Computing and Informatics, Vol. 29, 2010, 251–280

HIGHER-ORDER ATTRIBUTE SEMANTICS OF FLAT DECLARATIVE LANGUAGES

Pavel Grigorenko, Enn Tyugu

Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21
12618 Tallinn, Estonia
e-mail: {pavelg, tyugu}@cs.ioc.ee

Manuscript received 19 August 2008; revised 11 May 2009
Communicated by Ulrich Eisenecker

Abstract. A technique is described that provides a convenient instrument for implementation of semantics of simple declarative languages called flat languages. Semantics of a specification is defined in the paper as a set of programs derivable for solvable goals. We introduce higher-order attribute models that include more control information than conventional attribute models and explain the algorithm for dynamic evaluation of attributes on these models. A visual tool CoCoViLa is briefly described as an instrument for implementing attribute semantics of flat languages.

Keywords: Higher-order attribute models, flat languages, attribute semantics of declarative languages, synthesis of programs, domain specific languages

Mathematics Subject Classification 2000: 03B60, 68N19, 68T15, 68T30, 68T35, 68U20

1 INTRODUCTION

After Donald Knuth introduced attribute grammars for precise representation of semantics of programming languages in 1968 [7], they have become widely used tools in compiler construction. However, this technique is seldom applied to processing declarative domain specific languages, and if applied, it works only for a part

of the semantics of a language. The reasons are usually a weak syntactic structure of a domain specific language, and a great variety of domain specific semantic functions whose reuse in different domains is impossible. This has prevented the development of widely applicable semantic tools for declarative domain specific languages.

Our goal is to define a class of declarative languages that will have well-defined semantic properties, to propose a theoretically sound method of implementation of semantics of these languages, and to test it in practice. In the present work we first define a class of simple declarative languages – flat languages. Second, we describe a method of implementation of semantics of flat languages and show its practical applicability. The technique is an extension of attribute methods by explicitly introducing attributes of higher order. An essential contribution of the paper is the explanation of a method of evaluation of higher-order attributes in terms of maximal linear branches of the synthesized algorithm in Section 8. In a declarative language, like in Prolog, one may need a goal in order to get a program from a specification. We also use goals in addition to specifications. Unlike semantics of imperative languages, our semantics provides a set of programs for a specification – a program for every solvable goal.

Let us consider attribute semantics of a traditional programming language as defined originally by Knuth [7] and explained in terms of attribute models by Penjam [12]. If we look at an attribute model of a production of the language, or at an attribute model of a syntax tree of a text written in this language, we can see that it is just a collection of variables bound by functional dependencies. In other words, it is a functional constraint network representing the meaning of a production or a text. There is little control information in it. We call it a simple attribute model. In the present work we extend attribute models by allowing attribute dependencies to be, besides functional dependencies, also higher-order functional dependencies. This gives us a possibility to express more control of computations in an attribute model itself. Second, we consider declarative languages where a text can be not only a specification of a single program, but also a description of a device or a system (its model) that allows to ask several different questions about the specified thing. This means that a program can be obtained from a declarative specification and a goal (that is, a problem statement describing what is needed). We have restricted the set of specification languages considered here to structurally very simple languages that we call flat languages.

The present paper is organized as follows. In the next section we give a formal definition of the flat languages and bring out two examples. Then we introduce a core language and its standard extension. We discuss simple attribute models and attribute evaluation on them in Sections 5. In Section 6 we introduce attribute semantics of the core language. After that we introduce the central results of the work – higher-order attribute models and evaluation of attributes on them in Sections 7 and 8. An implementation of our approach to attribute semantics of flat languages is described in Section 9, and examples are given in Section 10. Related work is discussed in Section 11.

2 FLAT LANGUAGES

A *flat language* is a declarative language suitable for composing typed objects into descriptions of concepts and/or systems (a mathematical model in a broad sense) by connecting their components by equalities. We call it flat, because it has very little syntactic structure, in particular it does not contain explicit looping and branching statements. The meaning of a text in a flat language is hidden in the classes of objects and in the way the objects are connected. An inheritance relation is defined on types: a type is a subtype of another type that is called its supertype if and only if the objects of this type have all properties of the latter type. This is single inheritance without overriding. A connection is allowed only between two components whose types are the same or have one and the same supertype.

We define *flat languages* as follows. A finite set of *primitive types* and a countable set of *names* are given.

New *types* are defined by a construction

$$a : (a_1 : s_1, \ldots, a_k : s_k),$$

where a is a name given to a new type. The construction $a_i : s_i$ defines a *component* of a, where a_i is a unique name of a component, i.e. $a_i \neq a_j$ for $i \neq j$, and s_i is a name of a known type, $i = 1, \ldots, k, j = 1, \ldots, k$. We can refer to a component a_i of a as $a.a_i$. The new types constructed this way are called *compound types*. In the type theory such typing is called *nominal* [14], where types as well as their components are always defined with names.

We define ports that are connection points of typed objects as follows. Let us have a type b that has a component $b.a_i$. Then

$$(b.a_i, x)$$

defines a *port* with a name x and representing a component $b.a_i$ of b. The type of a port is equal to the type of a component it represents.

Types of objects in a flat language are represented by classes. A class includes a compound type as a part of it. A *class* is defined by a construction

$$c : (s, (p_1, \ldots, p_m), sem),$$

where c is the name of a class, s is its type, $p_1, \ldots, p_m$ are ports representing some components of s, and sem is a definition of the local semantics of the class that can be defined only in terms of s, i.e. using only the components of s. Local semantics sem is not defined here, it depends on a concrete flat language. There are no generally prescribed means for defining sem; however, we assume that sem is always defined computationally.

A text in a flat language is a declarative specification of an object, a model or a process. *Text* is a sequence of *statements*. The syntax of a flat language is given

by a set of classes and by the following simple rules in EBNF[1] as follows:

$$\begin{aligned} Text &::= \{Statement;\} \\ Statement &::= Object|Binding \\ Object &::= ClassName\ ObjectName \\ Binding &::= ObjectName.PortName{=}ObjectName.PortName. \end{aligned}$$

Ports in a binding must have one and the same type (or common supertypes in the case of existence of inheritance which is allowed, but not discussed here).

Semantics of a flat language depends only on the semantics of its classes and on the semantics of bindings. (This is the flatness!) Semantics of bindings is as follows:

a) if ports in a binding have a primitive type, then the components bound by the binding are in every aspect (except their names) the same;

b) if ports in a binding have a compound type, then a binding is recursively defined also for all pairs of the respective components of a type. Example: the binding $p.x = q.y$ for the ports x and y that have the type defined as $a : (a_1 : s_1, \ldots, a_k : s_k)$ defines also the bindings $p.x.a_1 = q.y.a_1, \ldots, p.x.a_k = q.y.a_k$ as well as bindings for all components of the types of $a_1, \ldots, a_k$.

Let us look at a small example of a flat language that is for specifying reliability of devices through the reliabilities of their components and the structure of a device. Types are *double*, *Basic*, *Parallel*, *Series*. The type *double* is a primitive type for numbers. The type *Basic* represents components of a device that have a value of reliability, let it be a probability of a correct operation of a device or a component during a given period of time. This probability is represented by a variable p that is of the type *double* and is a component of an object of type *Basic*. Let us introduce also another variable q that expresses a probability of error occurring during the given period of time. The type *Parallel* represents a substructure of a device that is composed of two parts denoted by *part1* and *part2* in such a way that if at least one part works correctly, then the composition works correctly. These parts are of type *Basic*. The type *Series* represents a substructure of a device that is composed of two parts denoted by *part1* and *part2* in such a way that it operates correctly if both parts operate correctly. These parts are of type *Basic*. This is a rather typical example of a small domain-specific language in engineering. There are some computations that can be performed on the objects of type *Parallel* and *Series*. These computations and the notations introduced above are summarized in Table 1.

Some words have to be said about the Computations column in Table 1. We see equations there, and one may expect that these equations can be used in different ways, not only for computing the value of a variable on the left side of an equation.

[1] For brevity, here and in further sections we use quotation marks for specifying terminal symbols only when it is required to distinguish them from the syntax of EBNF.

Name of type	Supertype	Components	Computations	Comment
double				primitive numeric type
Basic		*double p* *double q*	$p + q = 1$	
Parallel	*Basic*	*Basic part1* *Basic part2*	$q = part1.q * part2.q$	
Series	*Basic*	*Basic part1* *Basic part2*	$p = part1.p * part2.p$	

Table 1. An example of a flat language

This column presents a computational semantics of the types to a user not interested in an implementation of the language. The implementation can be made in several ways. In particular, one could extract two assignments from the equation for *Basic*:

p:=1-q

q:=1-p,

or one can use a numeric equation solver for solving the equation. We postpone the discussion of the implementation now and return to it in Section 10.

An example written in the language of reliability is as follows:

```
Basic c1, c2, c3;
c1.p=0.99;
c2.p=0.97;
Series sr;
sr.part1=c1;
sr.part2=c2;
Parallel pr;
pr.part1=sr;
pr.part2=c3;
```

This is a description of five objects `c1`, `c2`, `c3`, `sr`, `pr` connected by means of 4 equalities, and assignment of values to primitive objects `c1.p`, `c2.p` that are components of objects `c1` and `c2`. This specification can be presented visually, as soon as one has means to represent objects of types *Basic*, *Parallel*, *Series*, as well as a possibility to connect ports and to introduce values of primitive types. A visual representation of the specification is shown in Figure 1. This text (or the scheme in Figure 1) can be used as a specification of several computations. A computation is defined as soon as a goal is given (cf. Prolog). A *goal* states the input and the output of a computation. For example, the following two goals are interesting (although there are more meaningful goals):

```
c3.p->pr.p
```

and

```
pr.p->c3.q.
```

The first goal requires finding the resulting value `pr.p` of reliability of the device `pr` depending on the reliability value `c3.p` of the component `c3`, and the second requires finding the probability of error `c3.q` of the component `c3` that gives the given reliability of device `pr`.

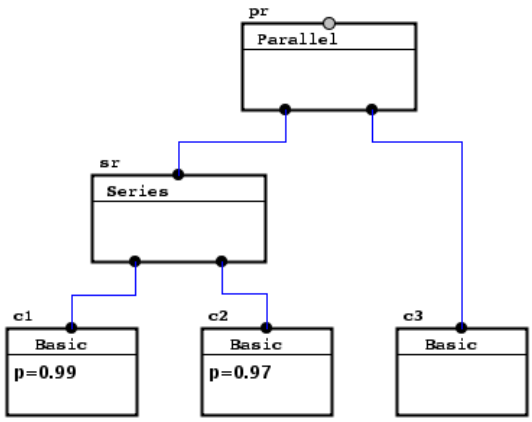


Fig. 1. Visual specification in a flat language

A more interesting flat language is a language of attribute semantics of a context free language. Let us have a context free language with a syntax given by rules of the form

$$p_0 \leftarrow w_1 p_1 w_2 p_2 \ldots w_k p_k$$

where $p_0, \ldots, p_k$ are nonterminal symbols of the language and $w_1, \ldots, w_k$ are possibly empty sequences of terminal symbols. Let the semantics of the language be given as an attribute grammar. First, we introduce a type tp for each nonterminal symbol p that includes the attributes of the symbol p as its components. Then we introduce types for rules as follows. A type of a rule r has components $c_0, \ldots, c_k$ with respective types $tp_0, \ldots, tp_k$ for its nonterminal symbols $p_0, \ldots, p_k$. Beside that, a type of rule r includes attribute dependencies as defined in the attribute grammar. This language is a flat language. By the definition of an attribute grammar, this language is sufficient for expressing attribute semantics of any sentence of the given language. A specification in the flat language, let us call it an attribute model, can be built from an abstract syntax tree of a text [12]. This specification can be used for computing the synthesized attribute of the nonterminal symbol representing the whole text. This is a dynamic evaluation of attributes of a language given by an attribute grammar. In order to be able to express attribute semantics of a language by means of one single specification in a flat language and to compose a static attribute evaluation algorithm, we have to extend the expressive power of types. This will be done later in the present paper.

Good examples of flat languages are visual languages where specifications are schemes [4], e.g. the language of class diagrams of UML and many simulation languages. Even many popular special purpose specification languages like, for instance, VHDL are in essence flat languages, although they have some features that are difficult to express through the local semantics of objects.

3 CORE LANGUAGE

In the present section we give syntax and intuitive semantics of a concrete flat language that can be used for two purposes:

- As a program specification together with some goal.
- For specifying a new type.

This language includes more features than presented in the example given in Section 2. The main extension is the usage of programs as implementations of functional types. We assume that a concrete implementation of the core language is based on some programming language that we call a *base language*. We have an implementation of flat languages with Java as the base language, see Section 9.

Types in the core language are primitive types, compound types and functional types:

a primitive type is any type of the base language (including reference types of Java in our implementation);

a compound type is introduced by writing its specification in the core language. In the general terms of flat languages, a compound type of our core language is a class. In the core language, all components of a compound type are ports and semantics of a class is represented by functional dependencies;

a functional type is introduced by writing its specifying formula and it is used only once in a specification for creating a functional dependency.

Specifications in the core language are written using five kinds of statements in the following syntax:

$$Specification ::= ClassName\ {:}'\ ('[\{Inherit;\}]\{(Decl|Bind|Val|FuncDep);\}')'$$

1. *Declaration of object*:

   $Decl ::= Type\ Id$

   $Type ::= \texttt{any}|PrimitiveType|ClassName$

   This declaration specifies an object with a given $Type$, and its name given by the identifier Id. The object as well as its components are called *variables*, and they can be bound by functional dependencies. (A component `b` of an object `a` has a compound name `a.b`.) If `any` is written instead of a type, then type of the object remains undefined, and it must be determined by some binding in a specification later, i.e. `any` is a type variable.

2. *Binding of variables*:

   $Bind ::= Var = Var$

   $Var \ ::= Id[.Var]$ where variables must have the same type or the same supertype. A binding $x = y$ denotes a possibility to compute a value of a variable (x or y) in the case a value of another variable is known. A binding $x = y$ is extended to the respective components of x and y, i.e. if x and y have a component a then $x.a = y.a$ is introduced.

3. *Valuation*:

   $Val ::= Var = Const$ where $Const$ is an object of a primitive type.

4. *Functional dependency*:

$$\begin{aligned} FuncDep &::= (SFuncDep|HOFuncDep)'\{'Impl'\}' \\ SFuncDep &::= [VarList] \rightarrow Var \\ HOFuncDep &::= Subtask[\{,Subtask\}][,VarList] \ \rightarrow \ Var \\ Subtask &::= '['VarList \ \rightarrow \ VarList']' \\ VarList &::= Var[,Var] \end{aligned}$$

   where the statement $SFuncDep$ has one arrow and specifies a functional dependency that represents computing a value of the variable on the right side (the result) from given values of variables on the left side (arguments). The statement $HOFuncDep$ has several arrows and specifies a higher-order functional dependency. In this case there are also arguments that have a functional type. These are the arguments in square brackets. These arguments are called *subtasks*. More about subtasks is written in Section 7.

   $Impl$ is a name of a program (a Java method in our implementation). This program is an implementation of the functional dependency specified by this statement. The type given by a functional dependency must be the type of the program given by its corresponding implementation, in particular, in the case of a functional dependency with functional arguments the program must also have procedural parameters of appropriate types.

5. *Inheritance*:

   $Inherit ::=$ `super` $Name$

   This statement defines a simple inheritance. If such a statement appears in a specification of some type, the specification of a type under the $Name$ is inlined instead of this statement. There can be only one statement of this kind in a specification. No overriding is permitted.

Bindings, valuations and functional dependencies bind variables and can be used for computations by means of value propagation, see Section 5. This is the intuitive semantics of a specification. In order to specify a computation one has to give also a goal, i.e. specify what should be computed.

A collection of statements is a specification of a type or it is a specification of a program, if a goal is given. An example of a specification has been presented in Section 2 as a specification of reliability of a device. (Note that the declarations of components `c1`, `c2`, `c3` are written in one statement for brevity there: `Basic c1, c2, c3;`).

4 STANDARD EXTENSION OF THE CORE LANGUAGE

There is a standard extension of the core language that can be easily translated in the latter by presenting new statements of the language as collections of statements of the core language. Defining a new flat language is defining new types of objects in the extended core language, where the syntax will remain unchanged otherwise.

The standard extension of the core language includes the following new statements:

1. *Alias*:
   $Alias ::= \texttt{alias}\ ['('Type')']\ Id =' ('VarList')'$

Alias defines a new variable with the name Id. This variable is a tuple of variables listed in the parentheses.

1. *Alias with a wildcard*:

$$AliasW ::= \texttt{alias}\ ['('Type')']\ Id =' (' Wildcard ')'$$
$$Wildcard ::= *.Id$$

Alias with a wildcard is a variable whose list of components depends on the particular specification where such statement occurs. This list includes all variables of the components defined in the same specification and names of such variables are equal to the identifier specified in $Wildcard$. The order of components in the list is not predefined, but it remains fixed during the computations. This alias is used for distributing and collecting some values for all objects defined in a specification.

1. *Equation*:
   $Equation ::= AExpr\ =\ AExpr$

Equation is a shorthand for a set of functional dependencies that can be derived from it. For example, `I = U*R;` will denote three functional dependencies: $I, U \to R\{f_1\}$; $U, R \to I\{f_2\}$; $I, R \to U\{f_3\}$; with the corresponding implementations derivable from the given equation: $f_1 : \texttt{R} = \texttt{I}/\texttt{U}$; $f_2 : \texttt{I} = \texttt{U} * \texttt{R}$; $f_3 : \texttt{U} = \texttt{I}/\texttt{R}$. We keep open the class of arithmetic expressions that can be used in equations, because this depends on the power of an equation solver for a particular flat language. (In our implementation based on Java [3], the equations are solved analytically and are restricted in such a way that they cannot include occurrences of a variable simultaneously on both sides of an equation. Beside arithmetic operations, an equation may include only functions implemented in `java.lang.Math` class.)

5 SIMPLE ATTRIBUTE MODELS AND EVALUATION OF ATTRIBUTES

In this section we give definitions of attributes, attribute dependencies and attribute models in a conventional way, not relating them to syntax of a flat language.

Attribute is a typed variable.

Simple attribute dependency is a functional dependency between attributes.

Let us use the following notation for a simple attribute dependency:

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

where f is a function of m arguments $x_1, \ldots, x_m$ computing a value of n-tuple $(y_1, \ldots, y_n)$, i.e. $x_1, \ldots, x_m$ are inputs and $(y_1, \ldots, y_n)$ are outputs of the attribute dependency. We say that the inputs and the outputs are bound by the attribute dependency.

Simple attribute model is a pair $\langle A, R\rangle$, where A is a finite set of attributes and R is a finite set of attribute dependencies binding these attributes.

Let U and V be two sets of attributes of an attribute model M. We denote by $U \to V$ a *computational problem* on the attribute model M, and say that U is a set of input attributes (or just inputs) and V is a set of output attributes (or outputs) of a computational problem. The computational problem states a goal that, given values of attributes from U, requires to find values of attributes of V using attribute dependencies of M.

If for two computational problems $U_1 \to V_1$ and $U_2 \to V_2$ we have $U_1 \subseteq U_2$ and $V_2 \subseteq V_1$, and at least one of these inclusions is strict, then we say that the computational problem $U_1 \to V_1$ is greater than the computational problem $U_2 \to V_2$.

We describe now a method that for a goal in the form of a computational problem $U \to V$ on a simple attribute model 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.

Value propagation is a procedure that, for a given attribute model M and a set of attributes $U \subset A$, decides which attributes are computable from U and produces a sequence of attribute dependencies. Constructed sequence of attribute dependencies becomes an algorithm for evaluating the attributes that are computable on the model M.

A well-known simple value propagation algorithm with linear time complexity (see [19], Section 4.3.2) works step by step as follows. At each step it tries to find an attribute dependency whose inputs are all known (initially given or computed) and some of outputs are not computed. In the positive case, the attribute dependency will be added to the algorithm being built and all its outputs will be added to the set of computed attributes. In the negative case (if there is no such attribute dependency and not all outputs of the problem have been found), the problem is

unsolvable. Initially the set of computed attributes equals to the set of given attributes U and the algorithm (i.e., the sequence of attribute dependencies) is empty.

The described method does not give a minimal algorithm for solving a problem in general – the algorithm may include steps that are unnecessary for solving the problem. There is a procedure that gives a minimal algorithm for solving a computational problem, see Section 4.3.4 in [19]. For each attribute dependency included in the algorithm, this optimization procedure checks, whether the computation of outputs of a problem depends on the application of a particular attribute dependency. If not, an attribute dependency is excluded from the algorithm. The procedure runs backwards starting from the end of the algorithm. This optimization procedure has also linear time complexity.

Example 1. Let us consider the following simple attribute model with eight attributes and six attribute dependencies:

$$\begin{aligned}\langle A = \{&a, b, c, d, e, f, g, h\},\\ R = \{&a \to c\{f_1\};\\ &a, e \to h\{f_2\};\\ &b, c \to d\{f_3\};\\ &d \to e\{f_4\};\\ &d, f \to g\{f_5\};\\ &h \to f\{f_6\}\}\rangle\end{aligned}$$

It is always possible to present an attribute model in the form of a directed bipartite graph, where arcs correspond to directed bindings between attributes and dependencies. Figure 2 shows it for the present example with additional labels $F_1, \ldots, F_6$ for attribute dependencies.

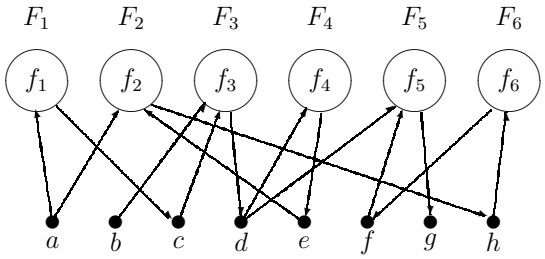


Fig. 2. Bipartite graph of a simple attribute model

Let us try to solve a computational problem $U : \{a, b, f\} \to V : \{g\}$ on a given attribute model. Value propagation procedure does not consider concrete values of attributes from the input set U, it just assumes that attributes are computable. For each attribute dependency value propagation stores a counter indicating the number

of unknown inputs. If an attribute is or becomes computable, counter is decreased for each dependency where such attribute is an input attribute. If for an attribute dependency a counter is zero, such dependency is added to the algorithm and its output attributes become computable. One of the outcomes of the value propagation in this example is a sequence $\{F_1, F_3, F_4, F_2, F_5, F_6\}$. It is clear that this algorithm is not minimal, i.e. some attribute dependencies used are not required for computing the output g. Once the optimization procedure is applied, the algorithm is reduced to $\{F_1, F_3, F_5\}$.

6 ATTRIBUTE SEMANTICS OF CORE LANGUAGE

Semantics of a specification in the core language is a set of all algorithms that can be composed on the attribute model of the specification for solving computational problems. In order to give attribute semantics of the core language, we have to

- define rules for translation of specifications into attribute models
- give a method for constructing an algorithm for any goal that describes a problem solvable on an attribute model.

An attribute model M of a specification is built stepwise in the following way. First, an empty model M is created. Thereafter, statements of the specification are translated and the translations added to M one by one as follows.

1. A declaration `T x` is translated into a new attribute x in the attribute model of the specification. If T is a class with a type $(a_1 : s_1, \ldots, a_k : s_k)$, then attributes $x.a_1, \ldots, x.a_k$ are added to the model M. Attribute dependencies binding the attributes according to the semantics of T are added to M. Processing of a declaration with the type variable `any` is postponed until the type will be known and then translated as described here.
2. A binding `x=y` is translated into new attribute dependencies $x \to y\{id\}$, $y \to x\{id\}$, as well as $x.z \to y.z\{id\}$, $y.z \to x.z\{id\}$ for each z that is a component of both types of x and y. (The implementation id must be of a suitable type for each attribute dependency!)
3. A valuation `x=v` is translated into an attribute dependency $\to x\{f\}$, where f denotes an assignment of v to x.
4. A functional dependency is added as a new attribute dependency to the attribute model M.
5. An inheritance `super x` is translated by finding the attribute model of x and adding all its variables and all its attribute dependencies to the model M. Any name clash is considered as a mistake.

A method of constructing an algorithm for solving a given goal has been described in the previous section for simple attribute models. This gives us the attribute semantics of the core language without higher-order attribute dependencies.

In the following sections we extend attribute models with higher-order attribute dependencies and describe a method of attribute evaluation in the general case.

7 HIGHER-ORDER ATTRIBUTE MODELS

Let us return to the example of reliability calculation again. Instead of calculating one reliability value, we may wish to calculate a table of reliabilities of a device depending on the given reliabilities of a component. In the presented example, one may wish to run the program for the goal `c3.p -> pr.p` several times, for instance, beginning from `0.80` with a step `0.05` to `0.99`. This cannot be specified in the core language with attribute models introduced above, but can be specified in the core language using a higher-order functional dependency. Indeed, let us have a program `tab` for tabulating a function that takes as input `from`, `step`, `to`, and produces a table with two columns – one for values of the argument and another for the respective values of the function. This can be represented in the core language by a higher-order functional dependency:

```
[funArg -> funVal], from, step, to -> table{tab};
```

The specifier of this functional dependency says exactly that having a program of computing one value of the function (`funVal`) from a given value of its argument (`funArg`), and having values of `from`, `step`, `to`, one can compute `table` by using the program `tab`. Let us notice that it is not expected that the program of computing `funVal` from `funArg` is given in a ready form. This program must be constructed on the basis of the specification that includes the higher-order functional dependency. However, the tabulation program `tab` must be preprogrammed. Our implementation of `tab` as Java method is presented in Section 10.2. This program is useful in many cases, and it can be used in a type `Table` that has the following specification:

```
double funArg, funVal, from, step, to;
String table;
[funArg -> funVal], from, step, to -> table{tab};
```

A specification of the whole problem will include, besides the specification given in Section 2, also the following lines now:

```
Table t;
t.funArg=c3.p;
t.funVal=pr.p;
t.from=0.80;
t.step=0.05;
t.to=0.99;
```

And the goal will be `->t.table;`

The type `Table` is in essence a control structure that prescribes performing computations in a particular way. One can introduce even more general control structures in a similar way, e.g. a **for**- or a **while**-loop and an **if-then-else** choice,

for instance. This will add the generality to attribute models. For example, a specification of a **while**-loop type with a preprogrammed loop control `whileProgram` can be

```
boolean binit, b;
any state, nextstate, initstate;
[state -> nextstate, b], binit, initstate -> state{whileProgram};
```

The subformula `[state -> nextstate, b]` prescribes the synthesis of a body S of a loop and computing of a loop control variable `b`. The whole program will be equivalent to **while** `b` **do** `S` **od;**

Here we have quite informally presented two examples of higher-order functional dependencies, and now we are going to define them precisely. Let A be a set of attributes and P a set of computational problems with inputs and outputs from A.

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

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 includes some higher-order attribute dependencies on the set of attributes A.

This extension makes a big difference in the following: higher-order attribute models are so expressive that enable to synthesize recursive, branching and cyclic programs where respective control structures, i.e. recursion, branching and loops are preprogrammed and represented as higher-order attribute dependencies (cf. the example in the beginning of this section). Detecting the solvability of a problem and synthesizing an algorithm on a higher-order attribute model has exponential time complexity with respect to the number of higher-order dependencies (see the remark in Section 8).

8 EVALUATION OF HIGHER-ORDER ATTRIBUTES

Often only one higher-order attribute dependency is used in a specification. The evaluation strategy is quite obvious in this case: first use only conventional attribute dependencies and at the end the higher-order one. Thereafter, if still needed, use simple attribute dependencies again. Time complexity of the search remains linear in this case with respect to the size of an attribute model.

In the general case, when an attribute model M contains several higher-order attribute dependencies, the strategy for construction an evaluation algorithm is as follows.

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 a *hoad* is applied, if it is applicable. A *hoad* 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 attribute dependencies obtained in this way is called

maximal linear branch (*mlb*). It contains one *hoad* at the end of the sequence. There are three possible outcomes of this procedure:

1. After constructing the *mlb* the problem is solvable (like in the case of a single *hoad*).
2. A *mlb* cannot be found and the problem is unsolvable.
3. A *mlb* can be found and the initial problem $U_1 \to V_1$ is reduced to a smaller one $U_2 \to V_2$, $U_2 = U_1 \cup Y$ and $V_2 = V_1 \backslash Y$, where Y is the set of outputs of the *hoad*.

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

It is important to notice that for applying a *hoad* we have to solve all its subtasks. This means that *the whole procedure of problem solving must be applied for every subtask*. 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 attribute 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 S for a subtask $[u_1, \ldots, u_m \to v_1, \ldots, v_n]$. Then a *hoad* has the form:

$$S_1, \ldots, S_r,\ x_1, \ldots, x_k \to\ y_1, \ldots, y_l\{f\},$$

where $x_s, y_t \in A$ are attributes and $S_j \in P$ are subtasks. Let us label *hoad*s with R_i, $i = 1, \ldots i_{max}$, where i_{max} is the number of all *hoad*s in the model.

Figure 3 shows a part of an *and-or* search tree for solving a problem S_0 on a higher-order attribute model. The *and*-nodes are the *hoad*s $R_\alpha, \ldots, R_\beta$ that can be applied first for solving a problem of their parent node. The successors of a *hoad* node R_α are its subtasks $S_{\alpha,1}, \ldots, S_{\alpha,m}$ and the reduced problem S'_α that remains to solve after applying the *hoad*. The search on the *and-or* tree is depth-first search with backtracking.

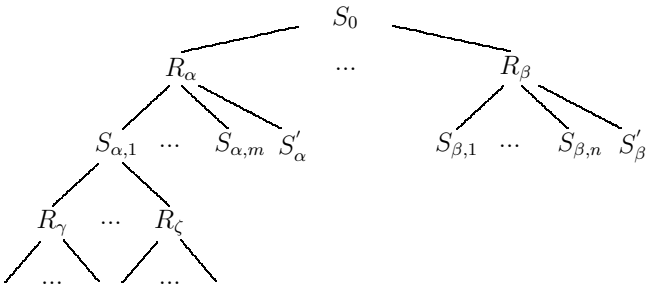


Fig. 3. *And-or* search tree for attribute evaluation on higher-order attribute model

Remark. It is useful to notice that types of attribute dependencies (and higher-order attribute dependencies) can be considered as propositional formulas where arrows denote implications and commas denote conjunctions. Building an attribute evaluation algorithm for a particular computational problem with inputs $u_1, \ldots, u_m$ and outputs $v_1, \ldots, v_n$ corresponds then to a derivation of the formula $u_1 \wedge \ldots \wedge u_m \supset v_1 \wedge \ldots \wedge v_n$ in the intuitionistic propositional calculus (IPC). This correspondence is known as the Curry-Howard isomorphism [5].

We would like to point out here that the evaluation of attributes considered in the present section is an efficient algorithm of program construction in propositional logic programming [10]. This approach gives also an algorithm of proof search for IPC, although some transformation of propositional formulas to the suitable form will be needed in the general case [8]. An unpleasant consequence of the fact that the proof search for IPC is PSPACE-complete [15], the higher-order attribute evaluation also has exponential time complexity.

Example 2. In this example we will show the problem solving on higher-order attribute models. Let us consider an attribute model with the following attribute dependencies:

$$[y \to a] \to b$$
$$[a \to b] \to x$$
$$x, y \to a$$

and a goal in the form of a computational problem with no inputs and one output $\{\to b\}$ on this model. In fact, solutions (there can be more than one) of this problem are equivalent to the derivation of a well-known Kripke's formula $((((A \to B) \to A) \to A) \to B) \to B$ which is an intuitionistic analog of the classically valid formula $((A \to B) \to A) \to A$ known as the Peirce's law. The idea of encoding arbitrary intuitionistic propositional formulae into sets of formulas with at most one subimplication on the left hand side of a outermost implication is described in [8].

For brevity, in this example we omit implementations of attribute dependencies and label dependencies as follows: $S_1 : [y \to a]$; $S_2 : [a \to b]$; $R_1 : S_1 \to b$; $R_2 : S_2 \to x$; $F_1 : x, y \to a$, where S_i is a subtask, R_j is a higher-order attribute dependency and F_k is a simple attribute dependency.

Figure 4 shows a complete and-or search tree for a solution of the computational problem. Let us explain the traversal step by step. The root of a tree S_0 is a top-level problem with a goal $\emptyset \to b$ with an empty set of inputs. First, value propagation is applied on S_0 returning an empty sequence of attribute dependencies, because the initial problem has no inputs. In order to find an *mlb*, a higher-order attribute dependency has to be considered. Thus R_1 is picked (first *hoad* in the model). The only input of R_1 is a subtask S_1 that has to be solved. S_1 has one input y and one output a. Again, value propagation is applied in the context of S_1 returning an empty sequence. In order to complete an *mlb* of S_1, a *hoad* has to be used again.

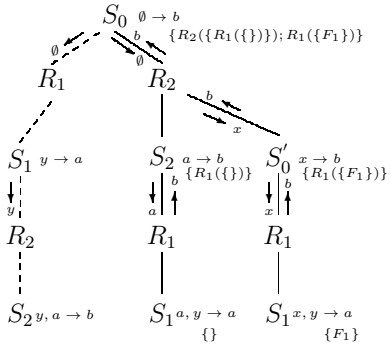


Fig. 4. Decorated and-or search tree of the example

R_1 is ignored, because it has just been tried, thus R_2 is picked. Subtask S_2 of R_2 has an input a and an output b. In addition, y from S_1 is also available. S_2 cannot be solved, because value propagation returns an empty sequence (cannot compute anything from y and a) and there are no other *hoad*s that can be used; consequently, *mlb* of S_2 cannot be constructed. We do backtracking to the node S_1 and then to the S_0 and mark the backtracked branch by dashed lines.

Back in S_0, second *hoad* R_2 is chosen. S_2 cannot be solved by value propagation, thus R_1 is used. S_1 is solved trivially (i.e. *mlb* of S_1 is empty), because an input a from S_2 maps to the output a of S_1. *Mlb* of S_2 is successfully constructed and its sequence contains one attribute dependency R_1. The output of R_1 is b and it is exactly what is needed to solve S_2. The subtask of R_2 is solved and an output x of R_2 becomes available. Here is a tricky part. First *mlb* of S_0 is complete with the following sequence of attribute dependencies: $\{R_2(\{R_1(\{\})\})\}$. Initial problem $S_0 : \emptyset \to b$ is not solved. Knowing x, S_0 is reduced (narrowed) to a new computational problem $S'_0 : x \to b$. Again, value propagation is applied on S'_0 returning an empty sequence. Then, R_1 is picked. In the context of S_1, x and y are known. The former is propagated from S'_0 and the latter is an input of S_1. The required output is a. Value propagation returns a sequence with one simple attribute dependency $\{F_1\}$, i.e. a becomes known after using F_1 with inputs x and y. The subtask S_1 is solved. The output of R_1 is b. After using R_1, the problem S'_0 is solved, consequently S_0 is solved. Finally, *mlb* of S'_0 is glued together with *mlb* of S_0 and the full algorithm of computing the goal b is as follows: $\{R_2(\{R_1(\{\})\}); R_1(\{F_1\})\}$.

9 IMPLEMENTATION

Our approach to attribute semantics of specifications has been implemented in a Java based tool CoCoViLa[2] [3]. This tool is used for the development and usage of domain-specific flat languages. In CoCoViLa, a specification in a flat language is presented visually as a scheme (e.g. Figure 5). A scheme language is a flat language

[2] Homepage: `http://www.cs.ioc.ee/cocovila`.

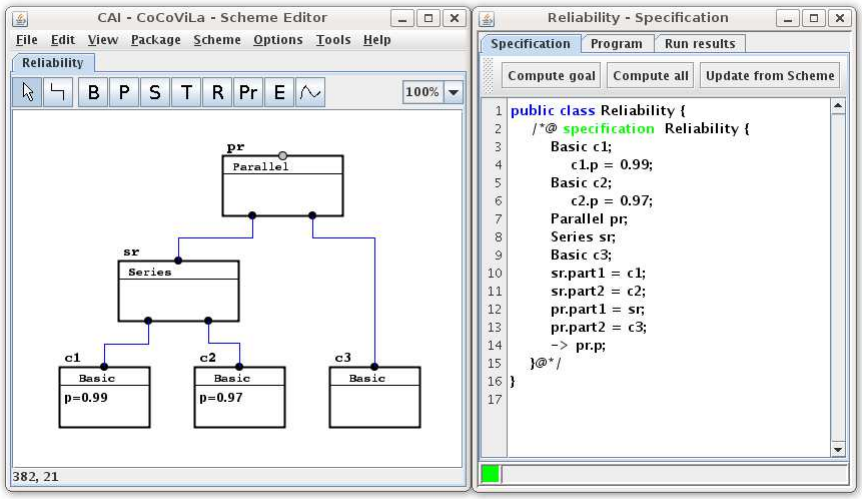


Fig. 5. CoCoViLa Scheme Editor and a specification window

where visual objects are bound by lines connecting ports that represent attributes of the objects. The scheme language is translated into the extended core language. Due to hierarchical specification of schemes, CoCoViLa is able to handle schemes with a large number of objects (including thousands of variables) and many subtasks.

Java classes annotated with specifications in a flat language are called *metaclasses*. A visual language is implemented as a package that is a set of components called *visual classes* which are metaclasses supplied with corresponding visual images. Attribute semantics of visual specifications (schemes) is implemented in such a way that an executable code for attribute evaluation is produced first and then the code is evaluated.

CoCoViLa consists of two runnables: the *Class Editor* and the *Scheme Editor*.

The *Class Editor* is used for implementing visual languages for different problem domains. This is done by defining attribute models of language components as well as their visual and interactive aspects.

The *Scheme Editor* is a tool for drawing schemes, compiling and running programs defined by a scheme and a goal. Algorithms for attribute evaluation are embedded into the *Scheme Editor* and not visible to the user. Figure 5 shows the *Scheme Editor* used for solving the problem of Reliability presented in Section 2.

CoCoViLa has been used for implementing a number of simulation packages for neural networks, electrical and logical circuits, analysis of mechanical drives, etc. Also, it has been used for educational purposes at Tallinn University of Technology.

10 EXAMPLES

We present first the implementations of two small examples in CoCoViLa that we have introduced in Section 2 and Section 7. This should explain how CoCoViLa works and how it uses the attribute semantics. The third example is a larger simulation package.

As it has already been mentioned in the previous section, the specification of components and problems in CoCoViLa are metaclasses (here we do not consider the visual representation) – Java classes annotated with specifications of attribute models. In a metaclass the specification is surrounded by the comment symbols /*@ and @*/ so that only CoCoViLa uses this information and the Java compiler ignores it.

The following three metaclasses: `Basic`, `Parallel` and `Series` constitute a simple Reliability package in CoCoViLa.

```
1 class Basic {
2     /*@ specification Basic {
3         double p, q;
4         p + q = 1;
5     } @*/
6 }
```

```
1 class Parallel extends Basic {
2     /*@ specification Parallel super Basic {
3         Basic part1, part2;
4         q = part1.q * part2.q;
5     } @*/
6 }
```

```
1 class Series extends Basic {
2     /*@ specification Series super Basic {
3         Basic part1, part2;
4         p=part1.p*part2.p;
5     } @*/
6 }
```

10.1 Simple Attribute Model

The description of a problem presented in Section 2 in CoCoViLa is the metaclass `Reliability` which is either written by the user or automatically generated by CoCoViLa from the visual scheme shown in Figure 1. (In the latter case the package has to contain visual classes as well.)

```
1 public class Reliability {
2     /*@ specification Reliability {
3         Basic c1;
```

```
4           c1.p = 0.99;
5           Basic c2;
6           c2.p = 0.97;
7           Parallel pr;
8           Series sr;
9           Basic c3;
10          sr.part1 = c1;
11          sr.part2 = c2;
12          pr.part1 = sr;
13          pr.part2 = c3;
14          c3.p -> pr.p;
15      } @*/
16  }
```

Line 14 in the specification of the `Reliability` states the goal – compute `pr.p` assuming that the value of `c3.p` is known.

CoCoViLa handles the given specification, parses and transforms it into the attribute model, invokes the value propagation on the model and produces the executable Java program that is the attribute evaluation algorithm:

```
1  public class Reliability implements IComputable {
2      public Basic c1 = new Basic();
3      public Basic c2 = new Basic();
4      public Parallel pr = new Parallel();
5      public Series sr = new Series();
6      public Basic c3 = new Basic();
7
8      public void compute( Object... args ) {
9          c3.p = ((Double)args[0]).doubleValue();
10         c1.p= 0.99;
11         c2.p= 0.97;
12         sr.part1.p = c1.p;
13         sr.part2.p = c2.p;
14         c3.q=( 1-c3.p);
15         sr.p= (sr.part1.p * sr.part2.p);
16         pr.part2.q = c3.q;
17         sr.q=( 1-sr.p);
18         pr.part1.q = sr.q;
19         pr.q= (pr.part1.q * pr.part2.q);
20         pr.p= 1-pr.q;
21     }
22 }
```

The generated Java program is an ordinary Java class where the specification has been replaced by variable declarations and a method containing the attribute evaluation algorithm. The `Reliability` class implements the `IComputable` interface from the CoCoViLa API which contains a single method `compute()`:

```
public interface IComputable {
    public void compute( Object... args );
}
```

After invoking the `compute()` method and giving the value c3.p = 0.85 (the system displays a corresponding dialog box), the following list of computed attributes is returned:

```
c1.p = 0.99
c2.p = 0.97
c3.p = 0.85
c3.q = 0.15
sr.p = 0.9603
sr.part1.p = 0.99
sr.part2.p = 0.97
sr.q = 0.0397
pr.part1.q = 0.0397
pr.part2.q = 0.15
pr.q = 0.005955
pr.p = 0.994045
```

The last line contains the desired result.

Only the specifications (visual or textual) and results are visible to the user in a usual case. The other steps will be invisible, unless one explicitly requests CoCoViLa to show them. The textual specification of a problem can be obtained from a visual representation, for example from the scheme in Figure 1 or written manually.

10.2 Higher-Order Attribute Model

In this section we demonstrate the usage of a higher-order attribute model in CoCoViLa.

We introduce a metaclass `Table`, whose specification has been presented in Section 7. `Table` consists not only of a specification of its attribute model, but also contains a method `tab()` with four arguments. Line 5 in `Table` includes a higher-order attribute dependency which can be read as follows – "synthesize an algorithm for computing the value of the attribute `funVal` from `funArg` and use it to compute the value of the attribute `table` from the attributes `from`, `step` and `to`". `[funArg -> funVal]` is a subtask. There can be more than one subtask in a higher-order attribute dependency and all of the subtasks have to be solved to make such dependency applicable. An algorithm for solving a subtask may require the usage of other higher-order attribute dependencies as we have shown in Section 8.

```
1 public class Table {
2     /*@ specification Table {
3         double funArg, funVal, from, step, to;
4         String table;
```

```
5           [funArg -> funVal], from, step, to -> table{tab};
6       } @*/
7
8       public String tab( Subtask st,
9                          double from, double step, double to ) {
10          String result = "\n";
11          try {
12              for ( double i = from; i <= to; i += step ) {
13                  Object[] out = st.run( new Object[] { i } );
14                  result = result + "i=" + i
15                                  + ",␣out=" + out[0] + "\n";
16              }
17
18          } catch ( Exception e ) {
19              e.printStackTrace();
20          }
21          return result;
22      }
23  }
```

Let us have a look at the realization of the higher-order attribute dependency. The identifier `tab` in curly brackets shows that there is a Java method with this name in the same metaclass that implements the given dependency. Arguments of the method `tab()` correspond to the inputs of the attribute dependency. The first input of the dependency is a subtask, i.e. one has to pass to the method `tab()` the algorithm for solving the subtask. Java does not support pointers to functions, so another approach had to be taken – the first parameter of the method `tab()` is an object `st` of type `Subtask`, where `Subtask` is the interface from CoCoViLa's API with one method `run()` that must solve the given subtask:

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

Other three inputs of the higher-order attribute dependency in `Table` are ordinary variables of primitive type `double` represented by the respective parameters of the method `tab()` (the names of the inputs of a dependency in a specification and the names of the respective parameters of its method do not have to be the same). The body of the method `tab()` contains a for-loop which iterates a local variable `i` beginning with the value defined by the variable `from` with the increment `step` until `to`. The value of `i` is passed as a single argument to the method `run()` of object `st` (see Line 13). The method `run()` returns a value that is used for constructing a string `result` that is the expected table. The method `tab()` returns the `result`.

Now we have to extend the specification of our reliability problem. The extended specification of `Reliability` is the following:

```
1  public class Reliability {
2      /*@ specification  Reliability {
3          ...here are the lines 3-13
4              of the metaclass Reliability from Section 10.1
5          Table t;
6          t.funArg=c3.p;
7          t.funVal=pr.p;
8          t.from=0.80;
9          t.step=0.05;
10         t.to=0.99;
11         -> t.table;
12     } @*/
13 }
```

The variable `t.funArg` is bound by an equality with `c3.p` and `t.funVal` is bound with `pr.p`. This means that it is possible to solve the subtask `[funArg -> funVal]` if it is possible to compute `pr.p` from `c3.p`. In Section 10.1 we have shown that `c3.p->pr.p` is solvable. In the current specification the goal is to compute the value of the variable `t.table` (Line 10).

The program synthesized from the given specification is shown below. The first ten lines of the body of the `compute()` method are value assignments and some computations that do not depend on the value of `c3.p`. Then there is a nested class `Subtask_0` which implements the `Subtask` interface. The body of the `run()` method of the class `Subtask_0` is the algorithm for computing `t.funVal` from `t.funArg` which is given as an input of this subtask. An instance `subtask_0` of `Subtask_0` class is created and passed as the first argument of the `tab()` method of the class `Table`. Note that inside `run()` some values computed outside of the `Subtask_0` are used (in order not to change the environment outside the subtask, copies of the respective variables are made inside the `Subtask_0`).

```
1  public class Reliability implements IComputable {
2
3      //...(omitted variable declarations for c1,c3,c3,sr,pr,t)
4
5      public void compute( Object... args ) {
6          c1.p = 0.99;
7          c2.p = 0.97;
8          sr.part1.p = c1.p;
9          sr.part2.p = c2.p;
10         sr.p = ( sr.part1.p * sr.part2.p );
11         sr.q = ( 1 - sr.p );
12         pr.part1.q = sr.q;
13         t.from = 0.80;
14         t.step = 0.05;
15         t.to = 0.99;
16
17         class Subtask_0 implements Subtask {
```

```
18              //...(omitted the constructor
19              //    and new declarations for c3,pr,t)
20              public Object[] run( Object[] in )
21                                          throws Exception {
22                  t.funArg = ((Double)in[0]).doubleValue();
23                  c3.p = t.funArg;
24                  c3.q = ( 1 - c3.p );
25                  pr.part2.q = c3.q;
26                  pr.q = ( pr.part1.q * pr.part2.q );
27                  pr.p = 1 - pr.q;
28                  t.funVal = pr.p;
29                  return new Object[] { t.funVal };
30              }
31          }
32          Subtask_0 subtask_0 = new Subtask_0();
33          t.table = t.tab( subtask_0, t.from, t.step, t.to );
34      }
35  }
```

Invoking the `compute()` method returns the desired result (value of `t.table`):

```
t.table =
i=0.80, out=0.992060
i=0.85, out=0.994045
i=0.90, out=0.996030
i=0.95, out=0.998015
```

10.3 Large-Scale Simulation

Figure 6 shows the usage of one of the largest applications of CoCoViLa – a package for modeling and simulation of hydraulic-mechanical load-sensing systems [2]. In principle, this package and its operation is similar to the example in Section 10.2. The only difference is in the complexity of an attribute model and Java classes. However, the role of higher-order attribute dependencies is more important in solving complex problems like the hydraulics problem here.

As it has been noted in the beginning of the paper, declarative flat languages have a weak syntactic structure, in particular, the structure of a specification does not represent the structure of a synthesized program. Simple attribute models (and conventional attribute models of syntactic rules) enable to construct only linear branches of programs. From the other side, a program for solving the simulation problem in Figure 6 must have a complex structure – nested loops, etc. The required control structures of a synthesized program are provided by the higher order attribute dependencies. Their subtasks specify newly synthesized blocks that are run as prescribed in a higher-order attribute dependency. (See the usage of a subtask in a loop in the example in Section 10.2). Let us note that any iterative method using one loop, including Runge-Kutta that is needed in the present case, can be

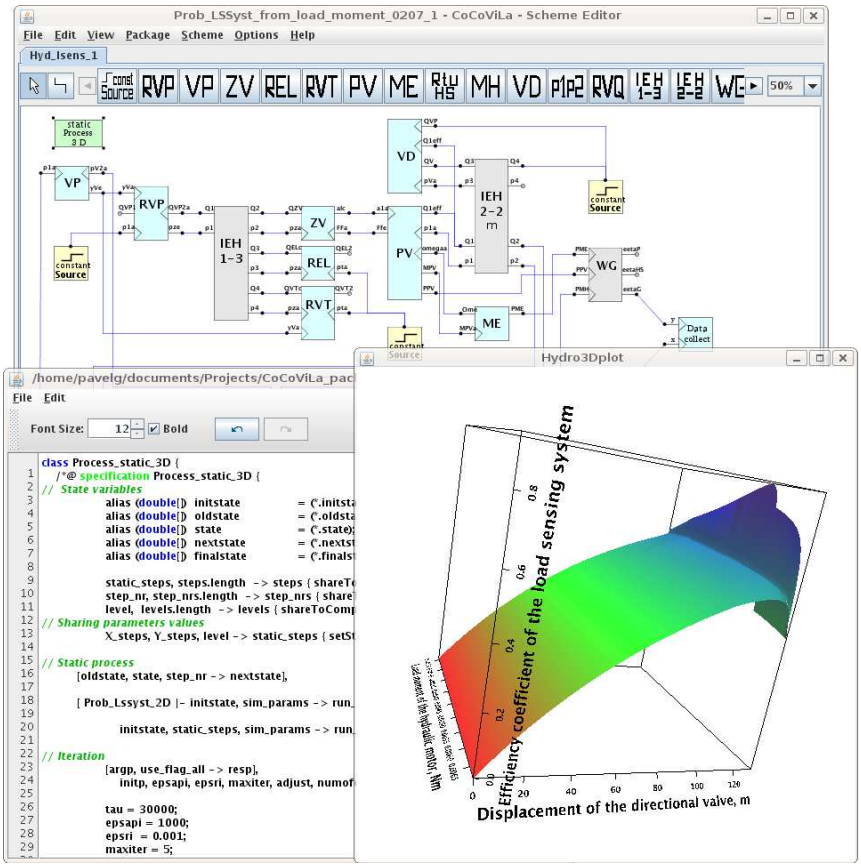


Fig. 6. A scheme with hydraulic-mechanical load-sensing system and the result of simulation

implemented in CoCoViLa using a subtask in the same way as in the class Table in Section 10.2 (note that it would be impossible to implement a looping algorithm without a subtask). For the present hydraulics problem, the synthesized Java program has almost 5k lines of code, includes loops with nestedness 4, and includes other control structures also implemented with subtasks.

Modeling and simulation of hydraulic systems (e.g. automatically regulated fluid power systems of stationary and mobile machines, steering mechanisms of cars and ships, drives of robots, etc.) has been investigated in Tallinn University of Technology for several decades. About 40 hydraulic elements (hydraulic motor, pump, resistors, volume elasticities, tubes, interface elements, etc.) from older packages for modeling and simulating hydraulic systems have been implemented in CoCoViLa. Some of the elements are shown in the scrollable toolbar in Figure 6. The developed package includes also visual classes for drawing charts and a simulation engine based

on iterative methods (e.g. Runge-Kutta) for solving ordinary differential equations. This package enables one to hierarchically construct mathematical models of large and complicated hydraulic systems that include thousands of variables, equations and functions. The tool checks the solvability of a task given by a goal, automatically generates problem solving algorithm and the corresponding Java program.

Steady state conditions and dynamic behavior of the hydraulic load-sensing system are simulated using the package described in this section. Figure 6 shows an example of simulation of steady-state conditions. The window in the background is the *Scheme Editor* with a scheme that represents the multi-pole model of a hydraulic-mechanical load-sensing system. The second window contains the corresponding textual specification of the model. The unfolded model of this problem includes 1 988 variables and 4 532 attribute dependencies. The generated simulation program is a Java class that has 4 958 lines of Java code. It is the attribute evaluation program for the particular problem. Attribute evaluation planning and code generation for this example takes about 2 seconds on a typical 2.0 GHz laptop. The window in the foreground in Figure 6 contains the result of this simulation – a 3D chart with calculated $1\,000 \times 1\,000$ points. It shows the efficiency coefficient of the load sensing system depending on the displacement of the directional valve and the load moment of the hydraulic motor.

11 RELATED WORK

The concept of a flat language has been used in several theoretical works [1, 6]. In principle, Albert et al. [1] use the flattening of a language for the same purpose as we do – for expressing operational semantics in a precise way. Their work is done on a rather abstract level in the context of semantics of functional programs, aiming to express aspects of modern multi-paradigm languages like laziness, sharing, non-determinism, equational constraints and external functions. In contrast, our work is oriented at the traditional representation of semantics in the context of practical application of automatic program synthesis.

Higher-order attribute grammars have been theoretically considered by Vogt et al. [20]. They have defined a class of ordered higher-order attribute grammars (HAGs) as an extension of classical attribute grammars in the sense that parts of the parse tree can be stored in an attribute, and a parse tree itself can be changed by attribute evaluation. The strict separation between attributes and parse trees is removed in HAGs. This adds considerable flexibility to the grammar. Vogt shows that pure HAGs have expressive power equivalent to Turing machines. The incremental attribute evaluation algorithm for HAGs is introduced that handles the higher-order case. We could not find any practical application of this approach. The reason is probably the unstructured character of HAGs and, as a consequence, the difficulty of implementation.

The NUT system [17] developed at Institute of Cybernetics in Tallinn and Royal Institute of Technology in Stockholm is a programming tool of a PRIZ family [9]

supporting declarative programming in a high-level language, automatic program synthesis and visual specification of problems by means of schemes. NUT restricts its attention to constructing programs from pre-programmed modules, rather than from primitive instructions of an imperative programming language. The specification language of NUT is an object-oriented language extended with features for program synthesis, the pre-programmed modules are methods of classes supplied with specifications. For automatic program construction, NUT uses rules of the *structural synthesis of programs* (SSP) [8], which is based on *intuitionistic propositional logic* [10]. For SSP, a specification of a problem is translated into the theory of intuitionistic propositional logic and the program is extracted from the constructive proof of that theory. CoCoViLa can be named a successor to NUT, being developed in the same institute, it includes most of the visual features of NUT and the attribute evaluation method is based on the ideas of SSP.

The relation between attribute evaluation and structural synthesis of programs has been also known earlier. In particular, Penjam [12] shows how attribute semantics of programming languages can be presented by means of computational models. Computational models are used for knowledge representation and problem solving using the method of structural synthesis of programs. He proves the semantic equivalence between attribute and computational models both being two approaches to program and compiler specification and implementation. We have continued the same line of research and introduced the higher-order attribute semantics of flat languages.

Attribute models with functional dependencies can be treated as functional constraint networks [18]. An attempt has been made in the NUT system to support constraint programming [13]. This has been achieved by using classes as a source of information for constraint satisfaction and by introducing *compute messages*, which are requests for automatic program construction and execution.

A good example of merging constraint programming with object-oriented programming is a multiparadigm programming language Oz and its primary open source implementation – Mozart [11]. However, a kind of the constraint programming incorporated in the NUT and CoCoViLa systems is different from the approach taken in Mozart/Oz. In the former systems constraint solving is carried out by the program synthesis. In other words, in CoCoViLa and NUT a static specification of a constraint satisfaction problem is given and then an executable code is generated if a problem is solved. CoCoViLa searches solutions of a problem on an attribute model using value propagation and higher-order attribute evaluation.

Mozart/Oz solve constraint-based problems dynamically, at runtime. User may interact and add new data to the specification that will be immediately propagated. Solutions of a problem are determined using the techniques of constraint propagation and constraint distribution.

There are several tools similar to CoCoViLa intended for visual specification of domain-specific problems. During the development of CoCoViLa, we have tried to take into account their features that make them user-friendly, first of all, for visual editing. One of the good representatives domain-specific modeling environments is

MetaEdit+ – a mature commercial CASE framework for rapid development and usage of domain-specific visual languages [16]. Both in CoCoViLa and in MetaEdit+ a domain-specific language first has to be designed and implemented by a domain expert. Then the language can be used by a user without a need to do any programming, problems are stated visually. The difference is in the semantics of visual specifications and the code generation. In MetaEdit+, code generators have to be defined on a metamodel using a built-in scripting language, which in a rather straightforward way transform models into some external language. CoCoViLa uses its precise semantics defined through higher-order attributes and automatically synthesizes a program from a declarative specification of a problem and a given goal. In other words, CoCoViLa is able to construct different programs on a single model depending on a particular goal. We have briefly discussed a semantics of visual languages already in a conference paper [4].

12 CONCLUSIONS

We have introduced a method of representing the semantics of a class of declarative languages that we call flat languages using higher-order attribute models. Methods of dynamic attribute evaluation on simple attribute models and on higher-order attribute models have been described. The methods are complete in the sense that they either produce an attribute evaluation algorithm for a given goal or show that the goal is unsatisfiable. It is important to understand how the complexity of attribute evaluation depends on the expressiveness of an attribute model. It varies from linear time complexity to the complexity of PSPACE-complete problems. Luckily enough, most of practical specifications require few higher-order dependencies, therefore the planning time remains in the limits of seconds.

The implementation of computations of higher-order attributes in Java, in particular the automatic generation of Java code is not a trivial task. Therefore we have included complete examples of some metaclasses and generated classes. These examples should also convince the reader that there are no restrictions on writing metaclasses in Java except the requirements of the correct usage of the interface `Subtask`.

The problem of consistency of higher-order attribute models has not been discussed here, but it is known that this problem is of high complexity in general, and can be solved only in some trivial cases. The experience of using the programming environment CoCoViLa during two years where the higher-order attribute semantics has been implemented has shown us that higher-order attribute semantics is a practically useful instrument for implementing domain specific languages.

Acknowledgments

This research was partially supported by the Estonian Science Foundation grant No. 6886 and the target-financed theme No. 0322709s06 of the Estonian Ministry of Education and Research.

We would like to express appreciation to the reviewers of this paper for giving substantial suggestions for making improvements to this work.

REFERENCES

[1] Albert, E.—Hanus, M.—Huch, F.—Oliver, J.—Vidal, G.: Operational Semantics for Declarative Multi-Paradigm Languages. In J. Symb. Comput., Vol. 40, 2005, No. 1, pp. 795–829.

[2] Grossschmidt, G.—Harf, M.: Multi-Pole Modelling and Simulation of a Hydraulic Mechanical Load-Sensing System Using the Cocovila Programming Environment. In: Proceedings of 6th International Fluid Power Conference "Fluid Power in Motio", Dresden: 6th International Fluid Power Conference (6. IFK), April 1–2, 2008 in Dresden, conference proceedings. Dresden: Dresdner Verein zur Frderung der Fluidtechnik e.V., 553–568, 2008.

[3] Grigorenko, P.—Saabas, A.—Tyugu, E.: Cocovila-Compiler–Compiler for Visual Languages. Electr. Notes Theor. Comput. Sci., Vol. 141, 2005, No. 4, pp. 137–142.

[4] Grigorenko, P.—Tyugu, E:. Deep Semantics of Visual Languages. In: E. Tyugu, T. Yamaguchi (Eds.): Knowledge-Based Software Engineering. Frontiers in Artificial Intelligence and Applications, Vol. 140, IOS Pres, 2006, pp. 83–95.

[5] Howard, W. A.: The Formulas-As-Types Notion of Construction. In J. P. Seldin and J. R. Hindley (Eds.): To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pp. 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

[6] Klunder, B.: Star-Connected Flat Languages and Automata. In: Fundam. Inf., Vol. 72, 2006, 1–3, pp. 235–243, Amsterdam, The Netherlands.

[7] Knuth, D.: Semantics of Context-Free Grammars. Mathematical Systems Theory, Vol. 2, 1968, pp. 127–145.

[8] Matskin, M.—Tyugu, E.: Strategies of Structural Synthesis of Programs and Its Extensions. Computing and Informatics, Vol. 20, 2001, pp. 1–25.

[9] Mints, G.—Tyugu, E.: The programming system PRIZ. J. Symb. Comput., Vol. 5, 1988, No. 3, pp. 359–375.

[10] Mints, G.—Tyugu, E.: Propositional Logic Programming and PRIZ System. J. Log. Program., Vol. 9, 1990, Nos. 2–3, pp. 179–193.

[11] The Mozart Programming System. `http://www.mozart-oz.org`.

[12] Penjam, J.: Computational and Attribute Models of Formal Languages. Theoretical Computer Science 1990.

[13] Penjam, J.—Tyugu, E.: Constraints in NUT. In: Mayoh, Brian; Tyugu, Enn; Penjam, Jaan (Eds.): Constraint programming: Berlin: Springer, (NATO ASI series. Series F, Computer and Systems Sciences), pp. 330–349, 1994.

[14] Pierce, B. C.: Types and Programming Languages. MIT Press 2002.

[15] Statman, R.: Intuitionistic Propositional Logic Is Polynomial-Space Complete. Theoretical Computer Science, Vol. 9, 1979, pp. 67–72.

[16] Tolvanen, J.-P.—Rossi, M.: 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, pp. 92–93, New York, NY, USA, 2003.

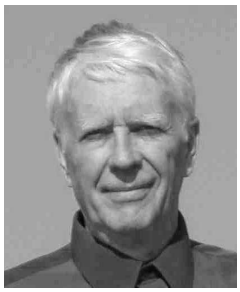
[17] Tyugu, E.—Matskin, M.—Penjam, J.—Eomois, P.: NUT – An Object-Oriented Language. Computers and Artificial Intelligence, Vol. 5, 1986, No. 6, pp. 521–542.

[18] Tyugu, E.—Uustalu, T.: Higher-Order Functional Constraint Networks. In: Mayoh, Brian; Tyugu, Enn; Penjam, Jaan (Eds.): Constraint programming: Springer 1994 (NATO ASI series. Series F, Computer and Systems Sciences), Berlin, pp. 116–139.

[19] Tyugu, E.: Algorithms and Architectures of Artificial Intelligence. Volume 159 of Frontiers in Artificial Intelligence and Applications, IOS Press 2007.

[20] Vogt, H.—Doaitse Swierstra, S.—Kuiper, M. F.: Higher Order Attribute Grammars. In PLDI '89: Proc. of the ACM SIGPLAN 1989 conference on programming language design and implementation, pp. 131–145, ACM Press, New York, NY, USA 1989.

Pavel Grigorenko received his Master of Science in Engineering degree from the Tallinn University of Technology in 2006. At present he is a Ph. D. student at the Tallinn University of Technology and a researcher at the Institute of Cybernetics. He has been research intern at Microsoft Research in Redmond in 2009. His research interests include artificial intelligence, declarative specification languages and synthesis of programs. He is also interested in automated theorem proving.



Enn Tyugu has Dr. Sci. degree in computer science from Leningrad Electrotechnical Institute, has served as a Professor of computer science and software engineering at the Tallinn University of Technology and at the Royal Institute of Technology (Sweden). He is a member of the Estonian Academy of Sciences, of the IEEE Computer Society, of the Estonian Information Technology Society. He has written computer science books in Estonian, Russian and English. His present position is leading research scientist at the Institute of Cybernetics of the Tallinn University of Technology. His research interests are in intelligent software, simulation and cyber-security.