency

$$(y_1, ..., y_n) = f(x_1, ..., x_m)$$

where f is a function of m arguments computing a value of n-tuple. The variables $x_1, ..., x_m$ are inputs and $y_1, ..., y_n$ are outputs of the functional dependency. When we present only the type of the dependency, then we denote it by

$$x_1, ..., x_m \to y_1, ..., y_n \ ,$$

which means for us that having $x_1, ..., x_m$ we can compute $y_1, ..., y_n$. When we present also the implementation f of the dependency, then we denote it by

$$x_1, ..., x_m \to y_1, ..., y_n\{f\} \ .$$

We assume here that attribute dependencies can be presented by equalities, structural relations, equations and preprogrammed procedures (methods of a class).

1. Equality $x = y$ can be rewritten as $x \to y; \ y \to x$.
2. Structural relation binding a tuple of variables $x_1, ..., x_m$ with a structured variable $x = (x_1, ..., x_m)$ can be presented as $x_1, ..., x_m \to x; \ x \to x_1, ..., x_m$.
3. Equation, i.e. $x = y + z$ can be presented as a collection of functional dependencies, in the given example as $y, z \to x; \ x, y \to z; \ x, z \to y$.
4. Preprogrammed procedure with attributes $x_1, ..., x_m$ as parameters producing a value of attribute y can be presented as $x_1, ..., x_m \to y$.

Definition 3.3. *An attribute model M is a pair $\langle A, R\rangle$, where A is a finite set of attributes and R is a finite set of attribute dependences binding these attributes.*

Attribute models $M' = \langle A', R'\rangle$ and $M'' = \langle A'', R''\rangle$ can be composed into a new attribute model binding some of their attributes by equalities. Let us have a set of equalities $s = \{M'.a = M''.b, ... \ ..., M'.d = M''.e\}$ binding some attributes of models M' and M''. By $\cup_s(M', M'')$ we denote an attribute model with the set of attributes $A' \cup A''$ and the set of attribute dependencies $R' \cup R'' \cup s$. Let us generalize the composition of attribute models for more than two models as follows.

Definition 3.4. *For a set of equalities s that bind some attributes of models $M_1, ..., M_n$ we denote by $\cup_s(M_1, ..., M_n)$ an attribute model with the set of attributes $A_1 \cup ... \cup A_n$ and the set of attribute dependencies $R_1 \cup ... \cup R_n \cup s$, and call it composition of $M_1, ..., M_n$ with bindings s.*

Remark 3.1. *When building a composition of attribute models, renaming of attributes may be required. A straightforward way to do it is to add the name of a model where an attribute came from to its name. This introduces composite names, e.g. $m.x, m.y$ for attributes x, y of an original model m.*

Remark 3.2. *Any attribute model can be presented in a flattened form where all attribute dependencies are functional dependencies. In order to do this, one has to consider relations between attributes as sets of functional dependencies and take their union for the set of attribute dependencies of the attribute model in the flattened form.*

A simple undirected graph $G = (V, E)$ is called bipartite if there exists a partition of the node set $V = V_1 \cup V_2$ so that both V_1 and V_2 are independent sets. $G = (V_1 + V_2, E)$ denotes a bipartite graph with partitions V_1 and V_2.

Remark 3.3. *An attribute model $\langle A, R\rangle$ can be presented as a bipartite graph with partition of the set of nodes $A \cup R$. There is an edge (a, r) in the graph if and only if the attribute a is bound by the attribute dependency r.*

Remark 3.4. *An attribute model $\langle A, R\rangle$ in the flattened form can be presented as a directed bipartite graph with sets of nodes A and R. There is an arc from a to r, if and only if a is an input of r and an arc from r to a, if and only if a is an output of r.*

3.3 Computational problems on attribute models

Let U and V be two sets of attributes of an attribute model M. We call a pair $\langle U, V\rangle$ a computational problem on the attribute model M, where U is a set of input attributes (or just *input*) and V is a set of output attributes (or *output*) of a computational problem. The meaning of a computational problem $\langle U, V\rangle$ is that given values of attributes from U find values of attributes of V using attribute dependences of M.

If for two computational problems $\langle U_1, V_1\rangle$ and $\langle U_2, V_2\rangle$ we have $U_1 \subset U_2$ or $V_2 \subset V_1$ then we say that the computational problem $\langle U_1, V_1\rangle$ is larger than the computational problem $\langle U_2, V_2\rangle$, and denote this $\langle U_2, V_2\rangle < \langle U_1, V_1\rangle$.

3.4 Evaluation of attributes

We show that there is a procedure that for any computational problem $\langle U, V\rangle$ on a attribute model $\langle A, R\rangle$ in the flattened form decides whether there is a way to compute values of attributes of V from given values of attributes of U, and in the case of the positive answer produces an algorithm for computing the values, i.e. produces an algorithm for solving the computational problem.

Definition 3.5. *Value propagation is a procedure that for an attribute model M in flattened form and a set of attributes U that belong to this model decides which attributes are computable from U and produces a sequence of functional dependences that is an algorithm for computing values of these attributes.*

A simple value propagation algorithm works step by step as follows. At each step it checks for each functional dependency whether its inputs are all computed and some of its outputs is not computed. In the positive case, the functional dependency will be added to the algorithm being built and all outputs of the functional dependency will be added to the set of computed attributes. Initially the set of computed attributes equals to the set of given attributes U and algorithm (i.e. the sequence of functional dependencies) is empty.

Remark 3.5. *There are very efficient algorithms for value propagation that work in linear time with respect to the size of attribute model, see [40].*

Remark 3.6. *Value propagation is a procedure that decides for a computational problem whether it is solvable, and in the case of positive answer gives an algorithm of solving the problem.*

Remark 3.7. *There is a procedure that gives a minimal algorithm for solving a computational problem, see* [13].

Remark 3.8. *Set of all solvable computational problems on an attribute model is well defined, hence we can decide that an attribute model is an attribute dependency with algorithms for solving computational problems as its set of functional dependencies.*

3.5 Higher-order attribute models

Unfortunately, attribute models as defined above are not very expressive. One can compose only linear sequences of functional dependencies for solving computational problems on them. We are going to define now more expressive attribute models that enable one to synthesize more complex algorithms.

Let A be a set of attributes and P a set of computational problems with input and output attributes from A.

Definition 3.6. *Higher-order functional dependency (hofd) is a functional dependency that has inputs from $A \cup P$ and outputs from A. Inputs from P are called subtasks.*

Definition 3.7. *Higher-order attribute model is a pair $\langle A, R\rangle$ where A is a set of attributes and R is a set of attribute dependencies that include some higher-order functional dependencies on the set of attributes A.*

Considering types of functional dependencies, one can see that besides functional types like

$$x, y \to z$$

we have now also types like

$$(u \to v), x \to y,$$

where $u \to v$ is a subtask. Its value is a function (represented by an algorithm) that computes v from u. This function must be given, in order to use the higher-order functional dependency. Sometimes this function can be synthesized on an attribute model that includes the higher-order functional dependency, and only in such a case the higher-order functional dependency can be used in computations.

This extension makes a big difference in the following: Higher-order attribute models are so expressive that enable one to synthesize recursive, branching and cyclic programs. In order to do this one has to present some control structures, e.g. loops and conditional branching as higher-order functional dependencies. Detecting solvability of a problem and synthesizing an algorithm on a higher-order attribute model has exponential time complexity [23].

A simple example of a *hofd* with a subtask is "compute the sum z of b_a for $a = 1, ..., x$ knowing the rule for computing b for any a" expressed conventionally by the following formula:

$$z = \sum_{a=1}^{x} b_a$$

We express this as

$$(a \rightarrow b), x \rightarrow z.$$

To have a more concrete example, let us take a problem of computing a double integral under the assumption that an integration method is implemented as a *hofd*. The attribute dependency is for computing value of integral of any function $f(x)$ specified on the interval $[0, \mathit{to}]$:

$$val = \int_0^{to} f(x)\,dx.$$

This is specified by

$$(arg \rightarrow res), to \rightarrow val.$$

where *arg* stands for argument x, and res stands for the value of function f. (Both *to* and f should be computed before the *hofd* can be applied.

The subtask here is analogous to the one in axiom of the formula for sum presented above. Now we can specify the double integral problem:

$$y = \int_0^x \int_0^{f2(v)} f1(u, v)\, dudv.$$

The attribute model of double integration uses two *hofd*s, and, for instance, after introducing attributes for intermediate values $fVal$, $integral1$, $upb1$ will include the following four attribute dependencies:

$$(u \to fVal), upb1 \to integral1$$
$$(v \to integral1), x \to y$$
$$u, v \to fVal$$
$$v \to upb1$$

where $fVal = f1(u, v)$, $integral1 = \int_0^{upb1} f1(u, v)\, du$ and $upb1 = f2(v)$.

As a yet another example, let us consider more attribute models with two *hofd*s. This is a rather typical case that covers computing double sums and products:

$$\sum\sum a_{ij} \qquad \prod\prod a_{ij} \qquad \sum\prod a_{ij} \qquad \prod\sum a_{ij}$$

The attribute model for double sum

$$z = \sum_{i=1}^{x}\sum_{j=1}^{y} a_{ij}$$

has the following attribute dependences:

$$(j \rightarrow val), y \rightarrow sum1$$
$$(i \rightarrow sum1), x \rightarrow z$$
$$i, j \rightarrow val$$

where attributes for intermediate values val and $sum1$ correspond to a_{ij} and $\sum_j^y a_{ij}$ respectively.

3.6 Evaluation of higher-order attributes

Let us consider first the evaluation of attributes of an attribute model that contains only one higher-order functional dependency with a single subtask. Then the following steps are performed.

First the procedure of simple value propagation is done using only attribute dependencies that are not higher-order. If this does not solve the problem (does not give values of all outputs of the problem), then the *hofd* is applied, if it is applicable. A *hofd* is applicable if and only if all its inputs are given and all its subtasks are solvable and it computes values of some attributes that have not been evaluated yet. A sequence of applicable functional dependencies obtained in this way is called maximal linear branch (*mlb*). It contains one *hofd* at the end of the sequence. There are two possible outcomes of this procedure:

1. A *mlb* cannot be found and the problem is unsolvable.
2. The constructed *mlb* reduces the problem to a simpler one. (If a *mlb* exists, then it obviously reduces the problem, because it computes some new values of attributes.)

Now the simple value propagation (using only attribute dependencies that are not higher-order) is done again. The problem is either solved in this way, or it is unsolvable, because no more possibilities for computing new values exist. It is easy to see that the algorithm above works in linear time.

If there is more than one higher-order dependency in an attribute model, then there are several options for choosing a *hofd*, and the planning algorithm cannot choose any appropriate *hofd* for solving the computational problem without deeper analysis. Then the exhaustive search on *and-or tree* is required. There are two basic strategies of construction of *and-or* trees for solving subtasks. The first one does not allow to use *hofd*s that have already been used in some *mlb*. Having this restriction we reduce the number of branches in the search tree to the number of all possible permutations of *hofd*s. The second strategy allows repetitions and then the algorithm is PSPACE-complete [25].

In a general case, when an attribute model M contains several higher-order functional dependences, the evaluation strategy is as follows.

First the procedure of simple value propagation is performed considering all attribute dependencies that are not higher-order. If this does not solve the problem (does not give values of all outputs of the problem), then any applicable *hofd* is chosen and a maximal linear branch is obtained. There are three possible outcomes of this procedure:

1. After constructing the *mlb* the problem is solvable (like in the case of a single *hofd*).

2. A *mlb* cannot be found and the problem is unsolvable.

3. A *mlb* can be found and the initial problem $\langle U_1, V_1 \rangle$ is reduced to a simpler one $\langle U_2, V_2 \rangle$, $U_2 = U_1 \cup Y$ and $V_2 = V_1 \backslash Y$, i.e. $\langle U_2, V_2 \rangle < \langle U_1, V_1 \rangle$, where Y is the set of outputs of the *hofd*.

This procedure (construction of *mlb*) is repeatedly applied until the problem is solved or no more *mlb* can be constructed.

It is important to notice that for applying a *hofd* we have to solve all its subtasks. This means that **the whole procedure of problem solving must be applied for every subtask**. As stated above, this requires a search on an *and-or* tree of problems (subtasks) on the attribute model. The root of a tree corresponds to the initial problem, and it is an *or*-node, because there may be several possible *mlb*s for this problem. *And*-nodes correspond to higher-order functional dependencies and have one successor for its each subtask, plus one successor for the reduced task that has to be solved after applying the *mlb*. *Or*-nodes of the tree correspond to the subtasks that have to be solved for their parent *and*-node.

Let us abbreviate $\underline{X}$ for $x_1, ..., x_n$ and S_i for $(u_{i,1}, ..., u_{i,n} \rightarrow v_{i,1}, ..., v_{i,m})$. Then a *hofd* has the form: $S_1, ..., S_m, \underline{X} \rightarrow \underline{Y}$, where $\underline{X} = x_1, ..., x_k$, $\underline{Y} = y_1, ..., y_n$, $x_i, y_j \in A$ are attributes and $S_k \in P$ are subtasks. Let us label each functional dependency in the tree with R_i and get the following notation for a *hofd*:

$$R_i : \; S_{i,1}, ..., S_{i,m}, \underline{X} \rightarrow \underline{Y}.$$

Figure 3.1 shows an *and-or* search tree for solving problems on higher-order attribute models. Its root is the original problem S_0. The *and*-nodes are the *hofd*s that can be applied first for solving the problem of their parent node. The successors of a *hofd* node R_i are its subtasks and the reduced problem $S_i^{'}$ that remains to solve after applying the *hofd*. Notice that each *hofd* R_i may appear several times on different levels in one and the same branch of the tree. The search on the *and-or* tree is depth-first search with backtracking.

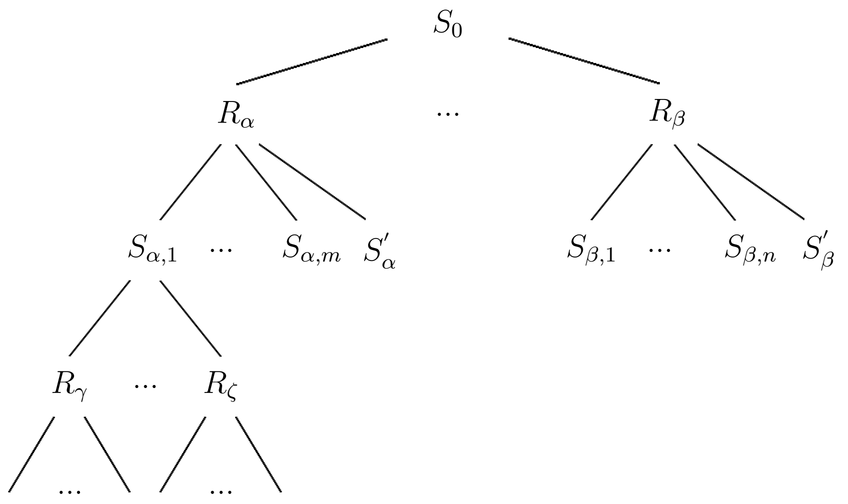

S_0
R_α
...
R_β
$S_{\alpha,1}$ ... $S_{\alpha,m}$ $S_\alpha^{'}$
$S_{\beta,1}$ ... $S_{\beta,n}$ $S_\beta^{'}$
R_γ ... R_ζ
...
...

Figure 3.1: *And-or* search tree for attribute evaluation on higher-order attribute model

Chapter 4

Syntax and Semantics of Schemes

In this chapter we define formal syntax of schemes in a very nonrestrictive way and introduce semantics of schemes using attribute models and evaluations of attributes on them as described in Chapter 3.

4.1 Syntax

Let us have a finite number of types. A *node type* is an expression $t(a, ..., b)$, where t is a type and $a, ..., b$ are typed variables, i.e. pairs of the form (x, t'), where x is a variable and t' is its type. The variables $a, ..., b$ are called ports of the node type $t(a, ..., b)$. A scheme is a set of pairs (u, t) called *nodes*, where u is a name (identifier) and t is a node type, and a set of equalities $u.a = v.b$ called *bindings*, where a, b are ports of one and the same type of nodes u, v respectively, and a set of valuations $u.a = v$, where v is a value of a suitable type.

A scheme as defined above can be presented by a text in a very simple language that has three kinds of statements:

1. declaration of a component:

   ```
   <type> <identifier>
   ```

This declaration specifies a component of a scheme with given type, and its name given by identifier.

2. binding:

```
<name of component>.<name of port>
      = <name of component >.<name of port>
```

This statement specifies an equality between variables of components. These variables are also attributes of attribute models of components.

3. valuation:

```
<name of component>.<name of port> = <value>
```

This statement defines a functional dependency with no inputs and one output that gets a constant value.

4.2 Shallow semantics

When we consider a particular problem-domain, a scheme representing something in this domain typically has what one could call the *real* semantics, the meaning of the scheme that a specialist of that domain recognizes and understands. The obvious aim is to represent this knowledge (at least to some extent) in a computer. For this, we have to reason about the semantics of schemes on several different levels. Firstly, one can consider the shallow semantics of a scheme - the textual representation of the graph underlying the scheme. On this level, all deeper information about the scheme is disregarded. To be more precise - it is hidden in the types (classes) of objects.

Definition 4.1. *The shallow semantics of schemes is a function SS, which, for each scheme G, returns a string including exactly the following:*

- *$Type$ obj; whenever G includes an object named obj of type $Type$.*

- *$obj.x = v$; whenever an attribute x of object obj has the value v in scheme G.*

- *$obj1.x = obj2.y$; whenever there is an edge between ports x and y, and these ports are associated with objects $obj1$ and $obj2$, respectively.*

4.3 Deep semantics

Definition 4.2. *Attribute model of a scheme is the composition of attribute models of its nodes bound by the equalities of its shallow semantics and including additional functional dependencies for valuations in the scheme. Its attributes, functional dependencies and attribute dependencies are called also attributes, functional dependencies and attribute dependencies of the scheme.*

More formally, if a scheme has nodes $obj1, ..., objn$ with attribute models $M_1, ..., M_n$ respectively, and the equalities of shallow semantics constitute a set $s = \{obji.a = objj.b, ..., objs.d = objk.e\}$, and valuations are $objq.x = v1, ..., objp.y = vl$, then the attribute model is $\cup s(M_1, ..., M_n) \cup \{objq.x = v1, ..., objp.y = vl\}$.

We know that there is a procedure that for any computational problem $\langle U, V\rangle$ on a scheme decides whether there is a way to compute values of attributes of V from given values of attributes of U, and in the case of the positive answer produces an algorithm for computing the values, i.e. produces an algorithm for solving the computational problem, see attribute evaluation in Chapter 3. Let us denote by $S1$ the function producing an algorithm for any solvable computational problem on a scheme.

Definition 4.3. *The deep semantics $DS1$ of schemes is a composition of the shallow semantic function SS and of the semantic function $S1$ that defines computability on a scheme.*

Let us denote by $S2$ the function that produces an algorithm for solving the largest solvable computational problem with empty set of input attributes on the scheme.

Definition 4.4. *The deep semantics $DS2$ of schemes is a composition of the shallow semantic function SS and of the semantic function $S2$ which defines largest computability on a scheme.*

The third semantic function is defined as an attribute evaluator of an attribute grammar [27]. Up to now we have silently considered attributes as meaningful variables of a problem domain. However, in the most general case attributes can be just variables bearing some semantic information. The real meaning of a scheme can be computed as a value of a distinguished attribute, let us call it a *scheme attribute* on the attribute model of the scheme. We denote by $S3$ the function that, given a scheme, computes its scheme attribute's value. The scheme attribute corresponds to the synthesized attribute of a nonterminal symbol representing the whole program in the conventional case of attribute grammars of programming languages, and in our case it represents the meaning of the whole scheme. The only essential difference between our approach and the conventional dynamic attribute evaluation is that we have to use a more sophisticated attribute evaluator, because instead of an abstract syntax tree we have a graph representing a scheme that in a general case is not a tree, and we accept higher-order attributes as well.

Definition 4.5. *The deep semantics $DS3$ of schemes is a composition of the shallow semantic function SS and of the semantic function $S3$ that computes the value of a scheme attribute.*

We have to notice that implementing $DS3$ is a much harder task than implementing $DS1$ and $DS2$. It is really a compiler-writing task for a visual language, whereas the latter two semantic functions are implemented easily by specifying domain-oriented functional dependencies.

Chapter 5

Implementation

Our approach to attribute semantics of schemes has been implemented in a tool CoCoViLa. The initial version has been presented by Ando Saabas [30] for simple functional dependences (without higher-order), the present chapter reflects improvements and additions that have been made to the framework.

This tool is used for development and prototyping of domain-specific visual languages in Java environment. A visual language is implemented as a package that is a set of components called *visual classes* that are Java classes annotated with *metainterfaces* and supplied with corresponding *visual images*. Deep semantics of schemes is implemented in such a way that for scheme's attribute evaluation, executable code is produced.

5.1 Metaclasses and metainterfaces

Java is the object-oriented programming language, where classes describe properties and the behavior of an object. In CoCoViLa, in order to describe the behavior of a component of a scheme, one has to annotate component's Java class with a *metainterface*. Java class supplied with metainterface is called *metaclass*. Metainterface is a textual presentation of attribute model of the component, i.e. all attributes as well as attribute dependences are specified within a corresponding metainterface.

The actual presentation of a metainterface is the specification in a form of Java comment, thus it is visible to CoCoViLa but ignored by Java compiler.

```
class VisualClass {
   /*@ specification VisualClass {
       //...
   }@*/
}
```

5.1.1 Specification language of metainterfaces

The specification language of metainterfaces consists of textual representation of attribute dependencies (defined in section 3.2), statements from the textual language of schemes (section 4.1) as well as some additional useful extensions. Initially, it has been presented in [30]. In this section we are going to describe briefly all features of the language, and are going to concentrate on axioms and their usage, as well as an inheritance, wildcards and aliases that have required considerable amount of work from the author. Formally, the syntax is written as a context-free grammar in BNF.

```
MetaInterface ::= ‘specification’ ClassName
                      [InheritanceDecl] ‘{’
                                  Specification ‘}’

Specification ::= (VariableDecl | Constant | Binding | Axiom
                      | Alias | Equation) ‘;’ [Specification]
```

The `ClassName` is the name of a metaclass where the metainterface is being declared.

Metainterface variables

Variables correspond to attributes of an attribute model behind the metainterface. The declaration follows the Java syntax.

```
VariableDecl ::= Type IdentList
IdentList ::= Identifier [‘,’ IdentList]
```

Identifier is the name of the declared variable. Type can be either

- a primitive type of the Java programming language or ‘`void`’,
- a class in the Java programming language, or
- a name of other metaclass declared in a visual language;

Constants

The declaration of a constant has the form:

```
ConstantDeclaration ::= ‘const’ Identifier ‘=’ value
```

Where `value` refers to a constant value in the Java programming language, e.g. `Math.PI`.

Bindings

Binding is an equality relation between variables used to show that the realization of the left-hand side variable is the same as the realization of the right-hand side variable. The specification of the binding has the form

```
Binding ::= Variable ‘=’ Variable
```

Where `Variable` can be in the form of

```
Variable ::= Identifier | Variable‘.’Identifier
```

This definition uses hierarchical structure of variables and compound names. The specification `a.x = b.y` means that the realization of the variable `a.x` is the same as the realization of the variable `b.y`. In an attribute model relations $x = y$ and $y = x$ are semantically equivalent because they both can be rewritten as $x \to y;\ y \to x$.

Axioms

Axioms, written in the form of implications, correspond to functional dependencies that also have implementations. On the left-hand side of the implication input attributes are listed, we call them *preconditions*, while on the right-hand side are the output attributes, or *postconditions* of an axiom. If all preconditions of an axiom are derivable, i.e. their values are computable, then the implementation of this axiom can be applied for attribute evaluation. The implementation of an axiom, specified in curly brackets, is the Java method defined in the metaclass of the component. When the axiom is applied, its input variables are passed to the implementation as parameters. Disjunction "|" symbol on the right-hand side is used for separating possible alternative outputs, for instance, exceptions, of the applied method.

```
Axiom ::= [VariableList [‘,’ SubTaskList]] ‘->’
           Variable [‘|’ ExceptionList] ‘{’ Realization ‘}’
ExceptionList  ::= ‘(’ ExceptionClass ‘)’ [‘,’ ExceptionList]
ExceptionClass ::= Type
```

Example. Here we show the metaclass `Foo` where its metainterface states that the `area` can be computed using `radius` as the input parameter of the method `calcArea`, which represents the implementation of the given axiom.

```
class Foo {
   /*@ specification Foo {
       double radius;
       double area;
       radius -> area {calcArea};
   }@*/

   double calcArea(double r) {
       return Math.PI*Math.pow(r, 2);
   }
}
```

From the example above it is important to notice that the specification variables are not variables of a Java class and we can see also how metainterfaces interact with Java methods (via axioms).

In addition, we have put the restriction that the right-hand side of an axiom cannot be empty. If a method's return type is `void`, a *control variable* should be used (see next section). This is necessary because each application of an axiom should add new evaluated variables into the algorithm, otherwise such axiom will be never applied.

Example. In this example we present how to specify exceptions in the metainterface.

Consider the metaclass `Matrix`:

```
class Matrix {
   /*@ specification Matrix {
       int row, col, element;
       int[][] m; //matrix as array

       row, col, m -> element
                     |(ArrayIndexOutOfBoundsException)
                                {getElement};
   }@*/

   int getElement( int i, int j, int[][] matrix )
                throws ArrayIndexOutOfBoundsException {
       return matrix[i][j];
   }
}
```

Here the method `getElement()` is used to evaluate the variable `element`. The provided method may throw the exception – `ArrayIndexOutOfBoundsException` – and to be able to handle this situation (e.g. surround the call of a method with a corresponding `try-catch` statement during automatic code generation, see 5.5.2), exceptions should be specified as postconditions of an axiom, separated from variables with the "|" symbol.

Control variables

Control variables do not have a computational meaning, though if such variable is used as one of the preconditions in the axiom, it should be derivable. In the metainterface, a control variable should be declared with type `void`.

```
ControlVariableDeclaration ::= ‘void’ ControlVariableList
ControlVariableList ::= Identifier [‘,’ ControlVariableList]
```

Example.

```
void ready;
String path;
int status;
path -> ready { init };           (1)
ready -> status { getStatus };    (2)
```

The purpose of the given example is to show how the control variable allows to control the execution priority of methods. Method `getStatus` should be executed after the method `init`, which returns `void`. If the axiom (1) had no postcondition, this would lead to a problem where we would be unable to guide the evaluation process. Control variables help solving this problem. When the axiom (1) is applied, `ready` becomes derivable. Only then it is possible to use `ready` as a precondition of (2) to apply the axiom.

Assumptions and goals

The computational problem can be defined using an axiom without implementation, where preconditions are the inputs and postcondition are the outputs of the computational problem. The output attributes of a problem are called *goals*, whereas input attributes – *assumptions*. In this case the system tries to generate an attribute evaluator that will compute goals from assumptions and remove all unnecessary (excessive) relations from the evaluation algorithm.

```
Goal ::= [Assumptions] ‘->’ Goals
Assumptions ::= Variable [‘,’ Assumptions]
Goals ::= Variable [‘,’ Goals]
```

Example. Let us have the specification as follows:

```
double a,b,c,d,e;
c = a^2;            (1)
e = 3;              (2)
b = e / c;          (3)
d = sin( c );       (4)
a -> d;             (5)
```

The line (5) denotes the goal `d` that should be computed from the assumption `a`. The evaluation algorithm could be the sequence of lines (1) and (4), assuming that `a` has already been evaluated and ignoring all other relations (i.e. (2) and (3)) that are not required for the computation of goal `d`.

Equations

Equations are used to get rid of excessive axioms in the metainterface, implementation of which could have been arithmetic expressions.

```
Equation ::== Expression ‘=’ Expression
Expression  ::=  [ ‘-’ ] Term [ ( ‘+’ | ‘-’ ) Term ]
Term  ::=  Factor [ ( ‘*’ | ‘/’ ) Factor ]
Factor  ::=  Primary [ ‘^’ Primary ]
Primary  ::=  Number | Variable |‘(’ Expression‘)’  |
              Function-name ‘(’ Expression ‘)’
Function-name ::= ‘sin’ | ‘cos’ | ‘tan’ |  ‘log’
```

Example. Instead of writing the following axioms

```
i, r -> u {calcVoltage}
u, r -> i {calcCurrent}
u, i -> r {calcImpedance}
```

with implementations required for calculating the voltage, current and impedance, we can use a single equation `u = i * r`.

Polymorphic types

Variables with type `any` provide an interesting form of polymorphism to the language - they can be used in specifications of equations and axioms before the concrete type is assigned.

Example. Assume we have two components, `Resistor` and `Capacitor`, both contain attributes `i,r` and `u`. And we also have another component `Par` which corresponds to the parallel connection of electrical elements. The metainterface of `Par` can be as follows:

```
/*@ specification Par {
       any x1, x2;
       double i,u,r;
       u = i*r;
       i = x1.i + x2.i;
       1/r = 1/x1.r + 1/x2.r;
       u = x2.u - x1.u;
}@*/
```

It is easy to see that variables `x1` and `x2` can be bound with components `Resistor` or `Capacitor` in a scheme, and the types of `x1` and `x2` can be `Resistor` or `Capacitor` accordingly.

Notice that the concrete type is assigned to a variable with the initial type `any` as soon as it is bound with another variable with a concrete type.

Subtasks

If an axiom contains a precondition in the form of an implication, it correspond to the higher-order functional dependency of an attribute model. This implication is a computational problem with assumptions on the left-hand side and goals on the right-hand side.

```
SubtaskList ::= Subtask [‘,’ SubtaskList]
Subtask ::= ‘[’IdentifierList ‘->’ Identifer ‘]’
```

The actual subtask is represented by a class, instance of which is passed as a functional parameter to the implementation of an axiom.

All subtasks implement a single Java interface

```
public interface Subtask {
    Object[] run(Object[] in) throws Exception;
}
```

where the body of method `run` is automatically generated (synthesized) by the system.

Example.

```
class Foo {
   /*@ specification Foo {
       double a,b,c,d;
       [a->b],c -> d {test};   (*)
   }@*/

   double test( Subtask s, double x ) {
       ...
   }
}
```

In the metaclass above, the meaning of axiom (*) is - having `c` compute `d` if there is a possibility to generate an algorithm for computing `b` from `a`.

We will demonstrate the synthesis and the code generation of subtasks in section 5.5.2.

Aliases

Aliases correspond to structural relations that binds tuples of attributes, i.e. $x = (x_1, ..., x_m)$. The extensive use of this feature can be made when defining visual classes – when more than one variable needs to be bound via a port (see section 5.2).

```
Alias ::= ‘alias’ [‘(’ Type‘)’] Identifier ‘=’
                               ‘(’ ( VariableList | Wildcard ) ‘)’
```

Since [30] this structure has been considerably extended. One of the new features is the type checking, i.e. if an alias is declared with the concrete type defined in `Type`, only variables of that type can be bound by this alias, this is especially useful in automatic binding with *wildcards*, see next section.

Previously, aliases have been used for representing lightweight structures, however now we allow declaring aliases that may contain other aliases. Additionally, if an alias is used in pre- or postconditions of an axiom or subtask, it is regarded as 1) an array of a concrete type if all variables in this alias have the same type, or 2) as array of type `Object[]` if variables have different types.

During the evaluation process, if we need to know how many variables are bound by an alias, we can refer to the variable `<alias_id>.length` that is automatically created after the declaration of an alias.

Example.

```
int a,b,u,v;

alias x = ( a,b );
alias y = ( u,v );

x = y;                (*)
```

This example demonstrates that during generation of an attribute evaluator, the relation (*) will be rewritten into

```
a = u;
b = v;
```

Example. In this example we present the usage of aliases (as array structure) in axioms.

```
class TempAvg {
   /*@ specification TempAvg {
       Temp t1, t2, t3;
       double avg;
       alias (double) tempr = ( t1.t, t2.t, t3.t );
       tempr -> avg {calcAvg};                         (*)

   }@*/

   double calcAvg( double[] tt ) {
       ...
   }
}
```

Assuming that class `Temp` contains variable `t`, metaclass `TempAvg` uses the axiom (*) for calculating average temperature, passing alias `tempr` as parameter to the method `calcAgv`.

Wildcards

From the definition of alias we can observe that instead of a list of variables alias can be declared with a wildcard.

```
Wildcard ::= '*.'Identifier
```

Wildcards are used for the dynamic binding of variables having the same name (equal to Identifier) that are defined in other metainterfaces of components declared on the same level with the wildcard.

Example. The following specification

```
Resistor r1, r2;
Capacitor c1;
Par p;
alias x = ( *.u );
```

is semantically equivalent to the specification

```
Resistor r1, r2;
Capacitor c1;
Par p;
alias x = ( r1.u, r2.u, c1.u, p.u );
```

assuming that each metaclass contains the declaration of variable `u` in the metainterface.

Wildcards are useful when we need to define relations before knowing how many and what components will be used in the scheme.

Inheritance

The system supports inheritance of metaclasses.

```
MetaInterface   ::= ‘specification’ ClassName
                     [InheritanceDecl] ‘{’
                                  Specification ‘}’
InheritanceDecl ::= ‘super’ SuperClassList
SuperClassList  ::= SuperClass [‘,’ SuperClassList]
```

Here we have to deal with to types of inheritances, Java object-oriented inheritance and inheritance of metainterfaces.

Metainterface of class `B` can inherit metainterface of class `A` if and only if `A` is a superclass of `B` in Java, i.e. B is declared as `class B extends A`. The inheritance of metainterfaces means that specifications of superclasses are unfolded into the specification of a subclass. The situation when a variable with the same name is declared multiple times in superclasses and/or in the subclass, is regarded as an error.

Example. Here we show three metaclasses that correspond to components `Point`, `Shape` and `Rectangle`. Shape is an abstract class that contains initial coordinates and a variable `S` for square. Metaclass `Rectangle` is a subclass of `Shape`.

```
class Point {
   /*@ specification Point {
       int x,y;
   }@*/
}

class Shape {
   /*@ specification Shape {
       Point p;  //initial coordinates
       double S; //square
   }@*/
}

class Rectangle extends Shape {
   /*@ specification Rectangle super Shape {
       double a,b;
       Point x2, y2;
       S = a * b;
       x2 = x + a;
       y2 = y + b;
   }@*/
}
```

5.1.2 Semantics of the specification language

The metainterface specification language has a rather straightforward translation into the internal representation in the form of a graph, like an attribute model's textual description has, see section 3.2.

5.2 Visual classes

In CoCoViLa, a visual language is a set of visual classes. A visual class consists of the following parts:

- metaclass
- visual image
- set of ports
- set of fields (attributes that are visible on a scheme or in object's property window)

As an example, we introduce a visual class that corresponds to the Resistor element from the visual language used for simulation of electrical circuits. The visual image of Resistor is shown in Figure 5.1 and its metaclass is presented in Figure 5.2

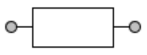
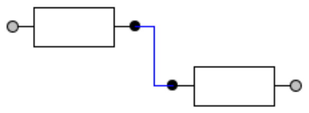


Figure 5.1: Visual image of Resistor

```
class Resistor {
    /*@ specification Resistor {

        double u, u1, u2, r, i;
        u = u2-u1;
        u = i*r;

        alias p1 = (u1, i);
        alias p2 = (u2, i);

    }@*/
}
```

Figure 5.2: Resistor metaclass

We denote instances of visual classes in a scheme as *visual objects* (or just *objects*). In a scheme, instances of visual classes (objects) are connected with each other through ports. Each port of a visual class corresponds to an attribute defined in a metainterface. From the declaration in section 4.1 we know that the binding of ports means an equality of attributes. Visual class `Resistor` contains two ports corresponding to attributes $p1$ and $p2$ of type `alias`. Figure 5.3 (a) shows a scheme with two instances of the class `Resistor` connected together, i.e. port $p2$ of the first `Resistor` is bound with port $p1$ of the second `Resistor`. The text in Figure 5.3 (b) corresponds to the shallow semantics of the given scheme. Figure 5.3 (c) shows the meaning of equality (*), i.e. how it is represented as a set of functional dependencies in the underlying attribute model of the scheme.


(a) Connection of ports

```
Resistor r1;
Resistor r2;

r1.p2 = r2.p1;          (*)
```

(b) Shallow semantics of scheme

```
r1.u2 = r2.u1;
r1.i  = r2.i;
```

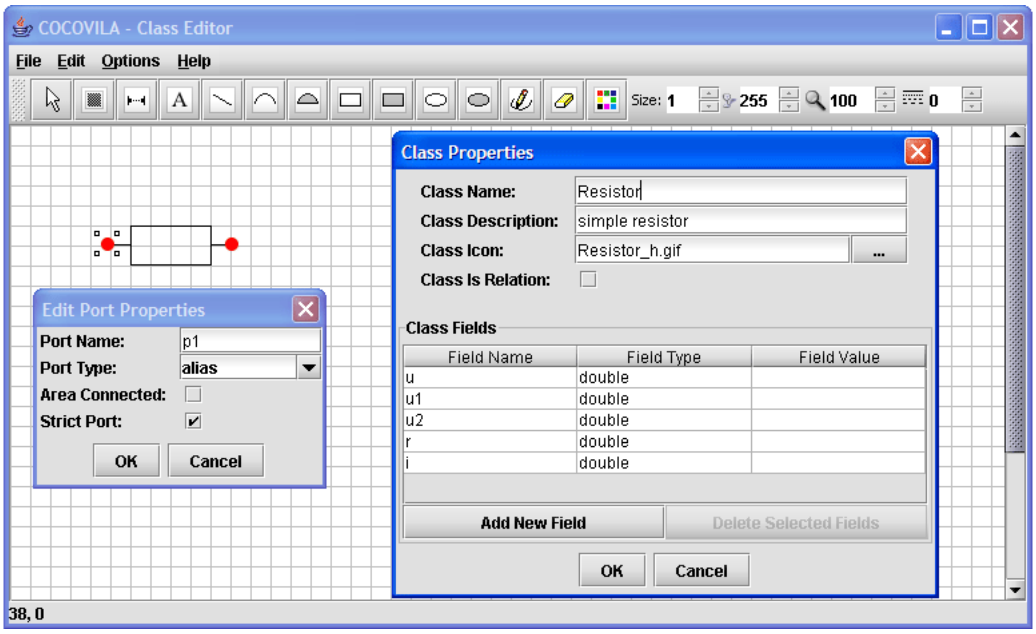
(c) Flattened view of equality (*)

Figure 5.3: A scheme with two Resistors and its textual meaning

5.3 Class Editor

From a users point of view CoCoViLa consists of two components (applications): *Class Editor* and *Scheme Editor*.

The *Class Editor* is used for defining attribute models of components of schemes as well as their visual and interactive aspects. In other words, *Class Editor* is used to map domain concepts to visual classes [9]. Its main window is shown in Figure 5.4. Pop-up windows for defining attributes of a component and port properties are visible there as well. Results of a visual language development are stored in a package (XML structure of a package has been described in [30]) 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*.



Size: 1 · 255 · 100 · 0

Edit Port Properties

Port Name:	p1
Port Type:	alias
Area Connected:	☐
Strict Port:	☑

Class Properties

Class Name:	Resistor
Class Description:	simple resistor
Class Icon:	Resistor_h.gif
Class Is Relation:	☐

Class Fields

Field Name	Field Type	Field Value
u	double	
u1	double	
u2	double	
r	double	
i	double	

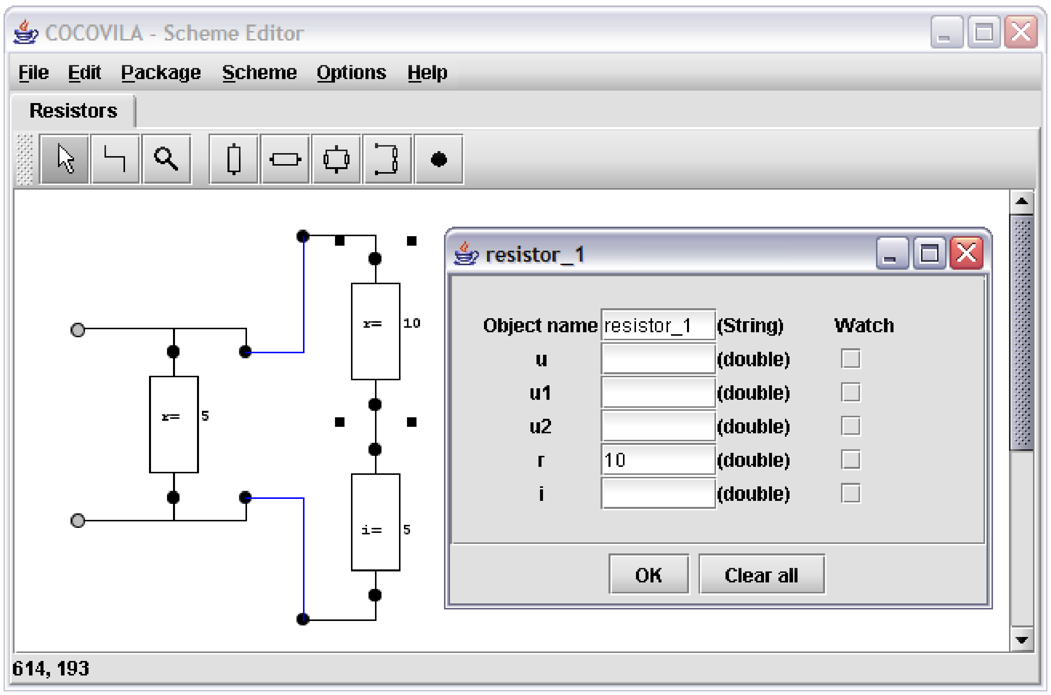
38, 0

Figure 5.4: Class Editor window

Initially, the *Class Editor* has been developed in cooperation with Aulo Aasma [1].

5.4 Scheme Editor

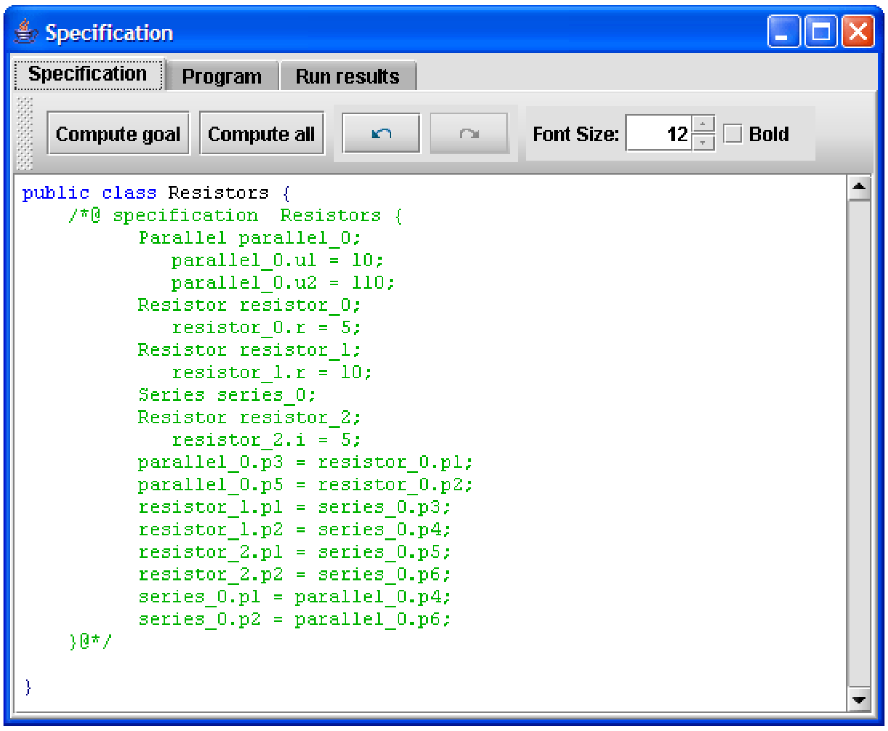
The *Scheme Editor* is a tool for usage of visual languages, i.e. developing schemes, compiling and running programs. It provides an interface for visual programming - building a scheme from visual images of components. The environment generated for a particular visual language allows the user to compose, edit and use schemes in computations through language-specific menus and toolbars.



Object name	resistor_1	(String)	Watch
u		(double)	☐
u1		(double)	☐
u2		(double)	☐
r	10	(double)	☐
i		(double)	☐

Figure 5.5: Scheme Editor window

In Figure 5.5 we can observe the visual language for simulating electrical circuits, loaded into the *Scheme Editor*. Buttons on the toolbar correspond to visual classes defined in the language and are used for instantiating objects in the scheme. The scheme in this figure has been composed using three objects of class `Resistor` bound (via ports) with objects representing parallel and series connections. We see also a pop-up window that shows the values of attributes for `resistor_1`. Figure 5.6 shows the window with the metainterface that contains the textual representation of this scheme, i.e. its shallow semantics (see section 4.2). In the next section we introduce the implementation of deep semantics of schemes and going further with this example will demonstrate the attribute evaluator for the given scheme.



```
public class Resistors {
    /*@ specification  Resistors {
          Parallel parallel_0;
             parallel_0.u1 = 10;
             parallel_0.u2 = 110;
          Resistor resistor_0;
             resistor_0.r = 5;
          Resistor resistor_1;
             resistor_1.r = 10;
          Series series_0;
          Resistor resistor_2;
             resistor_2.i = 5;
          parallel_0.p3 = resistor_0.p1;
          parallel_0.p5 = resistor_0.p2;
          resistor_1.p1 = series_0.p3;
          resistor_1.p2 = series_0.p4;
          resistor_2.p1 = series_0.p5;
          resistor_2.p2 = series_0.p6;
          series_0.p1 = parallel_0.p4;
          series_0.p2 = parallel_0.p6;
    }@*/

}
```

Figure 5.6: Scheme's shallow semantics

5.5 Implementation of deep semantics

The part of the system responsible for the synthesis of attribute evaluators of schemes is embedded into the *Scheme Editor* and hidden from the user. A generated attribute evaluator is a Java class and the evaluation algorithm is a method of this class.

The three kinds of deep semantics of schemes (defined in 4.3) are implemented in the following way.

The deep semantics *DS1* can be used for evaluating explicit computational problems, defined by introducing goals in the specification. In order to apply this kind of deep semantics, in the *GUI* of the specification window user has to choose "*Compute goal*" option (see Figure 5.6).

The deep semantics *DS2* can be used for solving the largest computational problem on a scheme. This means that the system will create an evaluation algorithm such that it will be able to evaluate all possible attributes defined in attribute model of a scheme. In the *GUI* the corresponding option is "*Compute all*".

With *DS3* the situation is different. Its implementation depends not only on the definition of functional dependences but their realizations as well. Then the methods of metaclasses have the same role which the semantic programs have that represent attribute dependencies in an ordinary attribute grammar.

Before running an attribute evaluator, the following implicit steps are performed by the system:

- the textual specification of a scheme as well as all related metainterfaces of components are recursively parsed;
- parsed information is translated into internal representation (graph) that corresponds to the attribute model in flattened form;
- the planner tries to build the sequence (called *evaluation algorithm*) of application of functional dependences (using algorithms discussed in sections 3.4 and 3.6) that will solve the problem;
- if previous step is successful, the sequence may need to be optimized (in case of *DS1*), otherwise system omits any further steps;
- the code generator creates source code for the new Java class of attribute evaluator that includes the method `compute()`. This method implements the sequence produced by the planner;
- the classloader compiles the source code of an attribute evaluator and executes it (calls method `compute()`) within the *Scheme Editor* using the Java reflection technology;
- the executed code evaluates all required attributes and the system propagates computed values back into the scheme.

In the next sections we will discuss most complicated steps in the synthesis of attribute evaluators – planning and code generation. All other steps are rather straitforward.

5.5.1 Planning

The planner is the core of program synthesis technology implementation in CoCoViLa. It implements algorithms of attribute evaluation (discussed in Chapter 3), and working with the internal representation of attribute models, searches the solution for the computational problem defined in the scheme.

The planner's general strategy is "first create such sequence of application of functional dependencies that will evaluate all attributes that can be evaluated". If in a particular computational problem goals have been defined (and *DS1* has been chosen), the planner works until all functional dependences used for evaluating goal appear in the sequence, then calls the optimization algorithm which works backwards from the end of a sequence and removes unnecessary dependencies, i.e. dependecies that are not used in the evaluation of goals.

5.5.2 Code generation

Java source code is extracted from a synthesized evaluation algorithm returned by the planner. Code generation is the final step in the synthesis process.

Let us have a metaclass for a scheme (i.e. textual representation of attribute model of a scheme):

```
public class ElectricCircuits {
    /*@ specification  ElectricCircuits {
        Resistor res;
           res.r = 10;
        res.i -> res.u;
    }@*/
}
```

The metaclass of a scheme is a starting point for the whole synthesis process. After the planner has returned an algorithm, a new class is generated by the code generator with the same name as the metaclass of a scheme. It implements the interface `IComputable` with the method `compute()`:

```
public class ElectricCircuits implements IComputable {
    ...
    public void compute( Object... args ) {
        ...
    }
}
```

Now we show that the body of the method `compute()` is the implementation of an evaluation algorithm. In our example the evaluation algorithm is as follows:

```
public void compute( Object... args ) {
    res.r = 10;
    res.u = (res.i * res.r);
}
```

Variables, declared in metainterfaces, are not proper members of Java classes. If they become usable in the synthesized Java program (attribute evaluator), these variables have to be declared in Java classes. In our example the metaclass of the scheme contains one variable `res`, i.e. the declaration of it is as follows:

```
public class ElectricCircuits implements IComputable {
    public Resistor res = new Resistor();
    ...
}
```

whereas `Resistor` is a metaclass, and all its variables have to be declared in Java as well

```
public class Resistor {
    double i;
    double r;
    double u;
    ...
}
```

In this example the specification contains one goal and one assumption expressed by `res.i -> res.u`, which means "compute `res.u` from `res.i`" assuming that the value of `res.i` is assigned from outside of attribute evaluator. I.e. the value of `res.i` must be passed as an element of array `args` being optional parameter of the method `compute()` when it gets executed. Finally, we get the class:

```
public class ElectricCircuits implements IComputable {
    public Resistor res = new Resistor();
    public void compute( Object... args ) {
        res.i = ((java.lang.Double)args[0]).doubleValue();
        res.r = 10;
        res.u = (res.i * res.r);
    }
}
```

The specification language allows specifying alternative outputs of realizations of axioms. Exceptions are used to guarantee the correct continuation or the termination of evaluation in case an error occurs. In the next example we introduce the exception handling in our system. Let us have the specification:

```
class Test1 {
    /*@ specification Test1 {
        ...
        x -> y |(IllegalStateException){calc};
  }@*/
  ...
}
```

After the code generation, exceptions are handled by surrounding the call of a method with `try-catch` statement

```
public class Test1 implements IComputable {
  ...
  public void compute( Object... args ) {
      try {
        y = calc( int x );
      }
      catch( IllegalStateException ex0 ) {
        ex0.printStackTrace();
        return;
      }
  }
  ...
}
```

The reason of using the `return` statement instead of `System.exit(1)` is that we do not actually shut down the program, the goal is to finish processing of the method `compute()`. Due to the fact that the synthesized program is compiled and launched within the bounds of the system, the usage of `System.exit(1)` would cause the shutdown of the whole application.

Subtasks

For each subtask the code generator creates a new Java class that implements the interface `Subtask`.

```
public interface Subtask {
    Object[] run(Object[] in) throws Exception;
}
```

This class is declared as an inner class in the method `compute()`. The advantage of inner classes is that they have a quick access to the member variables of the main class. The code generation for a subtask consists of the following steps:

1. define a subtask class as inner class of `compute()`,
2. create an instance of a subtask class,

3. generate a method call with a subtask instance as parameter

From the given metainterface

```
public class Test2 {
  /*@ specification  Test2 {
      int a, b, c, d;
      ...
      [a -> b],c -> d {calc};
      ...
  }@*/

  public int calc(Subtask sub) {
      ...
  }
}
```

the following source code is generated (comments “1”, “2” and “3” represent the code generation steps described above):

```
public class Test2 implements IComputable {
  ...
  public void compute() {
      ...
      //1
      class Subtask_1 implements Subtask {

        public Object[] run(Object[] in)
                          throws Exception {
          a = ((Integer)in[0]).intValue();     (*)
          ...
          return new Object[]{ b };
        }
      }
```

```
        //2
        Subtask_1 subtask_1 = new Subtask_1();
        //3
        d = calc(subtask_1, c);
    }
    public int calc(Subtask subtask, int count) {
        try {
            for( int i = 0; i < count; i ++)
                Object[] in = new Object[1];
                in[0] = i;                          (**)
                Object[] out = subtask.run(in);
                ...
            }
        } catch(Exception ex) {}
    }
}
```

Notice that the method `run()` is declared with the parameter `in` of type `Object[]`. This parameter is used to pass input values to the subtask. First, before calling the method `run()`, such array of Objects has to be created, whereas variables of primary types must be wrapped into corresponding objects (e.g. `int` to `Integer`), in Java version 5 this procedure, called *autoboxing*, is automatically handled by the compiler (**). However to retrieve the input value in the `run` method, an element of array has to be casted to the required type (*). The code generator deals with it automatically knowing the exact type of an attribute used solving the computational problem of a subtask.

For some computational problems the generated evaluation algorithm contains subtasks that have other subtasks nested inside. In such cases, the code generator uses recursive algorithm to generate the source code.

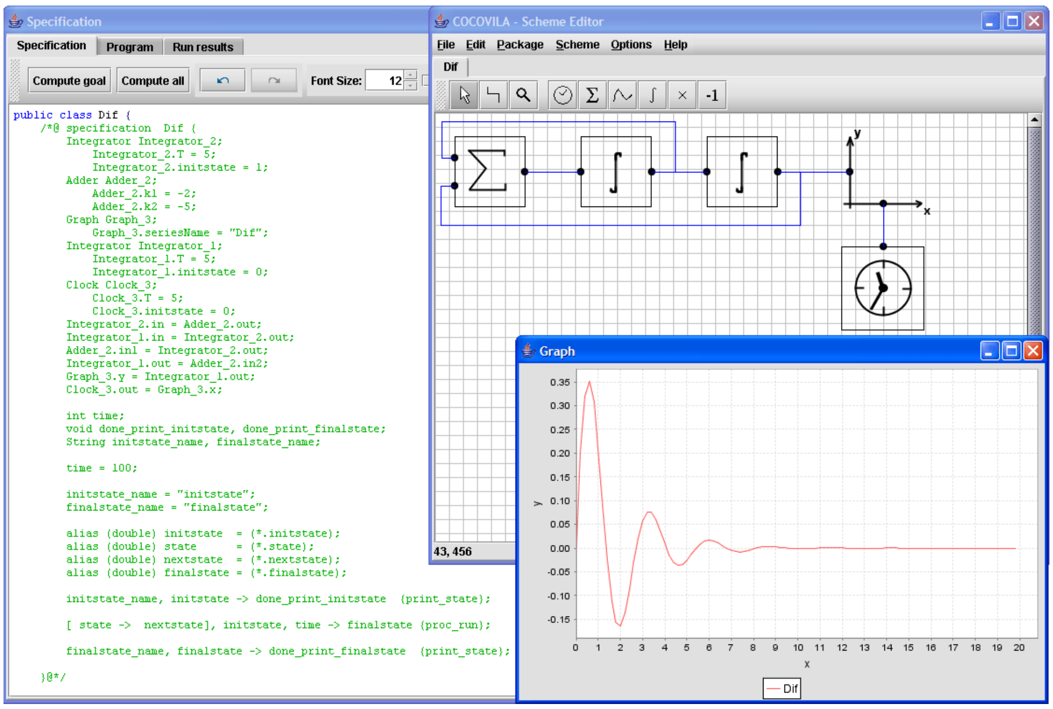
Chapter 6

Experiments

During the implementation of CoCoViLa several demonstration packages (i.e. visual languages) have been developed originating from different domains, e.g. mechanics (package for calculating kinematics of gearboxes), hardware design (package for simulating logical circuits), UML diagrams, neural networks, electrical circuits etc. We are certain that the system is able to handle large schemes.

6.1 Visual language for simulating dynamic systems

CoCoViLa with its automatic program synthesis technique provides a good opportunity for implementing simulation software. In this section we demonstrate the package for finding next state of a dynamic system from a given state and time. Such are systems described by ordinary differential equations. This package has been developed in cooperation with Mait Harf who had initially implemented similar package in NUT. The scheme for a computational problem, shown is Figure 6.1 has been composed using the following components: *Integrator*, *Adder*, *Clock* and *Graph*. As it may be seen from the given figure, a part of the attribute model of the scheme in permanent (starting from the line "`int time`"), i.e. it is possible to declare the common part of the model for all attribute models of schemes (computational prob-



```
public class Dif {
    /*@ specification  Dif {
        Integrator Integrator_2;
            Integrator_2.T = 5;
            Integrator_2.initstate = 1;
        Adder Adder_2;
            Adder_2.k1 = -2;
            Adder_2.k2 = -5;
        Graph Graph_3;
            Graph_3.seriesName = "Dif";
        Integrator Integrator_1;
            Integrator_1.T = 5;
            Integrator_1.initstate = 0;
        Clock Clock_3;
            Clock_3.T = 5;
            Clock_3.initstate = 0;
        Integrator_2.in = Adder_2.out;
        Integrator_1.in = Integrator_2.out;
        Adder_2.in1 = Integrator_2.out;
        Integrator_1.out = Adder_2.in2;
        Graph_3.y = Integrator_1.out;
        Clock_3.out = Graph_3.x;

        int time;
        void done_print_initstate, done_print_finalstate;
        String initstate_name, finalstate_name;

        time = 100;

        initstate_name = "initstate";
        finalstate_name = "finalstate";

        alias (double) initstate  = (*.initstate);
        alias (double) state      = (*.state);
        alias (double) nextstate  = (*.nextstate);
        alias (double) finalstate = (*.finalstate);

        initstate_name, initstate -> done_print_initstate  {print_state};

        [ state ->  nextstate], initstate, time -> finalstate {proc_run};

        finalstate_name, finalstate -> done_print_finalstate  {print_state};

    }@*/
```

Figure 6.1: Dynamic system - scheme, specification and outcome

lems) based on a particular package. The specification of such common part as well as realizations of its axioms (if there are any) are stored in separate files and added automatically into the metaclass of a scheme.

The method `proc_run()` being the realization of the axiom

```
[ state -> nextstate ], initstate, time -> finalstate
```

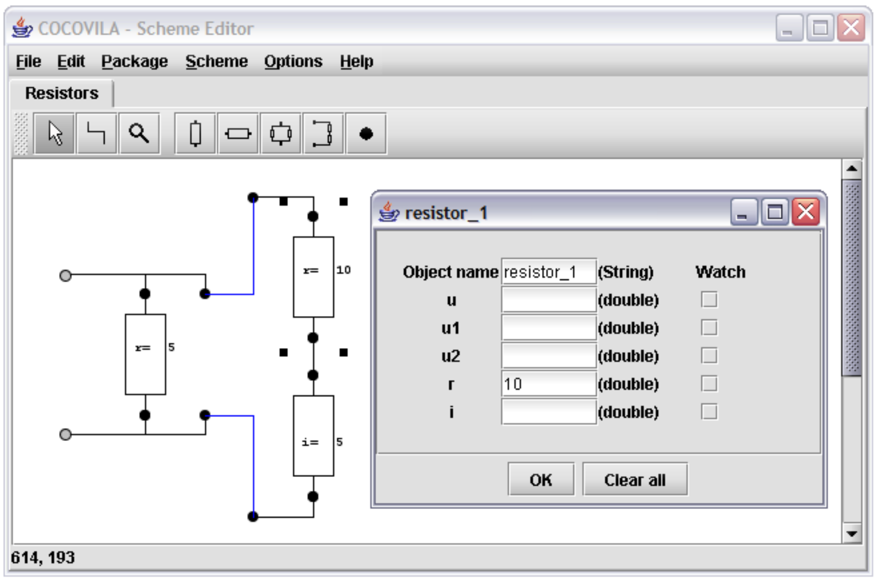
controls the simulation by repeatedly requesting to compute a new state of the system from the current state. This is done by calling the subtask `[ state -> nextstate ]`.

Objects of components *Integrator* and *Clock* have states which change in time and they both contain methods for computing *nextstate* from *state*. Objects of the component *Adder* simply compute output from given inputs. The actual algorithm for attribute evaluation is synthesized for every particular problem, depending on the scheme of a dynamic system.

6.2 Electrical circuits

The problem domain for this experiment is electrical (direct current) circuits. Let us define three concepts – *resistor* element, *parallel* and *series* connections. The specifications of attribute models (based on the Ohm's law) presented in CoCoViLa are as follows:

```
class Resistor {
    /*@ specification Resistor {
        double u, u1, u2, r, i;
        u = u2-u1;
        u = i*r;
        alias p1 = (u1, i, r);
        alias p2 = (u2, i, r);
    }@*/
}

class Parallel {
    /*@ specification Parallel {
        double u, u1, u2, i, i1, i2, r, r1, r2;
        i = i1 + i2;
        1/r = 1/r1 + 1/r2;
        u = i*r;
        u = u2 - u1;
        alias p1 = (u1, i, r);
        alias p2 = (u2, i, r);
        alias p3 = (u1, i1, r1);
        alias p4 = (u1, i2, r2);
        alias p5 = (u2, i1, r1);
        alias p6 = (u2, i2, r2);
    }@*/
}
```



Object name	resistor_1	(String)	Watch
u		(double)	☐
u1		(double)	☐
u2		(double)	☐
r	10	(double)	☐
i		(double)	☐

Figure 6.2: Circuit with three resistors, one parallel and one series branch

```
class Series {
    /*@ specification Series {
        double u, u1, u2, u3, r, i, r1, r2;
        r = r1 + r2;
        u = i*r;
        u = u2 - u1;
        alias p1 = (u1, i, r);
        alias p2 = (u2, i, r);
        alias p3 = (u1, i, r1);
        alias p4 = (u3, i, r1);
        alias p5 = (u3, i, r2);
        alias p6 = (u2, i, r2);
    }@*/
}
```

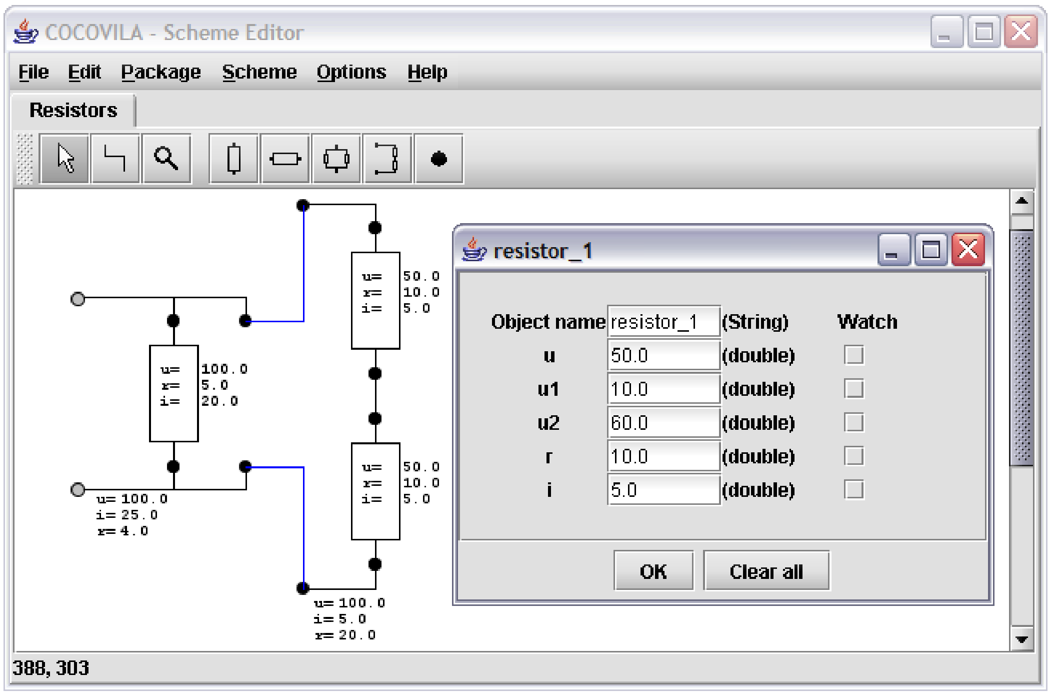
Figure 6.2 represents a scheme containing parallel and series connection of three resistors. We see here also a pop-up window of attribute values for `resistor_1`.

The corresponding shallow meaning of a given scheme is as follows:

```
Parallel parallel_1;
parallel_1.u1 = 0;
parallel_1.u2 = 100;
Resistor resistor_3;
resistor_3.r = 5;
Resistor resistor_1;
resistor_1.r = 10;
Series series_1;
Resistor resistor_2;
resistor_2.i = 5;
parallel_1.p3 = resistor_3.p1;
parallel_1.p5 = resistor_3.p2;
resistor_1.p1 = series_1.p3;
resistor_1.p2 = series_1.p4;
resistor_2.p1 = series_1.p5;
resistor_2.p2 = series_1.p6;
series_1.p1 = parallel_1.p4;
series_1.p2 = parallel_1.p6;
```

After invoking the planner, the following evaluation algorithm is produced:

```
resistor_3.r = 5;
parallel_1.u2 = 100;
parallel_1.u1 = 0;
resistor_1.r = 10;
resistor_2.i = 5;
series_1.r1 = resistor_1.r;
series_1.i = resistor_2.i;
resistor_3.u1 = parallel_1.u1;
series_1.u1 = parallel_1.u1;
parallel_1.u = (parallel_1.u2 - parallel_1.u1);
series_1.u2 = parallel_1.u2;
resistor_3.u2 = parallel_1.u2;
```



resistor_1

Object name	resistor_1	(String)	Watch
u	50.0	(double)	☐
u1	10.0	(double)	☐
u2	60.0	(double)	☐
r	10.0	(double)	☐
i	5.0	(double)	☐

OK Clear all

Figure 6.3: A scheme after evaluation

```
parallel_1.r1 = resistor_3.r;
resistor_3.u = (resistor_3.u2 - resistor_3.u1);
resistor_1.u1 = series_1.u1;
series_1.u = (series_1.u2 - series_1.u1);
resistor_2.u2 = series_1.u2;
parallel_1.i2 = series_1.i;
resistor_1.i = series_1.i;
series_1.r =( series_1.u/series_1.i);
resistor_1.u = (resistor_1.i * resistor_1.r);
resistor_3.i = resistor_3.u/resistor_3.r;
parallel_1.r2 = series_1.r;
series_1.r2 =( series_1.r-series_1.r1);
resistor_1.u2 = resistor_1.u+resistor_1.u1;
parallel_1.i1 = resistor_3.i;
parallel_1.r =(1/ ((1 / parallel_1.r1) + (1 / parallel_1.r2)));
parallel_1.i = (parallel_1.i1 + parallel_1.i2);
```

```
resistor_2.r = series_1.r2;
series_1.u3 = resistor_1.u2;
resistor_2.u = (resistor_2.i * resistor_2.r);
resistor_2.u1 = series_1.u3;
```

This algorithm corresponds to the deep semantics *DS2* of the scheme, namely it solves the largest solvable problem on the scheme. Figure 6.3 shows the results of computation as a visual feedback in the Scheme Editor window, as well as values of attributes of `resistor_1` in the pop-up window.

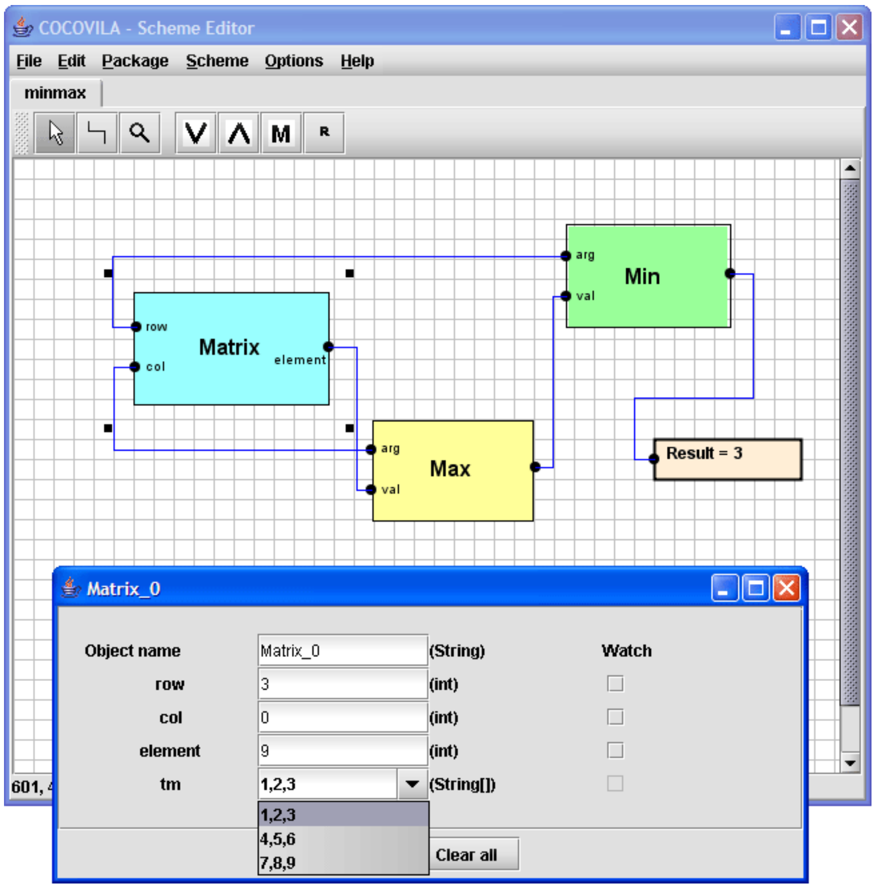
6.3 Minimax

Here we present the *minimax* computational problem: "find minimal value of maximal values of elements of rows of a matrix". With this example it is easy to demonstrate the usage of subtasks in computations.

This computational problem has been implemented using three components: *Matrix*, *Min* and *Max*. The metaclass *Matrix* contains the axiom $row, col, matrix \rightarrow element\{getElement\}$ for retrieving an element of a matrix (which is represented as two-dimensional array of integers). Metaclasses *Min* and *Max* contain axioms with subtasks for finding the minimum $((arg \rightarrow val) \rightarrow minval\{getMinVal\})$ and maximum $((arg \rightarrow val) \rightarrow maxval\{getMaxVal\})$ values of a function that computes val from arg respectively. The realizations of these axioms are similar, i.e. they both implement loops that iterate through the set of arguments and remember minimum or maximum value among values returned by the subtask $arg \rightarrow val$.

The shallow semantics of a scheme (in Figure 6.4) that represent the *minimax* computational problem is as follows:

```
/*@ specification  Minmax {
    Matrix Matrix_0;
        Matrix_0.tm = {"1,2,3", "4,5,6", "7,8,9"};
    Min Min_1;
    Max Max_2;
    Result Result_1;
```



Object name	Matrix_0	(String)	Watch
row	3	(int)	☐
col	0	(int)	☐
element	9	(int)	☐
tm	1,2,3	(String[])	☐

1,2,3
4,5,6
7,8,9

Clear all

Figure 6.4: The scheme of *minimax*

```
    Matrix_0.element = Max_2.val;
    Max_2.maxval = Min_1.val;
    Matrix_0.row = Min_1.arg;
    Max_2.arg = Matrix_0.col;
    Result_1.value = Min_1.minval;
    -> Min_1.minval;
}@*/
```

We use the deep semantics *DS1* for *minimax*, i.e. the planner creates an attribute evaluation algorithm for finding the value of the goal `Min_1.minval`. Figure 6.5 shows the *and-or* search tree for *minimax*. Arrows denote the branch that solves the problem. The Java source code of the attribute eval-

$$\begin{tikzcd}
 & & & S_0 \arrow[dl, dash] \arrow[dr, dash] \arrow[dr, shift left=1.5ex] & & & \\
 & & S_{max} \rightarrow maxval \arrow[dl, dash] & & S_{min} \rightarrow minval \arrow[dr, dash] \arrow[dr, shift left=1.5ex] & & \\
 & S_{max} \arrow[dl, dash] \arrow[dr, dash] & & & & S_{min} \arrow[dl, dash] \arrow[dl, shift right=1.5ex] \arrow[dr, dash] & \\
S_{max} \rightarrow maxval \arrow[d, dash] & & S_{min} \rightarrow minval \arrow[d, dash] & & S_{max} \rightarrow maxval \arrow[d, dash] \arrow[d, shift right=1.5ex] & & S_{min} \rightarrow minval \arrow[d, dash] \\
S_{max} & & S_{min} & & S_{max} & & S_{min}
\end{tikzcd}$$

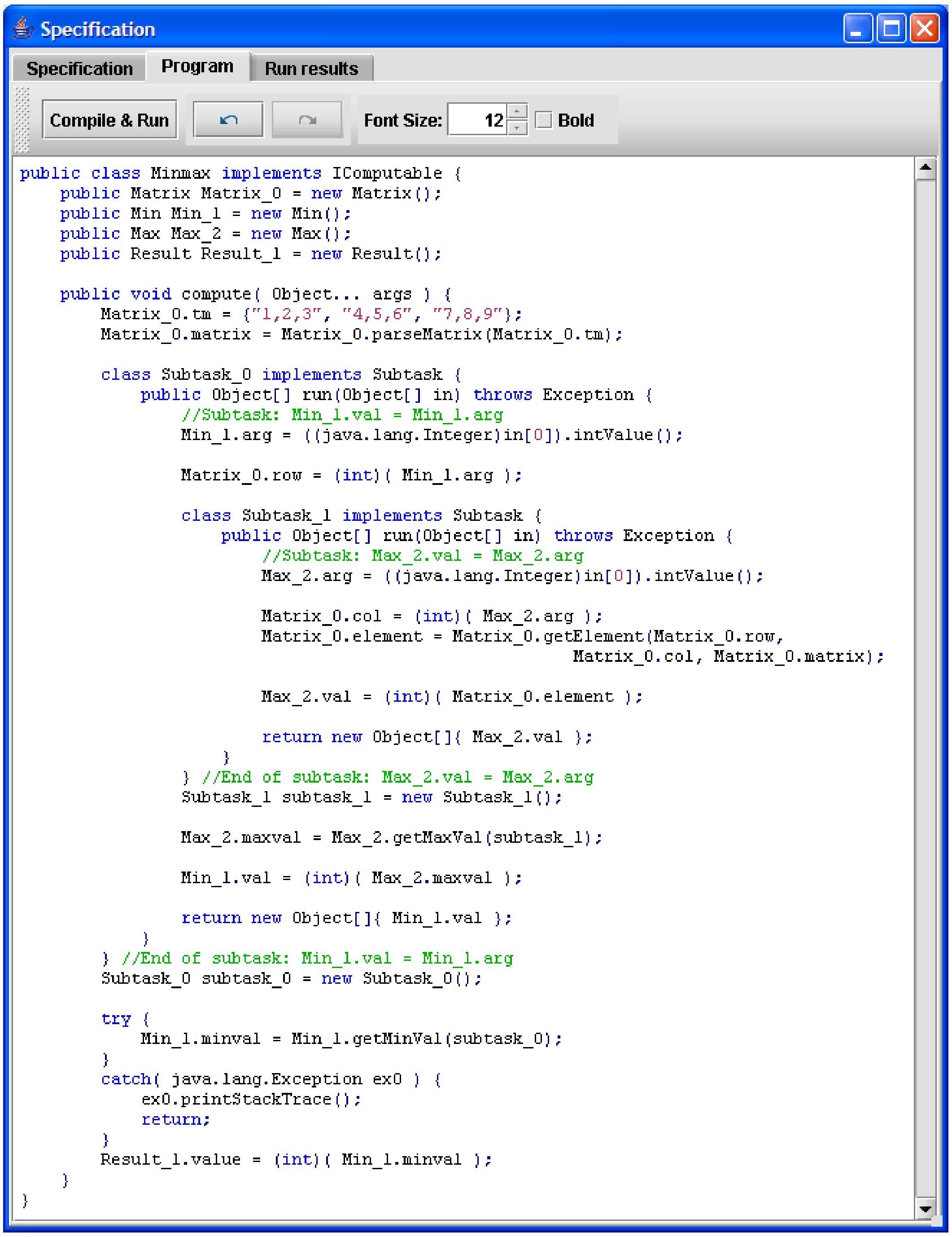
Figure 6.5: *And-or* search tree for *minimax*

uator for *minimax* produced by the code generator is shown in Figure 6.6.

6.4 Semantics *DS3*

The semantics *DS3* has been used in a package "UML Class Diagrams" developed by Ando Saabas that is intended for the generation of Java class templates from the class diagrams (Figure 6.7). The visual language includes images of classes and their relations: inheritance and aggregation. The synthesized program is a source generator for the class templates.

We are developing an UML-like visual language for the dynamic web service composition from ontologies described using OWL (Web Ontology Language). The usage of deep semantics *DS3* will be assured by the implementation of Java methods (realizations of axioms) that will generate OWL descriptions of composite processes from atomic processes of services defined and put together in the scheme.



```
public class Minmax implements IComputable {
    public Matrix Matrix_0 = new Matrix();
    public Min Min_1 = new Min();
    public Max Max_2 = new Max();
    public Result Result_1 = new Result();

    public void compute( Object... args ) {
        Matrix_0.tm = {"1,2,3", "4,5,6", "7,8,9"};
        Matrix_0.matrix = Matrix_0.parseMatrix(Matrix_0.tm);

        class Subtask_0 implements Subtask {
            public Object[] run(Object[] in) throws Exception {
                //Subtask: Min_1.val = Min_1.arg
                Min_1.arg = ((java.lang.Integer)in[0]).intValue();

                Matrix_0.row = (int)( Min_1.arg );

                class Subtask_1 implements Subtask {
                    public Object[] run(Object[] in) throws Exception {
                        //Subtask: Max_2.val = Max_2.arg
                        Max_2.arg = ((java.lang.Integer)in[0]).intValue();

                        Matrix_0.col = (int)( Max_2.arg );
                        Matrix_0.element = Matrix_0.getElement(Matrix_0.row,
                                                Matrix_0.col, Matrix_0.matrix);

                        Max_2.val = (int)( Matrix_0.element );

                        return new Object[]{ Max_2.val };
                    }
                } //End of subtask: Max_2.val = Max_2.arg
                Subtask_1 subtask_1 = new Subtask_1();

                Max_2.maxval = Max_2.getMaxVal(subtask_1);

                Min_1.val = (int)( Max_2.maxval );

                return new Object[]{ Min_1.val };
            }
        } //End of subtask: Min_1.val = Min_1.arg
        Subtask_0 subtask_0 = new Subtask_0();

        try {
            Min_1.minval = Min_1.getMinVal(subtask_0);
        }
        catch( java.lang.Exception ex0 ) {
            ex0.printStackTrace();
            return;
        }
        Result_1.value = (int)( Min_1.minval );
    }
}
```

Figure 6.6: Synthesized code of attribute evaluation algorithm of *minimax*

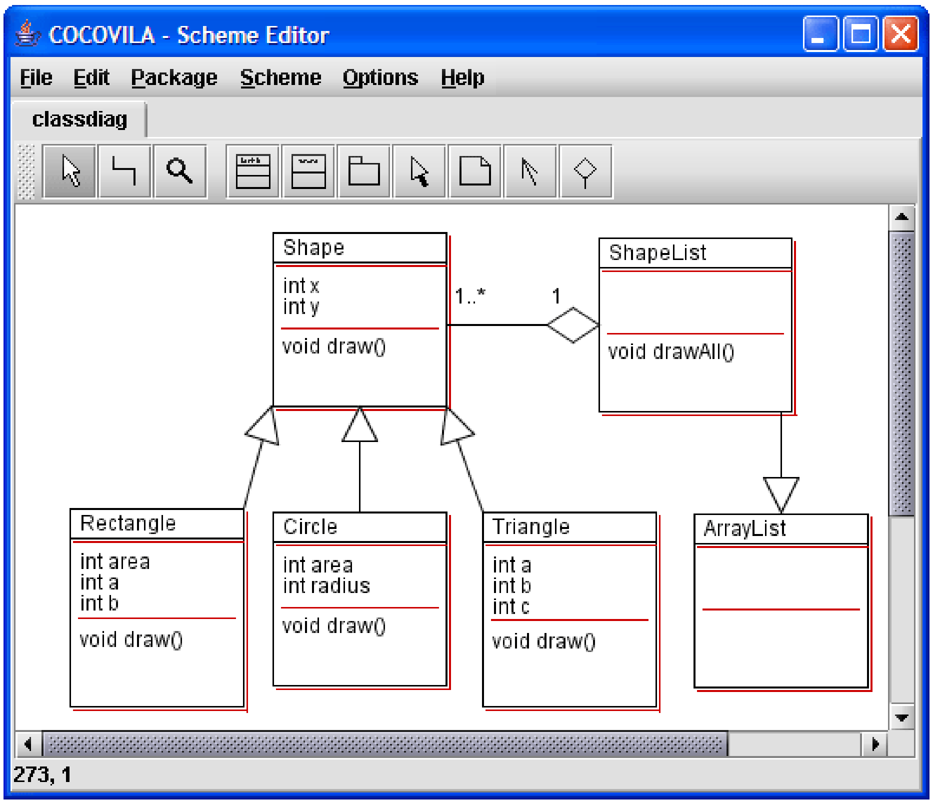


Figure 6.7: A scheme of UML class diagrams

Chapter 7

Related Work

There are a number of works where attributes are used in presentation of semantics of graphs. Götler [8] introduces a formalization of a notion that he calls *graphic*. A graphic is considered to consist of a graph describing the overall structure and a set of attributes describing the shape, placement, etc. of the nodes and edges of the underlying graph. The formal handling of graphics is done by attributing the rules of graph grammars and by passing the attributes up and down the derivation tree of the graphic.

The paper from Alpern, et al. [2] introduces the concept of *attributed graph specification (AGS)* and develops a theory of strongly typed graphs. Graphs are modeled as collections of boxes connected by cables. Cables and boxes contain ports. A cable and a box are connected through ports. A graph is specified as a composition of boxes and cables. Attributes are associated with boxes and and attribute evaluation rules are attached to the composition rules. An attribute evaluation rule associated with a box composition can additionally specify a direction of flow of the attribute values from one port to another port in the carrier. We use the similar technique in the construction of schemes, i.e. boxes are visual components with ports, and cables are the bindings. Authors pay much attention to syntax and specification language. Attribute evaluation strategies, *static* and *dynamic* are presented only briefly.

The NUT system [41] is a programming tool supporting declarative programming in a high-level language, automatic program synthesis and visual specification of problems by means of schemes. The central part of the system is Structural Synthesis of Programs (SSP), which uses intuitionistic propositional logic. NUT restricts its attention to constructing programs from pre-programmed modules, rather than from primitive instructions of a programming language. The NUT specification language is an object-oriented language extended with features for program synthesis, the pre-programmed modules are methods of classes supplied with specifications. Our attribute evaluation method uses the ideas of SSP from NUT.

A special purpose application of attribute semantics of graphs is presented in PhD thesis by Ramanand Mandayam [20]. In this dissertation author introduces a framework and a language for the specification and evaluation of performance attributes of VLSI (*Very Large Scale Integration*) systems. The PDL (*Performance Description Language*) is presented as performance modeling language based on *Attributed Nodes-Only Grammars*.

The PDL is used for the specification of performance attributes supplied with computable functions for the evaluation of these attributes and indicating temporal relationships between design instances and the evaluation of attribute values. The proposed language allows to attach attributes to objects in the design and to propagate attribute values up, down and laterally across various levels in the design hierarchy. Like the specifications introduced in [2], design objects in PDL are *modules*, *carriers* and *ports*. Modules are the building block of all designs. Carriers represent the wires and nets in the design. Modules contain other modules, carriers and ports. Carriers represent wires and contain ports connecting ports of one module to ports of other modules.

Mandayam explains why graph grammars are more suitable than string grammars for modeling VLSI and digital designs. The reasons are the following:

- Relationships between symbols in a graph grammar can be more complex that the linear relationships of string grammars.
- The production rules are more powerful.

- A graph is a natural representation for VLSI designs as a network of nodes and edges.
- The edges may improve expressive capabilities of grammars e.g. by specifying the context conditions.

A subset of graph grammars called Nodes-Only Grammars (NOG) are introduced as an appropriate formalism for specifying and recognizing designs presented in the paper. Node labels are considered as symbols in the grammar. Those symbols that appear in the right hand side of a production rule can appear in any arbitrary order in an actual instance of that production in an input.

```
object X contains
     objects P, Q, R, S;
end object X;
```

Figure 7.1: Design pseudo code

Figure 7.1 presents the design (in pseudo-code) as a valid instance of composition $C_1 = \langle X, \{P, Q, R, S\}\rangle$. A composition C_i in a NOG is defined as a 2-tuple $C_i = \langle \alpha_i, \beta_i \rangle$ where α_i is a single node label and β_i is a set of node labels. The composition C_i specifies that an occurrence of node α_i can be replaced by the set of nodes β_i. Composition rules represent the production rule applied (or applicable) at a node (the left-hand side) in a graph. In the context of recognizing valid designs it specifies that if the design contains and object α_i then α_i must be composed of a set of objects β_i.

More (hierarchical) composition rules are introduced, including *module*, *carrier* and *port* composition. The design is called acceptable by a graph grammar G if it is well-formed in G.

Attributed Nodes-Only Grammars (*ANOG*) are a subset of graph grammars that are extended with attributes and their evaluation rules. Attributes in PDL are named variables associated with non-terminal and terminal symbols in the NOG.

In the framework of ANOG a design is valid if: a) there exists a valid derivation graph for the design, and b) all instances in the attribute instance graph are consistently evaluated. An evaluation is consistent if values assigned to all attribute instances in the attribute instance graph satisfies the set of equalities specified by the attribute evaluation rules. In the context of a PDL program specification, a design is valid if: a) all objects in the design are valid instances of corresponding objects in the PDL program, and b) there exists an acyclic evaluation sequence for the design.

Performance attributes are classified into two categories: *static* and *dynamic*. Static attributes do not depend on dynamic attributes and once evaluated, do not change with a change in the data. The concept of controlled cyclic dependencies among attribute occurrences is introduced and a mechanism to conceptually break cycles in order to determine an evaluation sequence is presented.

The work of Mandayam presents a rather intricate set of concepts for handling semantics of a special visual language. Our goal will be to simplify the set of concepts and to make the application more general.

Penjam in [27] shows how attribute semantics of programming languages can be presented by means of computational models, initially introduced by Tyugu in [33]. He proves the semantical equivalence between attribute and computational models both being two approaches to program and compiler specification and implementation. Computational problems are used for knowledge representation and problem solving using the method of structural synthesis of programs [25].

Vilo in [44] presents the implementation of languages based on attribute grammars and computational models discussed by Penjam. The experimental work has been done in NUT. The *dynamic* implementation of attribute grammars has been accomplished in the following way: building the computational model of each derivation tree. The relation between all attribute instances of a corresponding syntax tree has to be included into the model. The attribute evaluation algorithm is synthesized deriving the sequent

$$M(Tree) \vdash X \underset{\lambda x.F}{\longrightarrow} Y,$$

where M is a computational model, X denotes the set of input variables and Y denotes the set of synthesized attributes of the tree. Each production of a given attribute grammar is represented by a class in NUT. Attribute dependencies are introduced with the equality operation, i.e. the node corresponding to right-hand side nonterminal of a production rule is made equivalent to the node on the left-hand side of an other production rule.

The *static* realization of attribute grammars allows generating attribute evaluators independently from the syntax trees during the compilation. As in the case with dynamic implementation, each production of a given attribute grammar is represented by a class in NUT. But instead of just having simple attribute dependences in the form of equalities, higher-order relations in the form of conditional computability statements (with subtasks) are added into the models that correspond to visits into the subtrees. This method works only when the class of attribute grammars is restricted to *absolutely noncircular* ones. The evaluation algorithm is a derivation of a sequent

$$\vdash Tree, P_0 \xrightarrow[\lambda t.F]{} S,$$

where S denotes the computability of synthesized attributes of the root of tree t and P_0 denotes the computational model corresponding to the only production rule with S_0 on the left-hand side.

The last two papers discussed conventional attribute semantics of programming (string) languages. It has been a good starting point for the present thesis introducing the semantics of scheme (graph) languages.

Chapter 8

Conclusions

In this thesis we have introduced a method for representing the semantics of visual schemes by means of attribute models.

First, we have presented attribute models, composition of attribute models, attribute models in flattened form and different types of attribute dependencies in a form suitable for presenting semantics of schemes. We use higher-order attribute models that contain functional dependences with subtasks (*hofd*). Higher-order attribute models are more expressive than models without *hofd*s and allow synthesizing recursive, branching or cyclic programs. We have shown the technique of dynamic attribute evaluation on simple attribute models as well as on higher-order attribute models.

Second, we have defined several kinds of semantics of schemes. The semantics of schemes has been described on two different levels. The shallow semantics gives a textual representation of the graph underlying a scheme. The deep semantics of a scheme gives a set of programs that can be automatically derived from the scheme or it gives the value of a distinguished attribute as a meaning of the scheme.

An essential part of the work is implementation of the attribute evaluation technique in the system CoCoViLa that is a tool for implementation of visual languages. We have used the concept of metainterfaces as an extension of Java classes and the specification language of metainterfaces is used for the textual specification of attribute models of visual components as well schemes of visual languages. In CoCoViLa, a visual language is implemented as a set of visual classes. There are two publications on CoCoViLa made during the last year in co-authorship with A. Saabas and E. Tyugu, see Appendix A. The first introduces the tool as a compiler-compiler for visual languages [9] and the second one as a visual tool for generative programming [10]. The third paper written together with Tyugu introduces the concept of attribute models and implementation of deep semantics of schemes in CoCoViLa. It has been accepted for JCKBSE'06 (Joint Conference on Knowledge-Based Software Engineering) that will be held in August 2006.

Visuaalsete Keelte Atribuut Semantika.

Pavel Grigorenko

Resümee

Visuaalsed spetsifitseerimiskeeled on muutumas populaarseks, kuid nende kasutamist piirab täpse semantika puudumine. Programmeerimiskeelte valdkonnas on semaktikat hõlbus realiseerida atribuutgrammatikate abil, mis võimaldavad lähteteksti järkjärgult teisendada masinkoodiks. Antud töös on atribuutgrammatikaid üldistatud ja sobitatud visuaalsete keelte semantika esitamiseks.

Esiteks, käesolevas töös on laiendatud atribuutmudelte mõistet. Me tõime sisse atribuutmudelite kompositsiooni, atribuutmudelite lameda kuju ja kasutame kõrgemat järku funktsionaalseid sõltuvusi sisaldavaid atribuutmudeleid skeemide semantika esitamiseks. Need mudelid võimaldavad sünteesida hargnevaid, rekursiivseid ja iteratiivseid programme,mida me kasutame atribuutide dünaamiliseks väärtustamiseks.

Teiseks, me defineerisime visuaalsetele skeemikeeltele mitut liiki semantikad. Semantikad on esitatud kahel tasemel: pindmine semantika annab tekstilise skeemide esituse. Süvasemantika annab skeemist kui spetsifikatsioonist sünteesitavad programmid, või ka skeemi tähenduse skeemi peaatribuudi väärtuse näol.

Suur osa tööst on pühendatud atribuutide väärtustamise realisatsioonile programmeerimiskeskkonnas CoCoViLa, mis on visuaalsete keelte realiseerimise vahend. See on Java-põhine keskkond, milles Java klasse saab laiendada metainterfeissidega. Viimased on atribuutmudelite kirjeldused klasside kommentaarides ja esitavad nii visuaalsete komponentide kui ka skeemide mudeleid. Selle kohta on viimasel aastal avaldatud kaks publikatsiooni koos A. Saabase ja E. Tõuguga, millest esimeses esitatakse CoCoViLa-t kui visuaalsete keelte kopilaatorite kompilaatorit [9] ja teises kui generatiivse programmeerimise visuaalset vahendit [10]. Skeemikeelte atribuutmudelid ja süva-semantika on esitatud ühises ettekandes koos E. Tõuguga, mis on vastu võetud 2006. a. augustis toimuvale rahvusvahelisele konverentsile (Joint Conference on Knowledge-Based Software Engineering 2006).

Bibliography

[1] Aulo Aasma. Visuaalsete keelte konstruktor. Diploma thesis, Tallinn University of Technology, 2004.

[2] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweeney, and F. Kenneth Zadeck. Graph attribution as a specification paradigm. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 121–129. ACM press, Boston, MA, November 1988.

[3] Horst Bunke. Graph grammars as a generative tool in image understanding. In *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 8–19, London, UK, 1983. Springer-Verlag.

[4] Juan de Lara and Hans Vangheluwe. Defining visual notationsand their manipulation through meta-modeling and graph transformation. *Journal of Visaul Languages and Computing*, 15:309–330, 2004.

[5] Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

[6] Martin Erwig. Visual graphs. In *15th IEEE Symp. on Visual Languages*, pages 122–129, 1999.

[7] Robert Esser and Jörn Janneck. A framework for defining domain-specific visual languages. In *Workshop on Domain Specific Visual Languages, in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Oopsla*, 2001.

[8] Herbert Göttler. Attributed graph grammars for graphics. In *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, pages 130–142, London, UK, 1983. Springer-Verlag.

[9] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Cocovila - compiler-compiler for visual languages. *Electr. Notes Theor. Comput. Sci.*, 141(4):137–142, 2005.

[10] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Visual tool for generative programming. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 249–252, New York, NY, USA, 2005. ACM Press.

[11] Mait Harf. Structural synthesis of programs using regular data structures. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *Proceedings 6th Nordic Workshop on Programming Theory, NWPT'94, Aarhus, Denmark, 17–19 Oct 1994*, pages 194–202. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[12] Mait Harf, Kristiina Kindel, Vahur Kotkas, Peep Küngas, and Enn Tyugu. Automated program synthesis for Java programming language. *Lecture Notes in Computer Science*, 2244:157–164, 2001.

[13] Mait Harf and Enn Tyugu. Algorithms for structural synthesis of programs. *Programming and Computer Software*, 1980.

[14] Gail E. Kaiser. Incremental dynamic semantics for language-based programming environments. *ACM Trans. Program. Lang. Syst.*, 11(2):169–193, 1989.

[15] Donald Knuth. Semantics of context-free grammars. *Mathematical Systems Theory*, 2:127–145, 1968.

[16] Vahur Kotkas, Jaan Penjam, and Enn Tyugu. Ontology-based design of surveillance systems with NUT. In *Proceedings 3rd Int. Conf. on Infor-*

mation Fusion, FUSION'00, Paris, France, 10–13 July 2000, Volume 2, pages WeB4–3–WeB4–9. Int. Soc. of Information Fusion / EuroFusion, 2000.

[17] Sven Lämmermann. Automated composition of java software. Licentiate thesis, Department of Teleinformatics, Royal Institute of Technology, Sweden, May 2000.

[18] Sven Lämmermann. *Runtime Service Composition via Logic-Based Program Synthesis*. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, June 2002.

[19] Sven Lämmermann and Enn Tyugu. A specification logic for dynamic composition of services. In *International Workshop on Distributed Dynamic Multiservice Architectures*, Phoenic, Arizona, USA, April 2001.

[20] Ramanand Mandayam. *Performance modeling of VLSI systems*. PhD thesis, Banaras Hindu University, 1994.

[21] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[22] Mihhail Matskin and Enn Tyugu. Strategies of structural synthesis of programs. In *Proceedings 12th IEEE Intl. Automated Software Engineering Conf., ASE'97, Incline Village, Nevada, USA, 3–5 Nov 1997*, pages 305–306. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[23] Mihhail Matskin and Enn Tyugu. Strategies of structural synthesis of programs and its extensions. *Computing and Informatics*, 20:1–25, 2001.

[24] Merik Meriste and Jaan Penjam. Attributed models of executable specifications. In *PLILP*, pages 459–460, 1995.

[25] Grigori Mints and Enn Tyugu. Justification of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[26] Jukka Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[27] Jaan Penjam. Computational and attribute models of formal languages. *Theoretical Computer Science*, 1990.

[28] Jaan Penjam and Enn Tyugu. Constraints in NUT. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Proceedings NATO ASI on Constraint Programming, Pärnu, Estonia, 13–24 Aug 1993*, volume 131, pages 330–349. Springer-Verlag, Berlin, 1994.

[29] Maarten C. Pennings. *Generating incremental attribute evaluators.* PhD thesis, Computer Science, Utrecht University, November 1994.

[30] Ando Saabas. A framework for design and implementation of visual languages. Master's thesis, Tallinn University of Technology, 2004.

[31] Joao Saraiva. *Purely Functional Implementation of Attribute Grammars.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1999.

[32] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, New York, NY, USA, 2003. ACM Press.

[33] Enn Tyugu. Knowledge-based programming. *Addison-Wesley, N.Y.*, 1988.

[34] Enn Tyugu. Declarative programming in a type theory. In B. Mller, editor, *Constructing Programs from Specifications*, pages 451–472. North-Holland, 1991.

[35] Enn Tyugu. Attribute models of design objects. In J. S. Gero and E. Tyugu, editors, *Proceedings IFIP TC 5 / WG 5.2 Workshop on For-*

mal Design Methods for CAD, Tallinn, Estonia, 16–19 June 1993, volume 18, pages 33–44. Elsevier, Amsterdam, 1994.

[36] Enn Tyugu. Using classes as specifications for automatic construction of programs in the NUT system. *Automated Software Engineering: An International Journal*, 1(3–4):315–334, 1994.

[37] Enn Tyugu. From visual specifications to executable code. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader: Proceedings ECOOP'98 Workshops, Demos, and Posters, Brussels, Belgium, 20-24 July 1998*, volume 1543, pages 499–501. Springer-Verlag, Berlin, 1998.

[38] Enn Tyugu. On the border between functional programming and program synthesis. *Proceedings of the Estonian Academy of Sciences: Engineering*, 4(2):119–129, 1998.

[39] Enn Tyugu and Ando Saabas. Problems of visual specification languages. In *Proc. of the 35th International Conference on IT + SE, Gurzuf*, pages 155–157, 2003.

[40] Enn Tyugu and Tarmo Uustalu. Higher-order functional constraint networks. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Proceedings NATO ASI on Constraint Programming, Pärnu, Estonia, 13–24 Aug 1993*, volume 131, pages 116–139. Springer-Verlag, Berlin, 1994.

[41] Enn Tyugu and Rando Valt. Visual programming in NUT. *Journal of Visual Languages and Computing*, 8(5-6):523–544, 1997.

[42] Enn Tyugu and Benjamin Volozh. Semantic processing of schemes: Concepts and implementation. Technical Report TRITA-IT R 93:04, Royal Institute of Technology, 1993.

[43] Tarmo Uustalu. Extensions of structural synthesis of programs. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *Proceedings 6th*

Nordic Workshop on Programming Theory, NWPT'94, Aarhus, Denmark, 17–19 Oct 1994, pages 416–428. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[44] Jaak Vilo. Implementing attribute grammars by computational models. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 485–486, London, UK, 1992. Springer-Verlag.

[45] Harald H. Vogt. *Higher order Attribute Grammars.* PhD thesis, Faculteit Wiskunde en Informatica, 1993.

Appendix A

Publications