Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ

Tohru Ogawa
Doctoral Program in Science and Engineering
University of Tsukuba
1-1-1, Tennodai, Tsukuba
Ibaraki, 305-8573, JAPAN
+81 298 53 5165
tohru@softlab.is.tsukuba.ac.jp

Jiro Tanaka
Institute of Information Sciences and Electronics
University of Tsukuba
1-1-1, Tennodai, Tsukuba
Ibaraki, 305-8573, JAPAN
+81 298 53 5343
jiro@is.tsukuba.ac.jp

Abstract : This paper describes a visual programming environment for an algebraic specification language where program editing and execution are shown visually. By expressing a program visually, programming becomes more intuitive and easier. We have developed the visual programming system CafePie for the algebraic specification language CafeOBJ. In CafePie, program editing and execution are all performed in one window. All operations of the program editing are handled in a uniform manner. Programming examples of CafeOBJ on our system CafePie are also given in the paper.

Keywords: Visual Programming System, Specification Languages, Human Interface

Introduction

Algebraic Specification Languages (ASL) are specification languages which express the models of the real world using elements such as sorts, operators and equations. We assume specifications written in ASL as programs. They are executable. The execution of the languages are performed by term rewriting.

There are many researches which carried out on visual programming[1]. Programs can be visualized by the use of graphics. It is important to reflect the user's view to the visual programming system. Most of the users try repeatedly to edit and to execute the programs. The visual programming system which can edit program visually and also can execute it in the same way is desired.

We have developed the visual programming system CafePie (Pictorial Interactive Editor for CafeOBJ)[2] for the algebraic specification language CafeOBJ[3]. CafeOBJ specification consists of module structures. Our system visualizes each module. We use the direct-manipulation techniques for program editing. Most of editing is performed only using mouse. The same visualization schema is used for both program editing and execution. Since the program editing and execution are performed in one window, program modifications are reflected directly to the program execution.

Program Visualization

The “visualization of program structure” means to express the structure of an original program using some pictorial or graphical objects. Several program visualization systems, such as PP[4], have already been proposed.

We visualize the program structures of CafeOBJ by expressing the program elements with pictorial objects. We call each pictorial object “Icon.” We have chosen the following primitive elements for CafeOBJ: Sort, Operator, Variable and Equation. These elements are expressed as specific icons in the screen.

We assume term as the basic data structures of ASL. By definition, a term is a variable or an operator which has terms in its arguments. On the other hand, the relation between sort and term is regarded as that between a container and its contents. The representation of the sort should be different from that of terms. We visualize a sort by a rectangle and an element of a term by an oval. Further, to distinguish operator and variable, we visualize operator by an oval with solid line and variable by an oval with no line. It is important that these icons have colors. User can distinguish colored icons easily. We express sort and operator, which is the main elements of the module, by bright colors, and the other elements by dark colors. The visualization rules for these elements are shown below.

Sort Icon: We use a directed graph to describe the implication relations of sorts in our CafePie system. Sorts are represented by nodes and implication relations by arcs. Many ordered sorts are used in CafeOBJ. Their relations mean implication. The implication relations are often described with Ben's diagram. This diagram is easy to understand as far as it is used to describe simple relations. But it becomes difficult when describing complicated relations.

A sort is represented by the green rectangle which has a label.

Relations between the sorts are represented with an arrow which goes from the subsort to the supersort. These relations shown in Fig. 1.

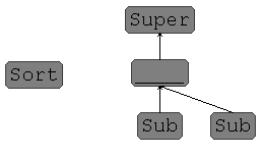

Figure 1: Sort Icon and Sort Relation

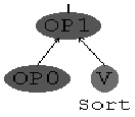

Figure 2: Term Icon

Term Icon: Before considering operator or variable representation, we consider the visualization of terms. We express term with tree structures. The component of the term, i.e. an operator or a variable, is represented by a node and super-sub relation between the component is represented by an edge (Fig. 2).

Operator and Variable Icon: An operator is represented with the oval of pale blue witch has a label for the operator name. The label for arities are arranged at the bottom part of the operator. The label of coarity is arranged at the top part of the operator. Arrows are drawn from arities to operator and from operator to coarity (Fig. 3).

A variable is represented with the orange oval, and the sort of the variable is represented at its lower part (Fig. 4).

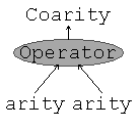

Figure 3: Operator Icon

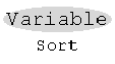

Figure 4: Variable Icon

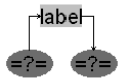

Figure 5: Equation Icon

Equation Icon: An equation represents a term rewriting rule. A label is arranged in the center of the equation. The initial term is arranged on the bottom-left side of the label, and the rewritten term is on the bottom-right side. To represent a term rewriting rule, arrows from the left term to the right term are drawn pass through the label (Fig. 5).

Module Field Icon: The program in CafeOBJ consists of modules. A Module contains other primitive elements: Sort, Operator, Variable, Equation. A Module is represented as a gray rectangle which called “Field Icon.” User edits the module definition on this special Icon.

Operations for Program Editing

We pay attention to the following points:

1. Use the direct-manipulation with mouse for program editing.
2. Use simple operations and make the working content clear.
3. Make operations being reversible.

Direct-manipulation is easy to learn and the user can recognize one's mistakes immediately. Complex and obscure operations can cause unexpected side effects. Using reversible operations, the user has a feeling to control the editing system and can edit programs more smoothly.

If programs are expressed as a combination of icons, icon-operations can be separated into two parts. One part is to generate a new icon, and the other is to edit the icons. To recognize clearly what user is doing, different techniques are used for these two operations. The double-click technique is used for generating an icon. The drag-and-drop technique is used for editing the icon.

Double-Click Operation

There are various mouse operations such as click, double-click, drag and so on. To select an icon, user ordinary moves the mouse cursor to the icon and clicks it with mouse. If this technique is used to generate a new icon also, unexpected error may happen. User wants only to select the icon, but unnecessarily a new icon is generated due to this method. We use double-click technique to generate a new icon. This technique it possible to distinguish icon manipulation from icon creation.

The primitive icons are sort, operator, variable and equation. We consider the symbols of icons' origins as follows (Fig. 6). A sort and an operator are newly generated by a user. A variable, which is a term, constructed from a sort. As we described earlier, variable are instantiated from the sort. An equation are generated from a term because it means a term rewriting rule and always has two terms: the initial term and the rewritten term.

For example, to make a new equation, first user generates a new term which has a variable, and edits the term, and next performs a double-click action at the root of the term. As the result of this action, a new equation which has two terms at its left-hand and right-hand is generated. The left-hand term is the term which should be rewritten. The right-hand term should be modified to show the term rewriting result.

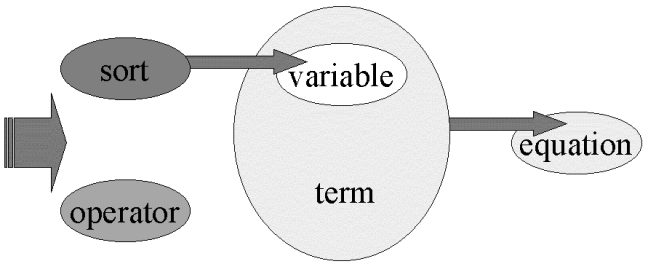

Figure 6: The Symbols of Icon's Origins

Drag-and-Drop Operation

In the case of the icon-movement, user move the icon by the drag-and-drop technique. If there is no icon at the place

Event Name	Source	Destination	Action
Make Sort-Relation	Sort	Sort	Relate one sort to an other(as Supersort)
Delete Sort-Relation	Sort	Sort	Delete the relation between two sorts
Add Arity	Sort	Operator	Add an Arity to an operator
Change Arity	Sort	Arity	Change the arity to one that has the sort name
Change Coarity	Operator	Sort	Change the coarity to one that has the sort name
Exchange Arities	Arity	Arity	Exchange one arity for the other
Create Subterm	Operator	Variable	The variable replaced the new term which made from the operator
Add Subterm	Term	Variable	The variable replaced the operator

Table 1: Operations in CafeOBJ language

where the icon is dropped, the icon is moved to the place. If there is an icon, these two icon are overlapped. Overlapping two icons with mouse plays an important role in editing.

"Cut-and-Paste" method consists of the following operations:
1. Select the icon to be removed from current field (Cut),
2. Move the mouse cursor over the destination icon and
3. Put the removed icon on the destination icon (Paste).
Cut-and-Paste method is easy to move an icon to a distant place. However, this method is not suitable for the direct-manipulation since user cannot see the manipulated icon between Cut action and Paste action. "Drag-and-Drop" method seems to more intuitive. The process of Drag-and-Drop method consists of:
1. Select an icon,
2. Move the selected icon to another icon (Drag) and
3. Overlap the selected icon with another icon (Drop).
The target icon moves with mouse cursor and it remains visible during the movement. User can move the icon by dragging, and can recognize what he is doing. Drag-and-Drop technique is well known for its simplicity. We reexamined this technique to realize the program editing. All operations of the icons-editing are handled in a uniform manner.

When an icon (A) is put on another icon (B), we call icon-A as "Source" and icon-B as "Destination." Editing process is the repetition of the elementary editing event. For example, there is a case that arity is added to the operator. This is called "Add Arity" event. This editing can be expressed by the pair of the sort and the operator. We consider the sort as "Source" and the operator as "Destination." This editing can be realized by Drag-and-Drop operation. Another example is called "Create Subterm." The user moves an operator onto a variable part of the term. After this operation, new term icon which has the label of the operator is created and the variable is rearranged by the icon. If the operator has arities, new variables' parts which belong the sorts of the arities is added to the term icon. In this way, the Drag-and-Drop technique can be adapted. Operations in CafeOBJ language are shown in Table 1.

Reversible Operations

The reversibility of operations means that the reverse operation should always be realized by ordinary operations. For example, user makes a new relation between sorts using drag-and-drop operation as shown in Fig. 1. Even if the relation is created by mistakes, the relation can be deleted by using "Delete Sort Relation" operation (Fig. 1).

We realize to make the icon-editing, which is implemented by the drag-and-drop technic (Table 1), the reversible operation.

The operation of "making sort relation" is reversible by the operation of "deleting sort relation." "Add arity" operation is also reversible. If an operation is added a new arity, the arity can be deleted from the operator using drag-and-drop operation. The operation of "changing or exchanging something" is reversible.

The operation of "adding subterm" is also reversible. If the operator part of a term can be overlapped by new term and the subterm has changed the new term, the term can not be reversed to the original one without the information of the original term. The operation is called destructive operation. The operation of "adding subterm" can be reversible without this destructive operation. The part of the term which is added to a term is only the variable part. If the variable is changed by the term, the variable can be returned to the original. Because the information is only the sort of the variable and is contained in the operation part which has the variable.

The System

Program Editor

Programming in the algebraic specification languages is expressed as a combination of icons. Therefore, the program is edited by the icon operations. Programming in CafeOBJ consists of the following process:
1. Make a module,
2. Declare sorts and relation between sorts on the module,
3. Declare operators and variables on the module and
4. Declare equations on the module.
All operations which appear in the following examples are defined in the former sections.

A snapshot of CafePie is shown in Fig.7. Assistant Operation Part consists of buttons, and is used for loading/saving file, watching the help and so on. Text Input Part is used to input the names of icons. Working Part consists of New-Field,

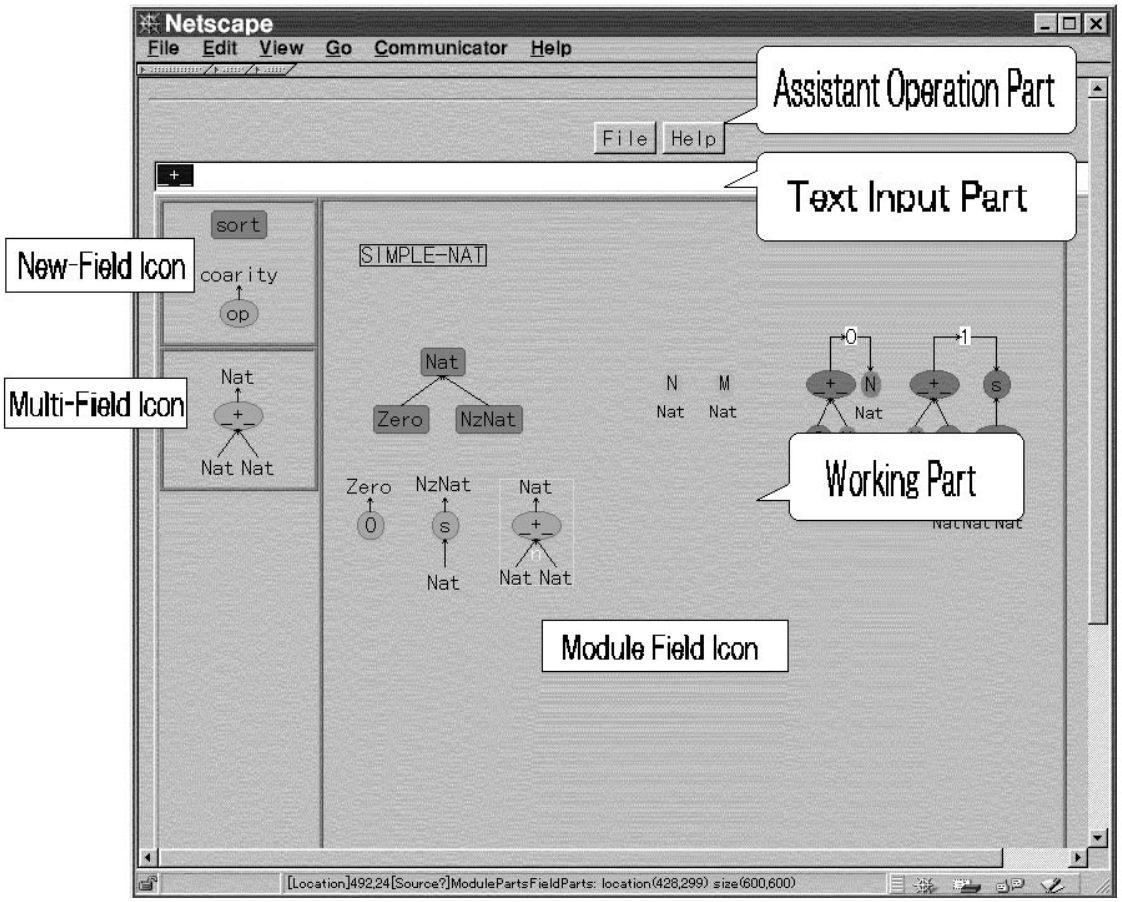

Figure 7: Snapshot of CafePie

Multi-Field and Module Field. User edits a program in the Working Part. New-Field is used to make a new icon (such as sort, operator, variable and equation) on the module. Multi-Field is used to delete and copy an icon. Module Field shows the current module to edit.

By using events which are defined in Table 1, a module “SIMPLE-NAT” (Fig. 8) is coded as follows (Fig. 9).

A module field icon has already been defined. User inputs the module name using the Text Input Part. A module has a default name (sort, operator, variable and equation have also default names). To change the name, user selects an icon (in this case, the icon is Module Field Icon), and inputs a name “SIMPLE-NAT” to Text Input Part. We define the contents of Modules as follows:

Make three new sorts (Zero, NzNat, Nat).
Add two new relations (Zero-Nat, NzNat-Nat).
Make three new operations (0, s, _+_).
Set coarity names (0:Zero, s:NzNat, _+_:Nat).
Add arities (0:none, s:Nat, _+_:Nat Nat).
Make two new variables (N, M).
Define the sort of the variables (N:Nat, M:Nat).
Make two new equations (0, 1).
Edit the left/right term of the equation ”0”
(left:0 + N, right:N).
Edit the left/right term of the equation ”1”
(left:N + s(M), right:s(N + M)).

To make the term “N + s(M)”, we need to perform the following operations (Fig. 10).

0. Find a term consisting of a variable (V) on a equation.
1. Move the operator “_+_” upon the variable (V).
   The variable (V) is replaced by the operator (_+_).
   and two new variables (V1, V2) appear.
2. Move the variable “N” upon the variable (V1).
   The variable (V1) is replaced by the variable (N).
3. Move the operator “s” upon the variable (V2).
   The variable (V2) is replaced by the operator (s).
4. Move the variable “M” upon the variable (V3).
   The variable (V3) is replaced by the variable (M).

Program Execution

In CafeOBJ, program execution is carried out by term rewriting. Our goal is visualizing the processes of term rewriting to show the execution processes in a more intuitive way[5]. The execution process is:

```
module SIMPLE-NAT {
 [ Zero NzNat < Nat ]
  signature {
    op 0 : -> Zero
    op s : Nat -> NzNat
    op _+_ : Nat Nat -> Nat
  }
  axioms {
    var N : Nat
    var M : Nat
    eq [0] : 0 + N = N .
    eq [1] : N + s(M) = s(N + M)
  }
}
```

Figure 8: SIMPLE-NAT in CafeOBJ

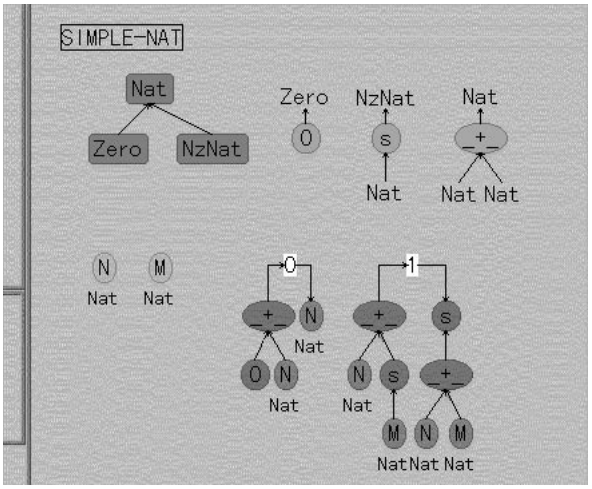

Figure 9: SIMPLE-NAT in CafePie

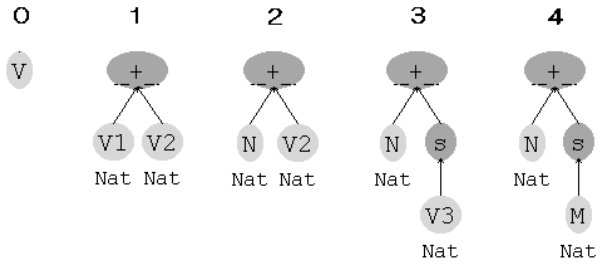

Figure 10: An Example of making a term

1. Make an initial term as an input (user can make a input term in the same way as editing modules)
2. Send the initial term to the external CafeOBJ interpreter[3].
3. Receive the execution result and make the tracing process for each rewriting step from the received data.
4. Show the tracing process.

```
SIMPLE-NAT> set trace on

SIMPLE-NAT> red s(s(0)) + s(s(s(0))) .
-- reduce in SIMPLE-NAT : s(s(0)) + s(s(s(0)))
1>[1] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> s(s(0)) }
1<[1] s(s(0)) + s(s(s(0))) --> s(s(s(0)) + s(s(0)))
1>[2] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> s(0) }
1<[2] s(s(0)) + s(s(0)) --> s(s(s(0)) + s(0))
1>[3] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> 0 }
1<[3] s(s(0)) + s(0) --> s(s(s(0)) + 0)
1>[4] rule: eq 0 + N:Nat = N:Nat
      { N:Nat |-> s(s(0)) }
1<[4] s(s(0)) + 0 --> s(s(0))
s(s(s(s(0)))) : NzNat
(0.010 sec for parse, 4 rewrites(0.070 sec), 10 match
 attempts)

SIMPLE-NAT>
```

Figure 11: Tracing Execution by CafeOBJ interpreter

Fig.11 shows how the term-rewriting is carried out on "SIMPLE-NAT" module (by CafeOBJ interpreter). The input term is $s(s(0)) + s(s(s(0)))$.

The term is edited on the "SIMPLE-NAT" module-field. Moving the term onto the label of the "SIMPLE-NAT" module involves the program execution. The process of the program execution is carried out as follows.

CafePie first sends the module definitions to the interpreter, and then queries the input term to the interpreter. Finally CafePie receives the execution trace as a result. The result is processed by CafePie and shown in the visualized form. Tracing result consists of terms which emphasizes the process of reductions (Table 2).

Term	Rule	Variable Replacement
$\underline{s(s(0)) + s(s(s(0)))}$		
$\Downarrow$	[1] N + s(M) = s(N + M)	N ⇒ s(s(0)), M ⇒ s(s(0))
$s(\underline{s(s(0)) + s(s(0))})$		
$\Downarrow$	[1] N + s(M) = s(N + M)	N ⇒ s(s(0)), M ⇒ s(0)
$s(s(\underline{s(s(0)) + s(0)}))$		
$\Downarrow$	[1] N + s(M) = s(N + M)	N ⇒ s(s(0)), M ⇒ 0
$s(s(s(\underline{s(0) + 0})))$		
$\Downarrow$	[0] 0 + N = N	N ⇒ s(s(0))
$s(s(s(s(0))))$		

Table 2: Term Rewriting Process

When CafePie gets the result, the input term is rewritten to the next term. CafePie shows the terms one after another like an animate cartoon. This is a dynamic representation and suits to check the rewriting flow at a time (Fig. 12). After showing the last term, CafePie represents the tracing diagram in Obi-shape (Fig. 13). This is a static display and suits to check one reduction process more closely[5]. To make the user know which equation is used for the reduction, the equation label is represented between the subterms on which the reduction is carried out. It is important dynamic representation to appeal the term rewriting.

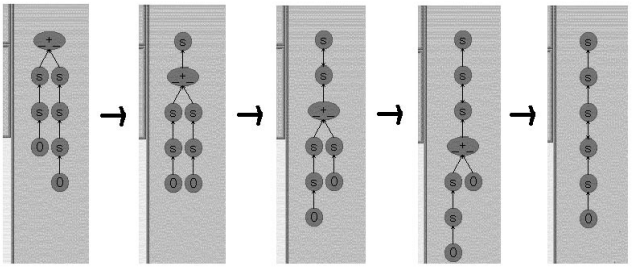

Figure 12: Dynamic Visualization of Execution

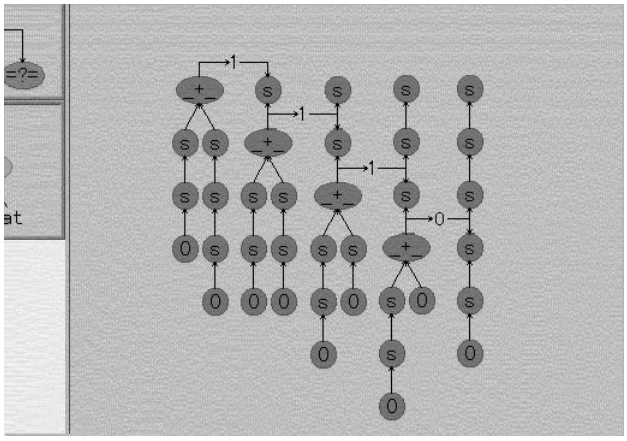

Figure 13: Static Visualization of Execution

Related Works

Various systems on which the user can watch and analyze term rewriting system (TRS) have been proposed.

ReDux[6] is a workbench for TRS realized by a text-interface. ReDux has various interface with Knuth-Bendix completion algorithm and so on. They comes up with various ideas in the text-interface to make the system to use. However the user cannnot manipulated terms intuitively.

TERSE[5] is a visual support environment for TRS. The system can show the proccess of term rewriting visually. The system supports the environment for the program execution, but not support the program editing. CafePie visually support not only program execution but also program editing. Users often understand the program through the execution and want to re-edit the program after that. Our strong point is that CafePie can edit and execute the program visually.

Summary and Further Research

We have implemented CafePie, a visual programming environment for CafeOBJ. The module structures are visualized with icons and can be edited intuitively using Drag-and-Drop. The execution process of the program, which is the term rewriting process for the initial term, is also visualized with icons. Program execution is described by using the same icon's description as the program editing.

Our system CafePie is useful for the beginners of ASL. And our goals are to improvve the system and to fascinate the advanced users. We think that more flexible user interface is desired.

Generally speaking, the program execution in ASL is defined by a process of term rewriting. It is important to verify programs in ASL. For the purpose of the verification, further research on the visualization of the program execution is needed.

REFERENCES

1. B. A. Myers. Taxonomies of Visual Programming and Programming Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
2. T. Ogawa. Drag and Drop based Visual Programing Environment for Algebraic Specification Language, 1997. Bachelor Thesis of College of Information Sciences, University of Tsukuba.
3. A. T. Nakagawa, T. Sawada, and K. Futatsugi. *CafeOBJ User's Manual*. IPA, 1997.
4. J. Tanaka. PP : Visual Programming System for Parallel Logic Programming Language GHC. *Parallel and Distributied Computing and Networks '97*, pages 188–193, August 11-13 1997. Singapore.
5. N. Kawaguchi, T. Sakabe, and Y. Inagaki. TERSE: Term Rewriting Support Environment. *Workshop on ML and its Application*, pages 91–100, florida, June 1994. ACM SIGPLAN.
6. R. Bundgen. Reduce the Redex $\rightarrow$ ReDuX. *Rewriting Techniques and Application*, LNCS 690, pages 446–450. Springer, 1993.