Deep Semantics of Visual Languages

Pavel Grigorenko[a], Enn Tyugu[a]

[a] *Institute of Cybernetics, Tallinn University of Technology*
Akadeemia tee 21
12618 Tallinn
Estonia

Abstract. Visual specification of software is gaining popularity, but its usage is restricted by the lack of precise semantics of visual languages. In the present work, precise semantics of visual languages is defined. Three kinds of deep semantics of schemes are presented as different ways of usage of attribute models of schemes. Attribute models of a wide class of schemes are defined and higher-order attribute models are introduced. Dynamic evaluation of attributes is used in defining semantics of schemes. **Keywords.** Software synthesis, higher-order attribute models, semantics of visual languages, shallow and deep semantics of schemes, dynamic evaluation of attributes.

1. Introduction

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. In the domain of programming languages this kind 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 been already used for representation of semantics of visual specifications in some specific applications [1], [5].

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. The only relation that binds the icons in this case is the relation expressing that the icons belong to one and the same container – the desktop. Speaking about visual representation of schemes, we allow hidden (not visible) relations in a scheme.

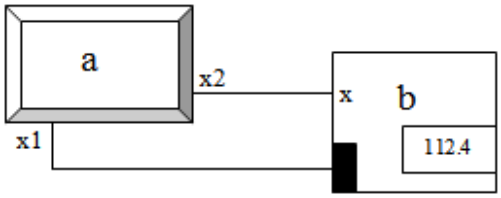

Figure 1. A scheme.

Fig. 1 shows basic features of a visual language: 1) visual identification of type of a component by shape of image, 2) 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.)

In the present work, we first introduce attribute models, including higher order attribute models, as well as attribute evaluation on these models. Thereafter we define shallow and deep semantics of visual specifications, and describe a method of representing the deep semantics of schemes. In the last sections, we give an example of deep semantics of a visual language and refer to the system CoCoViLa that is a tool for implementing the semantics of schemes.

2. Attribute models

In conventional attribute grammars [4], [6], attribute dependencies are related to productions of the syntactic part of a grammar. In our case we have no productions, and we are going to define a concept of attribute model that can represent both semantics of a single object or semantics of a scheme as a whole.

Definition 1. Attribute is a variable with a type.

Definition 2. Attribute dependency is a relation between attributes that is represented by one or several functional dependencies whose inputs and outputs are attributes that are 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

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

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

$x_1,..,x_m \rightarrow y_1,..,y_n[f]$.

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 $x \rightarrow y;\ y \rightarrow x$.

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 $x_1,..,x_m \rightarrow x;\ x \rightarrow x_1,\ldots,\ x_m$.

3. *Equation*, i.e. $x = y + z$ can be presented as a collection of functional dependencies, in the given example as $y, z \rightarrow x;\ x, y \rightarrow z;\ x, z \rightarrow y$.

4. *Preprogrammed procedure* with attributes $x_1,..,x_m$ as parameters producing a value of attribute y can be presented as $x_1,..,x_m \rightarrow y$.

Definition 3. An attribute model M is a pair (A,R), where A is a finite set of attributes and R is a finite set of attribute dependences binding these attributes.

Attribute models $M' = (A',R')$ and $M'' = (A'',R'')$ can be composed into a new attribute model binding some of their attributes by equalities. Let us have a set of equalities $s = \{M'.a = M''.b, \ldots\ \ldots, M'.d = M''.e\ \}$ 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 4. For a set of equalities s that bind some attributes of models $M_1,\ldots, M_n$ we denote by $\cup_s(M_1,\ldots, M_n)$ an attribute model with the set of attributes $A_1 \cup \ldots \cup A_n$ and the set of attribute dependencies $R_1 \cup \ldots \cup R_n \cup s$, and call it composition of $M_1,\ldots, M_n$ with bindings s.

Remark 2.1. 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 2.2. 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 vertex set V = V1 $\cup$ V2 so that both V1 and V2 are independent sets. G = (V1 + V2, E) denotes a bipartite graph with partitions V1 and V2.

Remark 2.3. An attribute model *(A, R)* 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 2.4. An attribute model *(A,R)* 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. Evaluation of attributes

Let *U* and *V* be two sets of attributes of an attribute model *M*. We call a pair *(U, V)* a computational problem on the attribute model *M*. The meaning of a computational problem *(U, V)* is that given values of attributes from *U* find values of attributes of *V* using attribute dependences of *M*. We show that there is a procedure that for any computational problem *(U, V)* on a attribute model *(A,R)* 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 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 3.1. There are very efficient algorithms for value propagation that work in linear time with respect to the size of attribute model, see [8].

Remark 3.2. 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 3.3. There is a procedure that gives a minimal algorithm for solving a computational problem, see [3].

Remark 3.4. 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.

4. 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. First let us notice that we can formally define computational problems on some set of attributes, without making precise which functional dependencies one can use for solving the problems.

Definition 6. Higher-order functional dependency on a given set of attributes is a functional dependency that has also computational problems on this set of attributes as inputs.

Definition 7. Higher-order attribute model is a pair (A,R) 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 $x,y \rightarrow z$ we have now also types like $(u \rightarrow v),x \rightarrow y$, where $u \rightarrow v$ represents a computational problem. 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. The problems on higher-order attribute models can be analyzed by the same technique that has been applied for higher-order constraint networks, see [8], therefore we are not going to discuss them here in any detail, but we will use the higher-order attribute models in semantics of schemes.

5. Syntax and textual presentation of schemes

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 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> <idenitfier>

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.

6. Semantics of schemes

6.1. 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 8. 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.*

6.2. Deep semantics

Definition 9. 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 values given 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 $\cup s\ (M_1,\ldots, M_n) \cup \{objq.x=v1, \ldots, objp.y=vl\ \}$.

Let *U* and *V* be two sets of attributes of scheme. We call a pair *(U, V)* a computational problem on the scheme. We know that there is a procedure that for any computational problem *(U, V)* 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 Section 3. Let us denote by *S1* the function producing an algorithm for any solvable computational problem on a scheme.

Definition 10. 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.

If $U1 \subseteq U2$ and $V2 \subseteq V1$ then we say that the computational problem *(U1,V1)* is larger than the computational problem *(U2,V2)*. 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 11. 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. 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 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 12. 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.

7. Example

The goal of this example will be to demonstrate the use of attribute models and the attribute evaluation process. Firstly, we define attribute models corresponding to the objects of problem domain, then give a computational problem and finally provide a solution as a result of attribute evaluation. The language for the specification of attribute models that includes equalities, structural relations, equations and preprogrammed functional dependences (see Section 2) will be extended with declarations of components as used in textual specifications of schemes (Section 5) i.e. declaration of component(s) has the form <*type*> <*idenitfier*>,.... We are going to use only textual specifications of attribute models from now on.

The problem domain for this example is direct current circuits. Let us define an attribute model of resistor, using the Ohm's law, by the following text:

```
Res:
numeric u, u1, u2, r, i;
u = u2-u1;                                  (1)
u = i*r;                                    (2)
p1 = (u1, i);                               (3)
p2 = (u2, i);                               (4)
```

Let us have a scheme with two resistors `R1` and `R2` connected that have resistances 6 and 12, as well as the current R1.i=0.5 and potential R1.u1=0. This is expressed by the specification:

```
Res R1, R2;
R1.p2=R2.p1;                                (5)
R1.r=6;                                     (6)
R2.r=12;                                    (7)
R1.i=0.5;                                   (8)
R1.u1=0;                                    (9)
```

The attribute model of this scheme is a composition of two attribute models of resistors. It is shown in Figure 2 as a bipartite graph, including 13 attribute dependencies. If we decide to show the attribute model in the flattened form, we have to draw 28 functional dependencies. According to the deep semantics DS2 one can calculate values of all attributes. This can be demonstrated by value propagation on the attribute model. One of the possible sequences of application of functional dependencies is `9[R1.u1=0], 6[R1.r=6], 7[R2.r=12], 8[R1.i=0.5], 5[R2.i=R1.i], 2[R1.u=(R1.i*R1.r], 1[R1.u2=R1.u+R1.u1], 2[R2.u=R2.i*R2.r], 5[R2.u1=R1.u2], 1[R2.u2=R2.u+R2.u1],`

where `5[R2.i=R1.i]` denotes application of the functional dependency `R2.i=R1.i` after flattening of the attribute dependency number 5, etc.

8. Implementation

The semantics of visual languages defined above has been implemented in a tool CoCoViLa [10], [11]. From a user's point of view this framework consists of two components: *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.

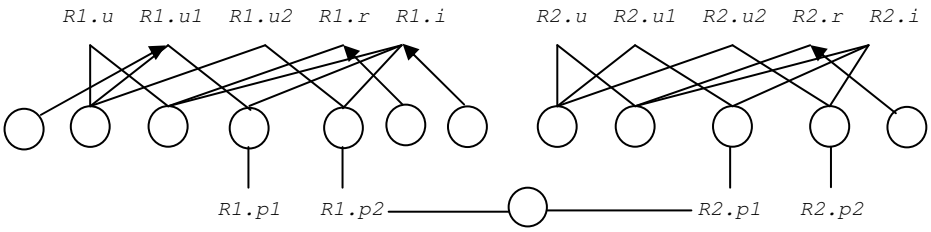

Figure 2. The composition of two attribute models of resistors

The *Scheme Editor* is a tool for usage of visual languages, i.e. developing schemes, compiling and running programs. It includes an attribute evaluator and implements all three deep semantics defined in the present paper. The *Scheme Editor* 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 draw, edit and use schemes in computations through language-specific menus and toolbars.

9. Example continued

In order to demonstrate CoCoViLa, we extend the example from the section 7 by adding two more concepts into our language of circuits – parallel and series connections. The specifications of attribute models 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);
    }@*/
}
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);
    }@*/
}
```

Having developed the visual language we are able to load it in the Scheme Editor and build schemes by putting visual objects on the drawing canvas and connecting them through ports. Figure 3 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.

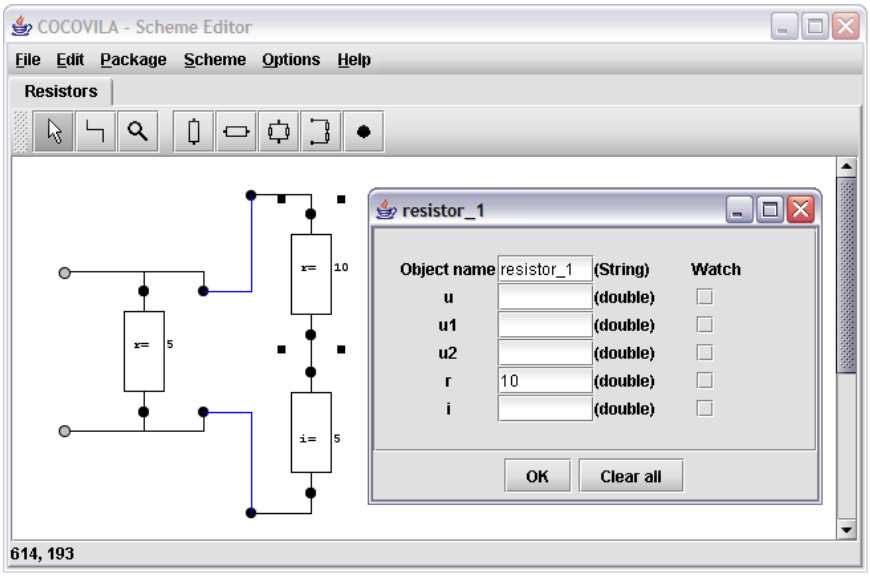

Figure 3. A scheme

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 attribute evaluator, an algorithm containing 32 computational steps is produced. This algorithm corresponds to the deep semantics DS2 of the scheme, namely it solves the largest solvable problem on the scheme. Figure 4 shows the results of applying this algorithm as a visual feedback in the Scheme Editor window, as well as values of attributes of resistor_1 in the pop-up window.

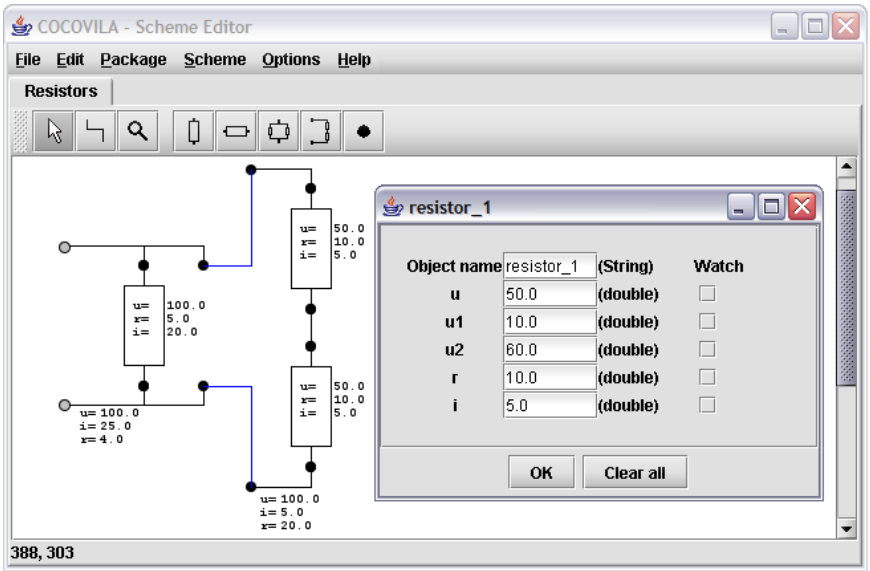

Figure 4. A scheme after evaluation

10. Related work

There are a number of works where attributes are associated with nodes of a graph. Götler [2] presents a formalization of the notion *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. [1] 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 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. We use the similar technique in the construction of schemes, i.e. boxes are visual components with ports, and cables are the bindings.

Mandayam [5] introduces PDL as performance modeling language based on Attributed Nodes-Only Grammars. ANGs are a subset of graph grammars that are extended with attributes and their evaluation rules. 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 [1], design objects in PDL are *modules*, *carriers* and *ports*. Carriers represent wires connecting modules using ports. 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 is order to determine an evaluation sequence is presented.

The NUT system [9] 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.

11. Conclusions

In this work we have proposed precise definitions of three kinds of semantics of schemes that constitute a wide class of visual languages. We have used attribute models in order to attach meaning to schemes. We have extended the attribute models by introducing a higher-order feature – subtasks as inputs of attribute dependencies. The subtasks require automatic synthesis of algorithms for their solution. The semantics of

schemes is implemented by using a dynamic attribute evaluator that works as an automatic program synthesizer. This work extends the results in attributed graphs in two ways: first, by introducing precise semantics of schemes, and second, by extending the attribute models. Our semantics of schemes has been implemented in a software tool CoCoViLa that is publicly available from *http://www.cs.ioc.ee/~cocovila/.* Finally, it has to be said some words about scalability of our method. Thanks to the linear time complexity of value propagation, the scalability of the attribute semantics without higher-order attribute relations is excellent. We have practically solved computational problems on attribute models with thousands of attribute dependences. In the general case, the search time depends very much on the number of higher-order attribute dependences in the synthesized attribute evaluation algorithm.

Acknowledgements. This work has been supported by the Estonian Science Foundation grant No. 6886.

References

1. Alpern, B., Carle, A., Rosen, B., Sweeney, P., Zadeck, K. *Graph Attribution as a Specification Paradigm.* ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 121—129, 1988.
2. Götler, H. *Attributed Graph Grammars for Graphics*. Graph Attribution and their Application to Computer Science, vol. 153 of LNCS, pp. 130-142, Stringer-Verlag, 1983.
3. Harf M., Tyugu E. Algorithms for Structural Synthesis of Programs. Programming and Computer Software, No.4, pp. 3 – 11, 1980.
4. Knuth, D. *Semantics of Context-free Grammars*. Mathematical Systems Theory, vol 2, pp. 127-145, 1968.
5. Mandayam R. *Performance Modeling of VLSI Systems*. PhD thesis, University of Cincinnati, Cincinnati, OH, 1994.
6. Penjam, J. *Computational and Attribute Models of Formal Languages*. Theoretical Computer Science, vol. 71, pp. 241 – 264, 1990.
7. Tyugu E. *Attribute Models Of Design Objects*. Proceedings IFIP TC 5 / WG 5.2 Workshop on Formal Design Methods for CAD, Tallinn, Estonia, pp. 16-19, 1993.
8. Tyugu E., Uustalu T. *Higher-Order Functional Constraint Networks*. Constraint Programming. NATO ASI Series F : Computer and Systems Sciences, Vol. 131, Springer-Verlag, pp. 116-139, 1994
9. Tyugu E., Valt R. *Visual programming in NUT*. Journal of visual languages and programming, Vol. 8, pp. 523 – 544, 1997.
10. Grigorenko P., Saabas A., Tyugu E. *Visual Tool for Generative Programming*. Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13). ACM Publ., p. 249 – 252, 2005.
11. Grigorenko P., Saabas A., Tyugu E. *COCOVILA – Compiler-Compiler for Visual Languages.* J. Boyland, G. Hedin. Fifth Workshop on Language Descriptions Tools and Applications *LDTA2005.* ETAPS, pp. 101 – 105, 2005.