Visual programming in NUT

Enn Tyugu, tyugu@it.kth.se[1]
Rando Valt, rando@it.kth.se[1]

Address for correspondence:
Enn Tyugu
KTH/IT Electrum 204
S-164 40 KISTA
Sweden

Abstract

The deep semantics of a scheme is defined as a set of programs derivable from the scheme. A uniform way of representing deep semantics of schemes is introduced based on the usage of a program synthesizer. An implementation of structural synthesis of programs in the NUT system and visual tools built on top of it are described. A visual compositional programming technique based on these tools is demonstrated on a number of examples.

1. Department of Teleinformatics, Royal Institute of Technology, Kista, Sweden

1. Introduction

A visual programming tool is described in this paper which is intended for compositional programming. It can be, first of all, used for developing problem-oriented visual languages in engineering domains. Visual compositional programming can be considered as an extension of visual object-oriented programming thoroughly discussed in [1]. The extension lies in different encapsulation of objects and different usage of classes which is inherent to the NUT system [2]. With this respect, our technique is similar to visual programming based on constraint nets [3] or data-flow [4].

The novelty of the present work is, first of all, in a uniform representation of semantics of visual images and schemes composed from the images. (We use images instead of icons for representing objects. An image includes, as a rule, a number of elements with their own meaning: buttons, ports, text fields etc. and represents an object in more details than just a clickable icon.) We define a concept of deep semantics of a scheme as a class of programs derivable from the scheme considered as a specification in section 3. Implementation of the deep semantics requires a program synthesizer, and the NUT system provides it for us. Our aim has been to make the usage of tools for common tasks (including defining new classes) as convenient as possible. In simple cases, the synthesizer and the run-time support of synthesized programs behave just as an equation solver. From the other side, the NUT system provides advanced features like object-orientedness, hierarchical composition, interoperability with other languages (C, Tcl) for programming in large, and these features have been made available for visual programming.

In order to demonstrate our approach to the compositional programming, we present here an example of a simple application: development of a visual language for calculating loads and kinematics of gearboxes. As soon as we have decided that the visual compositional approach is applicable, i.e. the specifications of problems in the application domain can be represented graphically, and the power of the synthesizer is practically sufficient, we develop an ontology of the problem domain. In the present case, the ontology consists of one concept: gear (toothed wheel) and its features: diameter *D*, rotations per second *n*, linear speed *v*, torque *T*, tangential force *F*, module of teeth *m* and number of teeth *z*. This concept is specified as a class, and its features, which are numeric parameters in this case, become instance variables of the class. Figure 1 presents a NUT class window with a specification of the gear. One can see the usage of equations in the class specification. There are two data structures *tangential* and *axial* defined in this class. These structures will be used for connecting gears tangentially and axially.

The next step will be development of an image for the class *gear*. This image, familiar to mechanical engineers, can be a simple rectangle with horizontal sides defined as ports for tangential connection of gears, and an axis through the rectangle with two more (axial) ports on its ends as shown in Figure 2a. Figures 2b and 2c show us configurations of axial and tangential connections of gears. Connecting two images via ports means equality of objects associated with ports. Looking at the class definition, we see that the tangential connection means equality of forces *F*, moduli *m* and velocities *v* of the connected gears, and the axial connection means equality of torques *T* and rotations per minute *n* of the gears. This gives us a visual language for specifying kinematics of gearboxes. Together with standard graphical features of NUT it is sufficient for solving computational problems about kinematics and loads in gearboxes.

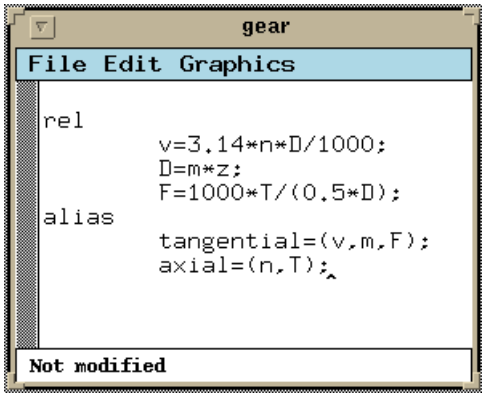


Figure 1. Class of gear

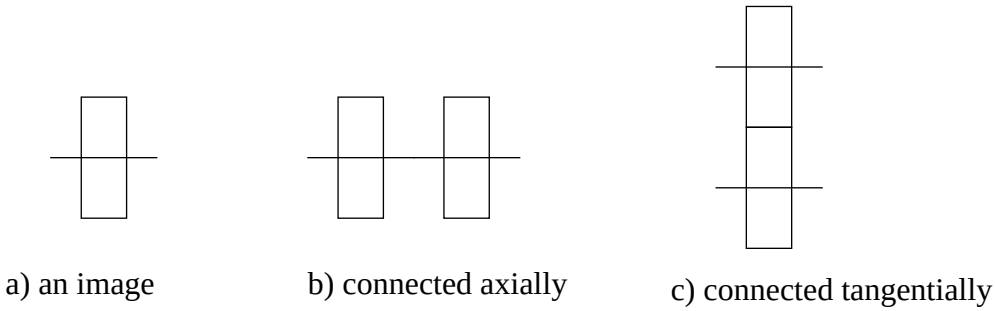


Figure 2. Visual ontology of gears

Figure 3 shows a specification of a gearbox in a scheme editor window and results of some computations in a NUT object window for the gearbox. As a standard feature of NUT, a window can be opened for any object, and it can be used as an input/output window. One such window for a selected gear is visible at the bottom of the figure.

This example demonstrates how simple can be the development and implementation of a visual language for some restricted engineering domain, if a tradition exists of graphical representation of objects (gearboxes in this case), and computations are in the scope of the NUT system capacities.

In the next section we consider the program synthesis and the specification language of NUT. In section 3 we define deep semantics of schemes. A discussion of visual tools of the NUT system is presented in section 4. The fifth section presents scheme editor as a computing environment. The last two sections of the paper are devoted to the visual compositional programming technique, illustrated by a number of examples.

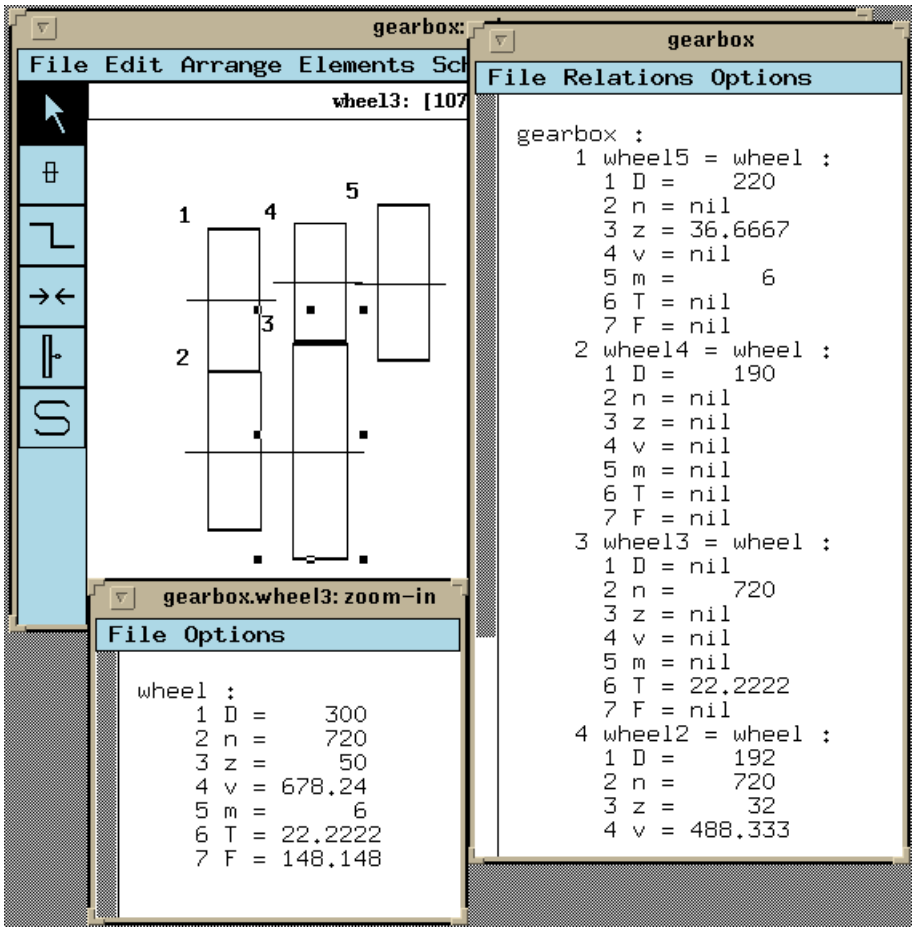


Figure 3. Specification of a gearbox and results of a computation

2. About the NUT system

The NUT programming environment used in this work has evolved from a prototype implementation developed in the framework of a new generation computer project [5] into a Unix-based tool for experimental programming and prototyping [2] used now in undergraduate education and applicable in simulation and engineering computations. The present version of NUT combines object-oriented programming, visual programming and automatic program construction paradigms. The object-oriented concept of class is extended in two ways: first, a class can be specified visually and, second, a class can serve as a specification for constructing programs automatically. The NUT lan-

guage includes two parts: a declarative specification language and a procedural language. The latter is needed for programming building blocks of software, i.e. programs in the form of methods of classes, and it is of no importance for understanding this paper. Here we describe briefly the specifications part and present the idea of program synthesis used in NUT.

The specification language is intended for describing classes in a rather conventional way. Its basic construction is

```
x : T;
```

which reads "*x* is a *T*", where *x* is a name of the component defined (of a new object), and *T* is its class. The language allows to bind objects by means of equalities, e.g. writing

```
x = y;  or  x.u = z.w;
```

where *x, y, z* are names of objects and *u, w* are components of *x* and *z*.

The built-in classes are: *num*, *bool*, *text*, *prog*, *array* and *any.* The latter is a universal class which has to be made precise before computing. We call it also undefined class. Explicit references to classes of objects can be omitted in many cases, if the actual class can be derived automatically.

Methods of a class are supplied with axioms which describe their applicability. Method specifications have the following general form:

<name>: <axiom> {<implementation>}

where the name (together with the colon) may be omitted, the axiom is a logical formula and implementation is a program written in the procedural language of NUT. The general form of the axiom is

$(x_{1,1},...x_{1,k1} \text{ -> } y_{1,1},...,y_{s,1}),...,(x_{m,1},...,x_{m,km} \text{ -> } y_{m,1},...,y_{m,sm})\ u_1,...,u_n \text{ -> } v_1,...,v_t$

where commas denote conjunctions and arrows denote implications. The precise meaning of the axioms will be defined below when the program synthesis will be explained (see section 2.3). In many simple cases, axioms have the following form:

$u_1,...,u_n \text{ -> } v_1,...,v_t$

and denote that $u_1,...,u_n$ are inputs and $v_1,...,v_t$ are outputs of the method, e.g.

r: x,y -> z {z := x+y};

Instead of writing methods explicitly, one can write equations which will be solved by the NUT system. The equation solver in itself is a complicated part of the system, but it is sufficient to know here that a powerful built-in equation solver is used for solving single equations. This allows us to use equations as sources of methods in classes. For instance, the equation

```
D=n*z
```

in the class of gear is semantically identical to the following three methods:

```
n,z -> D {D := n * z };
D,z -> n {n := D / z };
D,n -> z {z := D / n};
```

We can consider the equation above as an abbreviation for denoting these three methods.

Data structures are described by the construction

```
alias x = (y1,..., yn);
```

where *x* is a name of the new structure (which becomes a component of the class defined) and *y1*,..., *yn* are names of its components, which have to be names of already defined objects. Also this construction can be considered as an abbreviation for an extended specification. It is semantically identical to the following methods:

```
y1,...,yn -> x{ x := [y1,...,yn] };
x -> y1 { y1 := Select1(x) };
...
x -> yn{ yn:= Selectn(x) };
```

where $Select_i$ *i=1,...,n* are selector functions for selecting the *i*-th component of the argument.

Multiple inheritance is expressed by the construction

```
super t; ... super s;
```

where *t*, ... *s* are names of superclasses.

These constructions constitute the specification language into which visual specifications are translated. The gearbox scheme given in Figure 3 gives the following text:

```
var
    r1 : gear;
    r2 : gear;
    r3 : gear;
    r4 : gear;
    f5 : gear;
rel
    r1.tangential = r2.tangential;
    r2.axial = r3.axial;
    r3.tangential = r4.tangential;
```

```
r4.axial = r5.axial;
```

where *var* and *rel* are keywords for distinguishing specifications of components and methods (relations), and *axial* and *tangential* denote axial and tangential components of respective gears.

All information about the computability of components of a class is represented internally in NUT in the form of logical formulae - axioms of methods as described above. Each class has its own set of axioms. For a new class specified compositionally, e.g. by writing

```
x : C;...;y : D;
```

where *C,...,D* are known names of classes, a new set of axioms is built as a union of the axioms of these classes. Not going into the details of semantics, we can say that every class in NUT has a representation in a logical language in the form of a set of formulae which are the axioms of a theory about this class. It is important to notice that the axioms express computability, and every axiom has a realization - a method which can be performed for computing what the axiom describes. The inference rules for reasoning about a class (which are the same for all classes) are admissible rules of the intuitionistic propositional logic obtained from introduction and elimination rules for conjunction and implication [6]. The theories about classes are used by the NUT planner - a part of the NUT system which performs program synthesis in the following way.

NUT programs include statements which describe computational problems, i.e. goals for computations without explicitly given algorithms. A computational problem in an abstract form is represented as

c |- $x_1,...,x_m$ -> $y_1,...,y_n$

which means “compute $y_1,...,y_n$ from $x_1,...,x_m$ knowing c”. More precisely, c is a class name, x-s and y-s are components of this class and “knowing” means having axioms about the class. In order to solve the problem, first, the implication $x_1,...,x_m$ -> $y_1,...,y_n$ will be derived in the theory about c, thereafter a program is composed from the realizations of the axioms used - this is the synthesized program. Finally, this program is executed and the values of $y_1,...,y_n$ are computed from the given values of $x_1,...,x_m$. This is a special scheme of deductive program synthesis called structural synthesis of programs (SSP). It has been thoroughly described in literature [6]. The SSP has a pleasant property of decidability: one can decide for each computational problem whether the problem is solvable or not (i.e. whether the implication $x_1,...,x_m$ -> $y_1,...,y_n$ can be derived), and in the case of solvability, a program for solving it can be built from the solvability proof. Theoretically, building the solvability proof may require quite long search, but in practical cases the search time is quite small [2].

The logical language of SSP has only the following three kinds of formulae:

1. Propositional variables denoting computability of objects. We use the boldface names of objects as names of propositional variables so that $\boldsymbol{x}$ in the logical language means “the object x is computable”.

2. Simple implications (we denote conjunctions by commas here as in the NUT language):
x,...,y -> u,...v
which express the computability of *u,...,v* from given *x,...,y.* Each axiom must have a program associated with it for performing the computations.

3. Nested implications of the form
F,...,G x,...,y -> u,...,v
where ***F,...,G*** in their turn are implications as in case 2. Let us look at one of these implications, and let it be ***a,...,b -> c,...d***. This implication represents a subtask "compute *c,...,d* from *a,...,b*" in the whole formula. The meaning of the whole formula is "*u,...v* are computable from *x,...y* if the subtasks ***F,...,G*** are solvable. (Here is an analogy with the Prolog clauses - ***F,...,G*** constitute a body and the last implication is a head.)

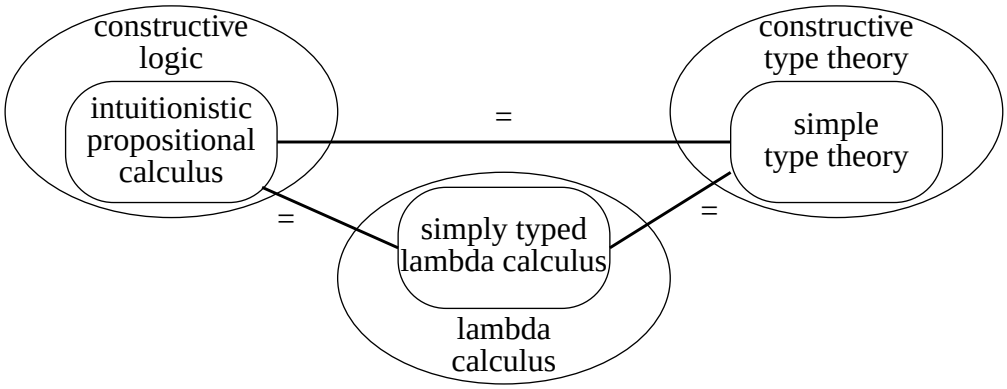


Figure 4. Three representations of program synthesis logic

Here we explained the semantics of specifications and the structural synthesis of programs in terms of intuitionistic propositional logic (IPL). It can be described in terms of a constructive type theory as well. Then it corresponds to a part which is called simple type theory. Finally, all the axioms derivable from specifications could be written in a lambda calculus. It is sufficient to use simply typed lambda calculus in this case. Figure 4 summarizes the possibilities of formal representation of semantics of NUT specifications for synthesis purposes. Choosing decidable fragments of powerful calculi, we obtain good performance characteristics for program synthesis, preserving sufficient expressive power under the assumption that one can accept large theories (with thousands of axioms). We are able to handle thousands of axioms which can represent computability of more than ten thousand objects. The time needed for synthesizing a program remains less than a second in all our practical applications. Roughly speaking, the time-complexity of the synthesis problem depends on the nestedness of loops in the synthesized program, which characterizes the number of nested subtasks used. In the case of one subtask in a specification, the time is linear with respect to the size of a specification in the logical language.

Now we give an example derivation of a program for computing the minimal value of the maximal values of elements of rows of a matrix. The building blocks (methods) needed for constructing the required program are as follows:

```
rownr,columnr,matrix -> element {f1}
(columnr -> element) -> max {f2}
(rownr -> max) -> minimax {f3}
```

where *f1* is a program for selecting the element of the matrix, *f2* is a program for computing the maximum and *f3* is a program for computing the minimum. The goal is represented by the following formula:

matrix -> minimax .

Using axioms of methods as logical formulae, we can build the following derivation tree where each derivation step (denoted by a horizontal line) is application of a derivation rule of SSP:

$$\dfrac{\dfrac{\textbf{\textit{(columnr -> element) -> max}} \qquad \textbf{\textit{rownr,columnr,matrix -> element}}}{\textbf{\textit{(rownr -> max) -> minimax}}} \qquad \textbf{\textit{rownr,matrix -> max}}}{\textbf{\textit{matrix -> minimax}}}$$

and the extracted program (in functional style) is
(λ matrix) f3((λ rownr) f2((λ columnr) f1(rownr,columnr,matrix))).
For more details about formal aspects of structural synthesis of programs see [6].

A computational problem may appear in several forms. First, it can be a subtask in an axiom as described in the beginning of this section. Second, it may appear as the keyword `spec` in the body of a method which has to be synthesized according to its external specification as in the following example:

```
r: x -> y {spec};
```

The keyword `spec` here shows that a program for computing `y` from `x` has to be synthesized using the class where `spec` is written as a specification. Third, it can be a message *compute* sent to any object *w*:

```
w.compute(a,b)
```
or `w.compute()`

The first message specifies the goal “compute *a, b* on the object *w*” and the second - “compute all what is possible on the object *w*”. In these cases the specification is not only the class of the object *w*, but also the values of components of *w* which are already computed.

The largest modular unit in NUT is package. It is a collection of programs, specifications and data, related to a computation. More precisely, it is a collection of texts, classes, global objects and graphic views (icons, images and schemes) intended for usage in the particular computation or in a whole problem domain.

3. Deep semantics of schemes

We use schemes as specifications of problems. Schemes are built from images of objects by connecting them in various ways. A meaning of a scheme can be always considered to be a data structure representing a graph marked with the types of its nodes and arcs. We call this **shallow semantics** of schemes. However, in application domains other than pure graph theory, a scheme must have another - a deeper meaning which is hidden in the types associated with its elements. This meaning can be explained verbally and, to a certain degree, represented in a computer by the programs which can manipulate schemes.

We define **deep semantics** of schemes in a uniform way as a set of computations (programs) automatically derivable from a scheme for solvable goals, using a semantic definition of the scheme language. For the NUT language the set of solvable problems is well-defined due to the decidability of theories about classes (see section 2). The idea of representing semantics of graphics computationally has been expressed already in [7] where attribute grammars were used for this purpose. Figure 5 shows the relation between a conventional operational semantics of a scheme and its deep semantics as implemented in NUT.

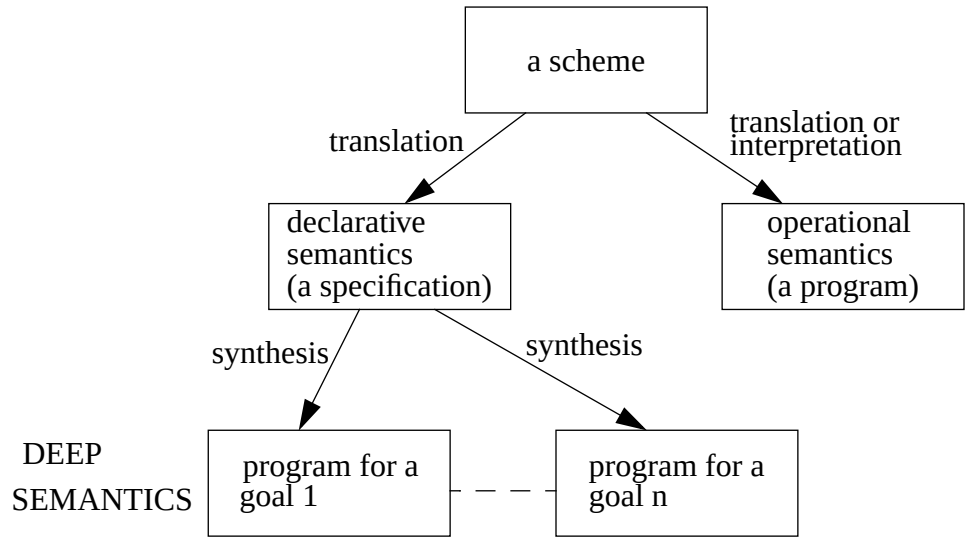


Figure 5. Deep semantics of a scheme

In order to give a deep semantics of a visual (scheme) language in this setting, one has 1) to define meanings of possible components of schemes and 2) to show how to get the meaning of a scheme

from the meanings of its components and their connections. The first part is defining an ontology of the problem domain, i.e. defining concepts and their possible relations. In the second part we use automatic program synthesis. This can be done in two stages: first, translating visual representation into a text in a specification language which is understood by a program synthesizer, and, second, synthesizing a requested program from the specification. Obviously, the program synthesis stage is crucial in this implementation of semantics.

4. Functions of scheme editor

Scheme editor of NUT is the main programming (specification) tool for users of problem-oriented visual languages developed in NUT. Its typical usage is drawing a scheme which represents a class used as a specification of a device, a system, a process etc. Figure 6 shows a scheme editor

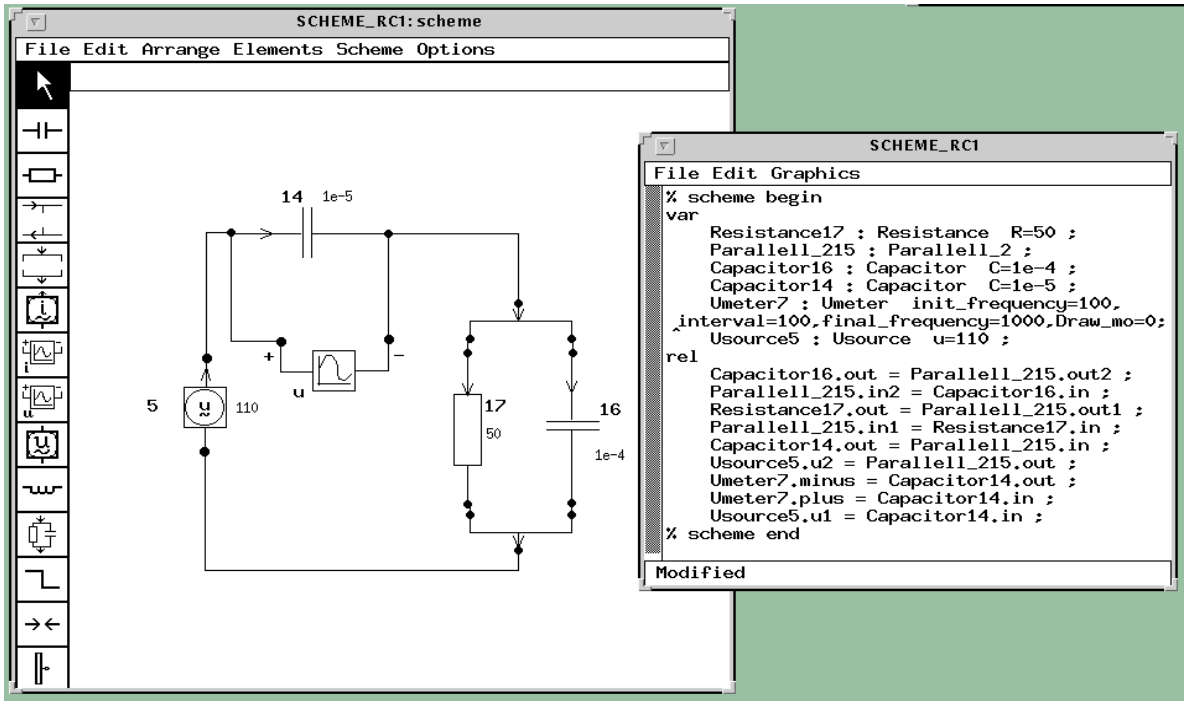


Figure 6. Scheme editor window

window of a package for analysis of alternating current circuits with a scheme of a circuit. This package has been developed by students as a course assignment. The palette of the window contains at the bottom of the palette three permanent icons for connecting images in a scheme, and one permanent icon on top of the palette for selecting a graphics mode. All the other icons are specific for the problem domain and represent classes needed for specifying the problems (e.g. capacitor, inductivity, voltage source, parallel and series connections etc.). Images of classes contain ports for connecting them. Each port represents a component of the object represented by the image (or the object itself, if the name of the port is ^*self*). A scheme like in Figure 6 is a

complete specification of a problem which can be solved by selecting an element of the scheme as a goal and giving a command for computing it from the *Elements* or *Scheme* menu. This command is performed, first, by translating the scheme into a text of a class (shown on the right side of the Figure 6), then by synthesizing a program from the text and, finally, by running the synthesized program. If the problem is solvable, the user does not need to follow these steps, and gets only the results of computation.

The textual specification can be used for debugging of specifications, and can be manually edited as well. Although, it is unusual to use the text window during problem solving.

There are several ways of connecting the images of a scheme. Let us demonstrate the possible connections on the following classes:

```
class C1:          class C2:          class T1:
var                var                var
    a: T1;             x: T1;             u: num;
    b: T2;             y: any;            v: text;
```

Let the classes *C1* and *C2* have ports for all their components, and besides that, let the class *C1* have a port called ^*self*. Figure 7 shows various connections of instances of these classes together with textual descriptions of connections printed later as comments in the window. We have the following cases:

a) Two ports having one and the same class are connected by equality. An equality between two ports (which are objects) means that all their respective components are bound by equalities, which in our example are *C11.a.u = C22.x.u; C11.a.v = C22.x.v;*

This defines a relation between the objects *C11* and *C22*. A type of the relation is determined by the ports, i.e. by the equalities established by the connection as well as by the relations of the class of the connected ports.

b) A port *C24.y* with the class *any* is connected with another port, the class of *C13.b* becomes also the class of *C24.y* after the connection is established.

c) A port called ^*self* of the object *C15* is connected with the port *C26.y* with the class *any*, the latter obtains the class of *C15* and becomes a proxy of *C15* in the object *C26* (one can say that a slot in *C26* is filled).

d) Two ports are connected, one of them has the class *any*, the port *y* gets the class of another port, which is now different from the class in case b. This is visual support of the polymorphism of the NUT language.

A connection between the ports *x* and *b* is not included into the examples. It can be legal only, if the classes *T1* and *T2* are compatible (have components of compatible types). In such a case, it had been reasonable to choose one and the same graphical notation for these ports.

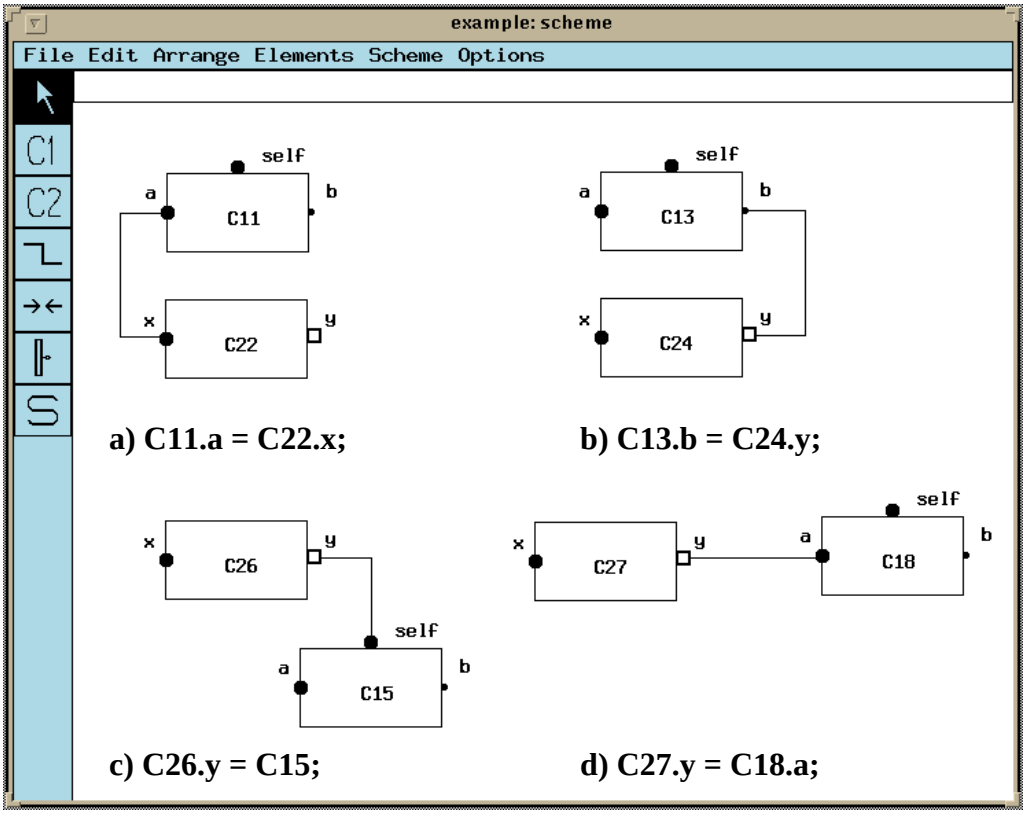


Figure 7. Connected objects

If we look at the circuit analysis package (Figure 6) deeper, we can see that all ports are of one and the same class. The class of ports includes complex variables *U*, *I* and *Z* together with formulas describing some transformations. Even in the very simple example of gears we had ports containing three components. This allows one to synthesize programs which start on one object, e.g. *C11* in Figure 7a, continue on another (*C22*) then return to the first object and so on. Each connection can be considered as a bundle of dataflow paths between the connected objects. The case shown in Figure 7c is even more interesting. The object *C15* fills in a slot in another object (*C26*) and can be used exactly as any other component of the latter.

A connection between ports specifies always a relation between the objects to whom the ports belong. The type of this relation depends on which ports are connected. However, one can represent a relation even more explicitly, as a separate object. Figure 8 shows two sets: *people* and *children* connected by a relation *subset*, and also the text of a class corresponding to the scheme. There are several packages developed in NUT for computing on sets which contain relations like subset, filter, projection etc. Roughly speaking, these packages implement relational data model together with pure relations on sets. These packages have been considered from the program synthesis point of view in earlier papers [8].

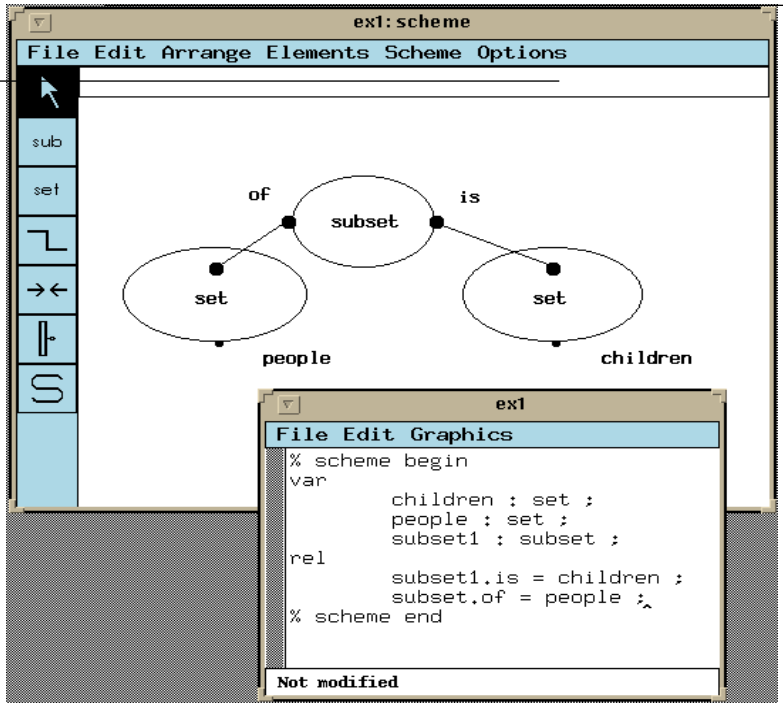


Figure 8. Explicitly represented relation.

There are a number of other useful features of the scheme editor applicable for compositional programming shown in Figure 9:

- Superclasses can be declared in a scheme by means of the superclass command which is the letter S in the palette. In the figure we have a superclass *gears*. This is visible from the text of the class created from the scheme and visible on the left side of the figure. Any user-defined class can be used in the superclass image.

- Images can be glued together as shown in the picture of a shaft. The gluing command is denoted by two arrows in the palette. Semantically it has the same meaning as an ordinary connection of ports. This can be seen from the class text. This text shows that the shaft is considered to be a scheme built of steps, an object representing length of the shaft, and a key.

- Images can be declared modifiable by size. All steps of the shaft are images of one and the same class *step*, but they have different size and shape given them when the shaft was drawn. Also the image of length of shaft denotes an object. Its class is *length*. The length of this image was adjusted manually to the length of the shaft when it was drawn.

- Structures can be created by means of the *alias* command which is the second from the bottom in the palette. The example contains a structure with the name *alias1* composed of three gears.

This name has been generated automatically, although any name can be given to a structure by means of the *Name* command from the *Elements* menu.

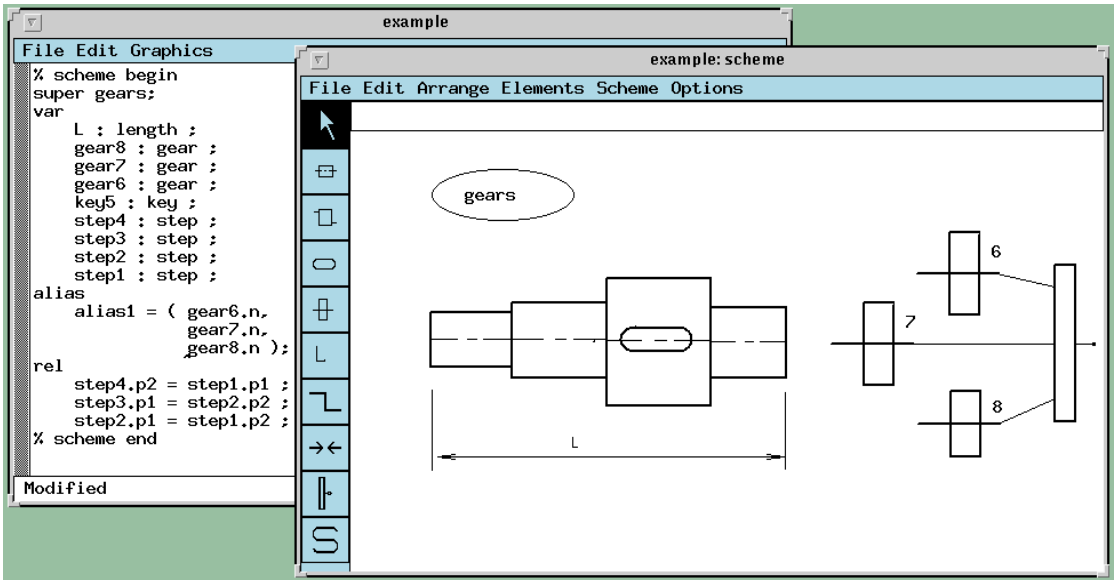


Figure 9. Standard features of scheme editor

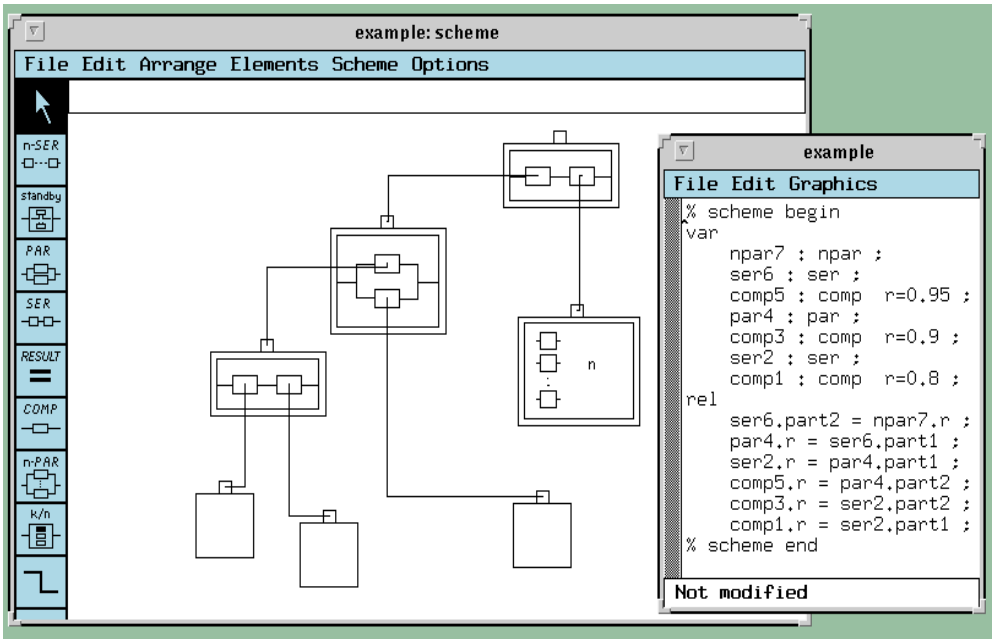


Figure 10. Reliability analysis example

There are numerous engineering application problem domains where problems can be mathematically described as sets of equations, and on a conceptual level by schemes. These problems can be easily specified visually. Here we have as an example the analysis of reliability of complex systems, Figure 10. Each such system is described by its structure in terms of various blocks and compositions of blocks. The problem is finding resulting statistical characteristics of reliability of a system from its structure and reliability characteristics of its blocks. Even an inverse problem can be solved - finding a required reliability of some block so that the requirements on the reliability of the whole system will be satisfied. Palette in Figure 10 shows the visual ontology of the reliability analysis language, and a separate textual window shows a textual specification derived from the scheme. Also this language was implemented as a student course project in the course Knowledge-Based Software Tools given at the Royal Institute of Technology.

5. Scheme editor as a computing environment

Until this point, we have considered a scheme purely as a representation of a specification which takes the form of a class on the textual level. Visual programming assumes also interaction with a scheme during the computations. This includes browsing among objects, sending messages, and changing values of some variables. In the present version of the scheme editor, we use one and the same scheme for all these purposes, and assume single assignment property on the topmost level of the specification. Besides the scheme editor window, an interactive graphics window is available during computations.

We describe interactions during computations on an example shown in Figure 11. This figure represents computations in a package for analysis of the accessibility of a GSM network for mobile clients. The main classes are *station* and *client.* The respective objects are represented in the scheme editor window as antennas and cars. The picture in the scheme editor window has been created, as described earlier, by clicking on icons in the palette and placing images into the window. In the present package, images of clients and cars have also a point whose coordinates in the window are passed to the object and can be used further in computations. For performing interactive computations, one has to create an object of the class *AREA* by command *Open Object* from *Scheme* menu. Stations and clients have methods for visualizing their actual position during computations in the interactive graphics window used as an output window in the present package (in the upper right window). A method can be activated by clicking on an element of an image which has been defined as active element and bound with the method by name. This has been used for drawing clients and stations together with access areas in their actual positions in the output window. Clients can move along the roads, their positions change, and this is reflected in the output window by activating their drawing method from a simulation program. The scheme editor window keeps initial positions of the objects.

In the upper left corner we can see zoom windows for *station1* and *client22*. They are opened by *Zoom in* command from the *Elements* menu of the scheme editor. The values in these windows can be edited, and this is a way to give input for computations. One can create images of objects with input/output fields of textual components. This is another way to give input for computations. We cannot change a shape of an image during the computations, but it is possible to visualize textual components of objects in output fields defined in images.

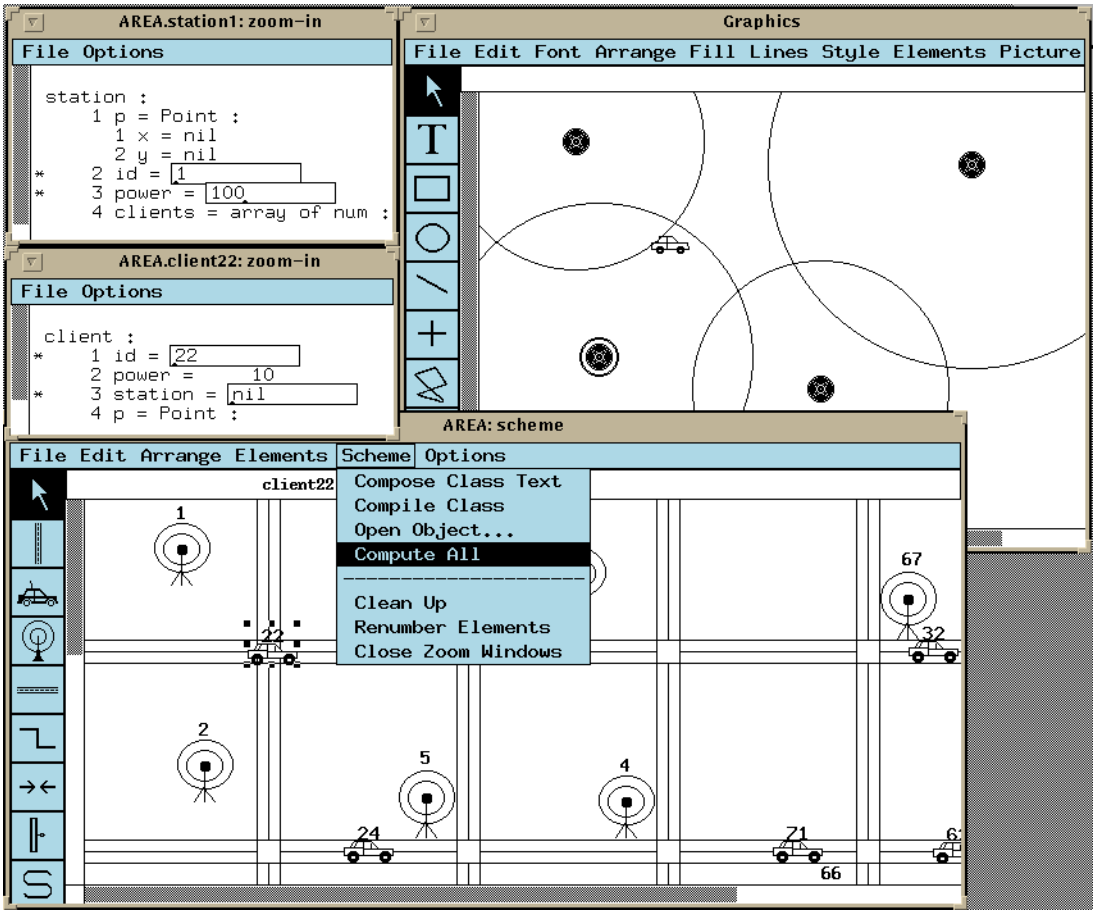


Figure 11. Interactions with scheme editor

An interesting command of the *Elements* menu is *Compute*. By giving this command, a request is sent to the program synthesizer for creating a program for computing all components of a selected object. (In the figure it is *client22*.) If this task is solvable, the program is run and the results are computed and visualized, otherwise, a message about the unsolvability is printed.

Let us look now at the *Scheme* menu of the scheme editor. It contains the commands for composing a class text from a scheme and for compiling this text. These commands are useful for debugging purposes. In a well-developed package one uses only the *Open Object* command (see above) and the *Compute All* command which is a request for computing as much as possible from the given specification. This is often used, when computations are not too long and one can afford to compute everything which can be computed from a given specification. Meaning of other commands of the *Scheme* menu is quite obvious.

6. Developing a visual language

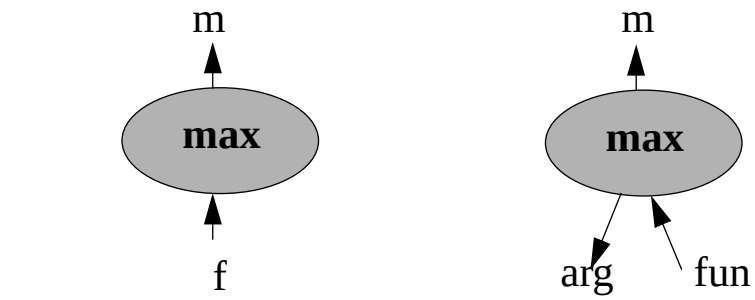
Here we describe the process of developing and implementing a visual language in the framework of the visual editor and the program synthesizer. In order to develop a visual specification language for a particular domain, we have to fill in the framework with concepts and relations, i.e.

to add a visual ontology. We assume that the language has been conceptually designed already, and we deal with its visual representation and implementation here. The following steps have to be taken:

- define the concepts as classes in the NUT specification language
- develop visual images of the classes
- define relations for binding the classes.

The first two steps are straightforward. Each concept is defined as a NUT class. Each class is supplied with an image which represents objects of the class in a scheme. The NUT system contains a graphic editor for developing images of classes, and for defining properties of elements of these images, in particular, ports bound to components of a class. Besides the images of concepts, small icons have to be developed which represent the concepts in the palette of a scheme window. This is done by means of a bitmap editor called directly from a class editing window.

The third step requires some explanations. In order to bind a concept by means of relations, some components of the concept have to be defined as ports. Connecting ports of two objects graphically establishes a relation whose type is determined by the ports. The internal logic of concepts has to be defined in such a way that it will support the program synthesis on a schemes built from images of classes bound via ports.




```
var
    f : any;
    m : num;
rel
    f -> m {<prog.of max.>}
```

a)

```
var
    x,y : num;
    m : num;
rel
(arg -> fun) -> m {<prog.of max.>}
```

b)

Figure 12. Two classes of maximum

Let us develop, as an example, a visual representation of a concept of maximum of a function defined on an initial fragment of natural numbers. This concept can be useful also for finding maximal element in a finite set, maximal element of an array, maximal value of a function defined

on the elements of an array etc. The class representing this concept has to contain a program which takes a function *f* as an input and produces the value *m* of maximum. This small program is a building block which can be easily programmed in NUT once and forever. The function *f* may be given in different ways, therefore we define its class as *any*. The function can be given explicitly as a text of program, because NUT supports run-time compilation. In this case, the axiom representing the semantics of maximum will be

f -> m

and the image of maximum will have two ports *f* and *m* as shown in Figure 12a.

The class of maximum may contain a slot for the unknown function which can be filled in by a specification of the function. In this case the specification has to contain just right names for the argument and value of the function, in order to be used properly. The image can be in this case again as shown in Figure 12a, but the value of *f* is now an object representing the function implicitly.

In order to show data dependencies more explicitly for the synthesis purposes we can define a port of argument *arg* of the function and a port of value *fun* of the function, as shown in Figure 12b. In this case, the axiom will contain a subtask *arg -> fun* for finding a program of function:

(arg -> fun) -> m

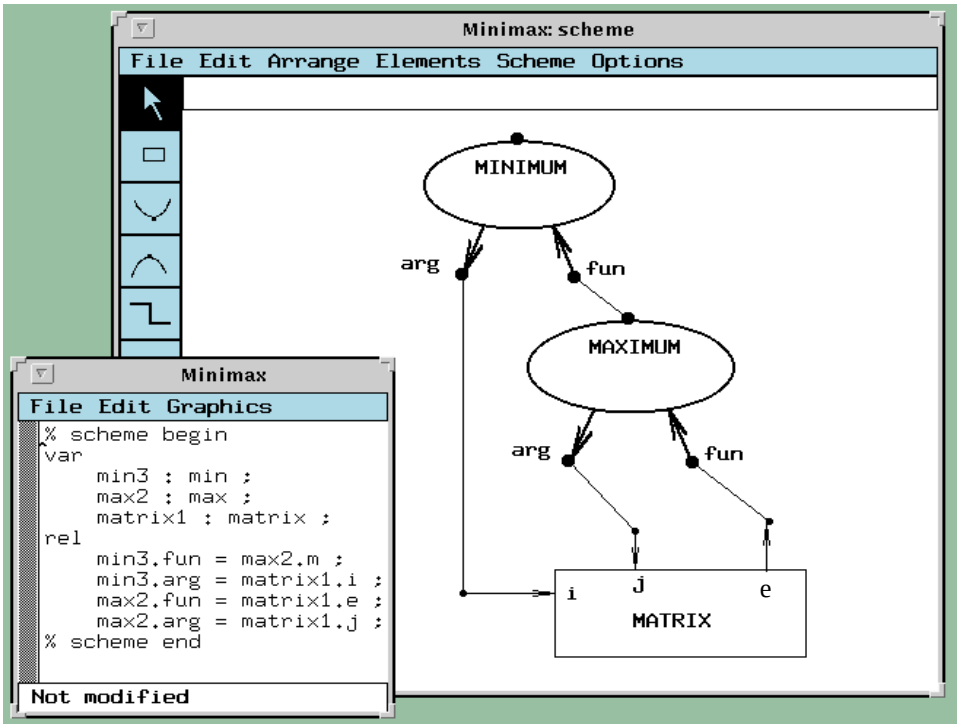


Figure 13. Visual specification of minimax problem

After developing a visual ontology, we get a palette with icons of concepts (i.e. classes) in the scheme editor window, and can start using the language. Figure 13 shows the window for a language containing three classes: *matrix, minimum and maximum.* Three instances of classes bound by relations are visible in the window. They represent a complete visual specification of the minimax problem: “find minimal value of maximal values of elements of rows of a matrix”. This specification should be understandable as soon as the classes of minimum, maximum and matrix are known. The figure shows also a window with the class text generated from the visual specification.

The class of matrix contains an axiom

i, j -> e

stating that for given number of row *i* and number of column *j* the corresponding value of element *e* can be computed. This axiom and axioms

(arg -> fun) -> m

of maximum and minimum are sufficient for synthesizing the required program. We described the program synthesis for this problem in section 2. This is a simple example of visual compositional programming. In order to perform a computation, one has to click on the desired object (which is obviously the image of minimum for our problem) and to select the compute command from the Scheme menu.

Besides a text obtained from a visual representation of a problem, the complete specification of the problem can contain text which describes persistent part of problem specifications. In particular, a number of simulation problems can be solved as follows. A control class is developed which implements a simulation method. This control class is used as a superclass in each particular problem specification. The text generated from a visual specification is always generated between the comments *% scheme begin* and *% scheme end*. If there is a text outside these comments, it is left unchanged by the scheme editor, and can be kept permanently for all problems solvable by a particular method.

Figure 14 shows a scheme of a logical device (an adder) represented in a visual language implemented by students as a course project. The same figure shows also a part of the textual specification of the problem of simulating this device. It begins with a superclass declaration

super engine;

The *engine* class contains a simulation method which requires computing the next state of the device from its given state using the scheme as a specification for the computation. Both states are composed of the respective components *next* and *current* of elements of the scheme which is specified by the following lines in the engine class (the keyword *all* is a wildcard symbol used in NUT):

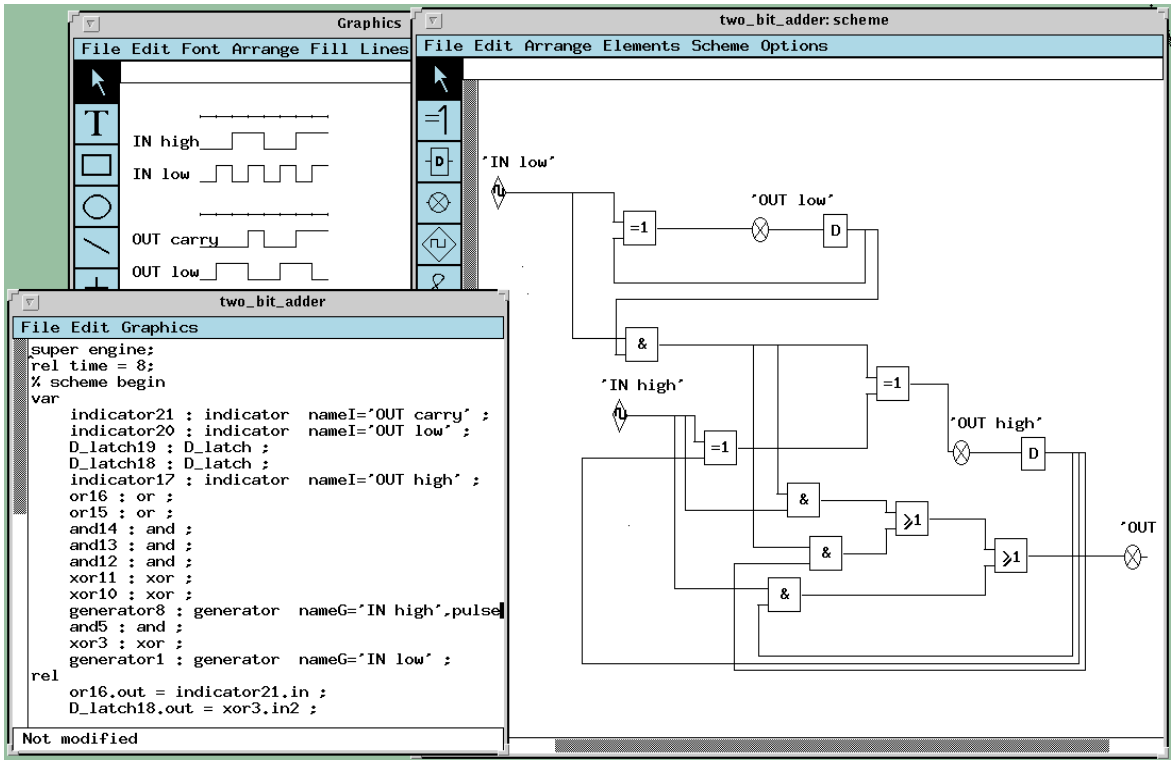


Figure 14. Specification with a control class

alias

state = (all.state);

nextstate = (all. nextstate);

The simulation method is a simple loop where computation of the next state, specified as a subtask of the method, is repeated for a given number of steps. Also the simulation time is given in the textual specification. This is a simple way of entering values of parameters of problems, although it requires opening of the class text window. The results of simulation of the device are visible in the top left corner of the figure.

7. Visual specification of large programs

We have implemented means for an easy navigation between visual and textual specifications and between scheme and its components. This enables us to present schemes of unlimited size by decomposing them and, if needed, sending them for processing to different workstations in a local network. The latter is done on the basis of the distributed computing toolkit rNUT [9] which is built on top of the PVM software.

Schemes of real devices may be large. Although a scheme in the scheme window is scrollable and not restricted by size, it is inconvenient to work with a single large scheme. The NUT system allows to encapsulate any part of a scheme into a separate class, develop an image for the class and use it as a component in schemes. A large system will be presented by a hierarchy of classes bound into a single scheme on top level. Practically, a system of any size can be represented as a hierarchy of schemes. Let us note that this is not an inheritance hierarchy usual in object-oriented software development. (The inheritance hierarchy is supported by NUT as well and can be used for representing concept-subconcept hierarchies.)

In Figure 15 we show the windows opened from a hierarchical scheme of the class *SCHEME_RC* representing an electrical device. These windows are the following: window of a class describing a RC-fragment (*Block_RC*), scheme of this class (*Block_RC:scheme*) and a zoom-in window of an object *Capacitor14* of the device. The following tasks of interactive computing, besides drawing the schemes of specifications, are denoted by respective numbers in Figure 15:

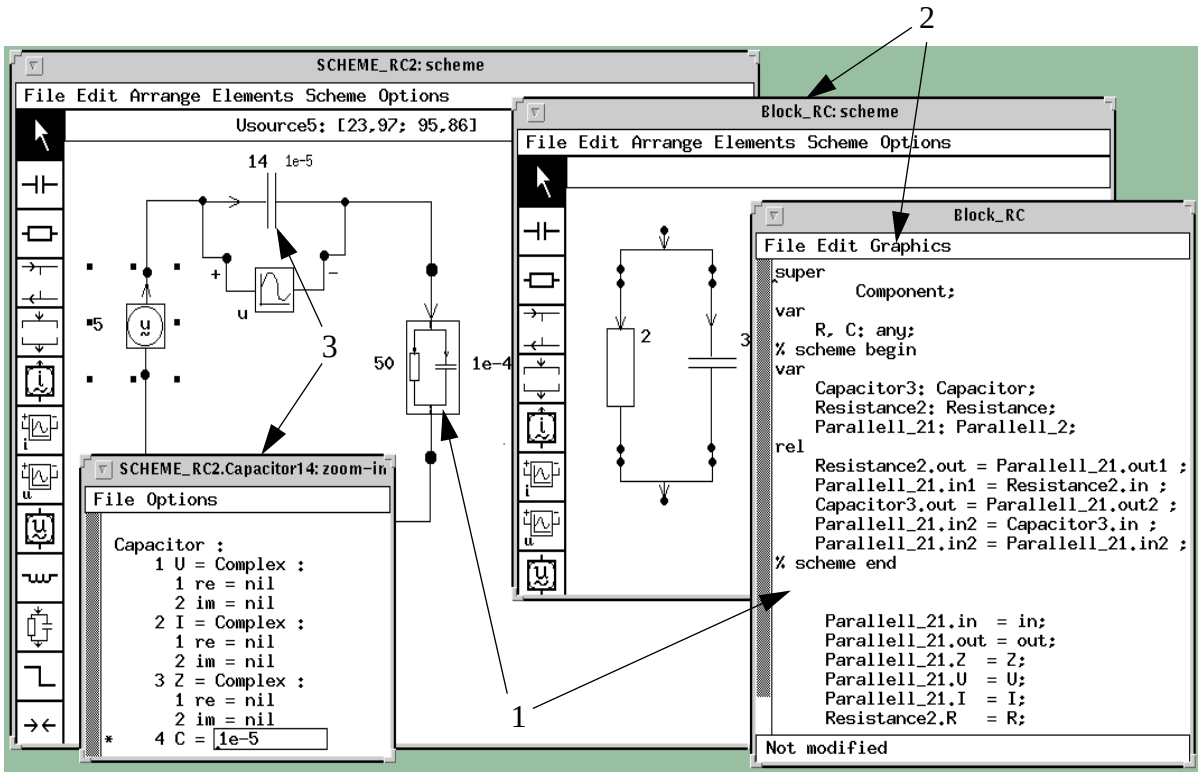


Figure 15. Hierarchical scheme

(1) Class window of an object in a scheme is opened by double-clicking on the image of the object.

(2) Scheme of a class is accessible from a menu of the class window. (This can be repeated recursively, so a hierarchy of schemes can be handled.)

(3) Objects represented by images in a scheme can be zoomed and values of their components can be observed and changed.

Modularisation of specifications can be combined with distributed computations as well. As soon as the whole specification becomes too large for an ordinary NUT program, the distributed version rNUT [9] can be used. In this case, however, breaking down a scheme into parts has to be done carefully, considering the communication between the NUT processes running on different computers.

8. Concluding remarks

We believe in visual programming languages, but not as substitutes for FORTRAN or C++. The procedural specifications, doubtlessly, can be best written in a textual form, because they have to be linear. This is why we didn't present here visual images of loops, conditionals or any other conventional control structures. Although this is possible, and we have developed them as examples for students, and even used some of them earlier in a small programming system for PCs.

We have discussed mostly development of problem-oriented visual languages. However, compositional programming can be applied directly to the development of large programs, in the same way as object-oriented programming. An ontology of the particular program (system) is developed during the design phase, then implemented, and applied to the implementation of the whole project. Advantages of this approach are clarity of design, modularity and, as a consequence, reliability and good maintainability of the developed system.

For those who are interested in testing the visual programming tools described here, we recommend to fetch them by means of anonymous ftp from *ftp://ftp.it.kth.se/labs/se/Software/NUT/* , archive files *nut3.0_sun4.1.tar.Z* or *nut3.0_sol2.4.tar.Z* and try them respectively in SunOS 4 or Solaris 2 environments on Sun computers.

Acknowledgements

We are much indebted to Benjamin Volozh who developed the first version of the scheme editor. We express our thanks to people who have been developing and supporting the NUT system during the years, first of all, to Tarmo Uustalu, Mait Harf, Mari Kõpp and Mihail Matskin. We are thankful to the anonymous reviewers whose suggestions have helped us to clarify our ideas in a considerably better way. This research has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK) under the grant number 9303405-2.

References

1. M. Burnett, A. Goldberg & T. Lewis (1995) *Visual Object-Oriented Programming* Manning Publications Co., Greenwich.
2. E. Tyugu (1994) Classes as program specifications in NUT. *Journal of Automated Software Engineering* **Vol. 1**, pp. 315 - 334.
3. P. Szekely & B. Myers (1988) A user-interface toolkit based on graphical objects and constraints. In: *Proceedings of the 1988 ACM Conference on Object-Oriented Systems, Languages and Applications*, San Diego, pp. 36-45.
4. M. Najork (1996) Programming in Three Dimensions. *Journal of Visual Languages and Computing* **7(2)**, pp. 219-242.
5. E. Tyugu (1991) Three new generation software environments. *Communications of the ACM* **34(6)**, pp. 46 - 59
6. G. Mints & E. Tyugu (1982) Justification of the structural synthesis of programs. *Science of Computer Programming* **2(3)**, pp. 215-240.
7. H. Gottler (1982) Attributed graph grammars for graphics. *LNCS*, **Vol. 153**, pp. 130 - 142.
8. E. Tyugu (1991) Declarative Programming in a Type Theory. In: *Constructing Programs from Specifications* (B. Möller, ed.), North-Holland, pp. 451-472.
9. V. Vlassov et al. (1994) Distributed programming toolkit for NUT. Technical Report TRITA-IT R 94:34, Dept. of Teleinformatics, Royal Institute of Technology, Stockholm.