CafePie: A Visual Programming System for CafeOBJ

T. Ogawa * and J. Tanaka [a†]

[a]Institute of Information Sciences and Electronics, University of Tsukuba,
Tsukuba, Ibaraki 305-8573, Japan

CafePie is a visual programming system for CafeOBJ, an algebraic specification language based on term rewriting. Program editing and execution in CafePie are performed in one window. All program editing operations are handled in a uniform manner.

An abstract visualization schema is necessary to understand the program at the programming language level. In this paper, we propose visualized term rewriting with more realistic expressions. With our approach, users can customize the term expression as they like by using visual transformation rules. These rules can also be edited using drag-and-drop operations.

1. Introduction

"CafePie" [1–4], or CafeOBJ Pictorial Interactive Editor, is a visual programming system (VPS [5]). CafePie is a visual interface for the term rewriting portion of CafeOBJ. CafePie can be used as an environment for program editing. It visualizes each module of the CafeOBJ program and allows the user to edit it visually. Combined with the CafeOBJ interpreter, it can also serve as a visual rewriting environment for CafeOBJ.

Direct manipulation [6] is performed by using a mouse with CafePie. Each visualized element can be moved in accordance with the mouse movement. The same visualization schema is used for both program editing and execution. Since the program editing and execution are performed in one window, it is possible to directly reflect any program modification onto the program execution.

2. Features of Visual Programming

A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions. The latter is taken to be the definition of a visual programming language (VPL). VPLs may be further classified according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages [7]. Naturally visual languages have an visual expression for which there is no obvious textual equivalent [8]. Two-dimensional displays for programs, such as flowcharts and a program visualization system Incense [9], have long been known to

*tohru@softlab.is.tsukuba.ac.jp
†jiro@is.tsukuba.ac.jp

be helpful aids in program understanding. In addition, there have been much research on executable VPS, such as Pict [10], HI-VISUAL [11] and PP [12].

In general, a more visual style of programming could be easier to understand and generate for humans, especially for non-programmers or novice programmers since visual programming (VP) can be presented attractively. Moreover, the style is useful in software specification area, such as component based software [13,14]. A tool qualifies as a visual programming if it is possible to build some application without textual programming. However, currently very few practitioners use VP.

CafePie is based on the algebraic specification language (ASL) CafeOBJ, which is a high-level declarative programming language. A declarative programming language is suitable for visualization by a VPS because visual programming is also declarative. Visual expressions using the box-and-line representation have less commitment to the order of interpreting code than textual expressions. One can regard visual expressions to be declared spatially. A specification language requires comparatively fewer programming elements than a procedural programming language, and therefore specification language can be visualized with fewer kind of icons. Generally speaking, an environment with a user-friendly graphical interface has the advantage of enabling easy interpretation of the structures of the terms and rewriting processes.

3. The "CafePie" System

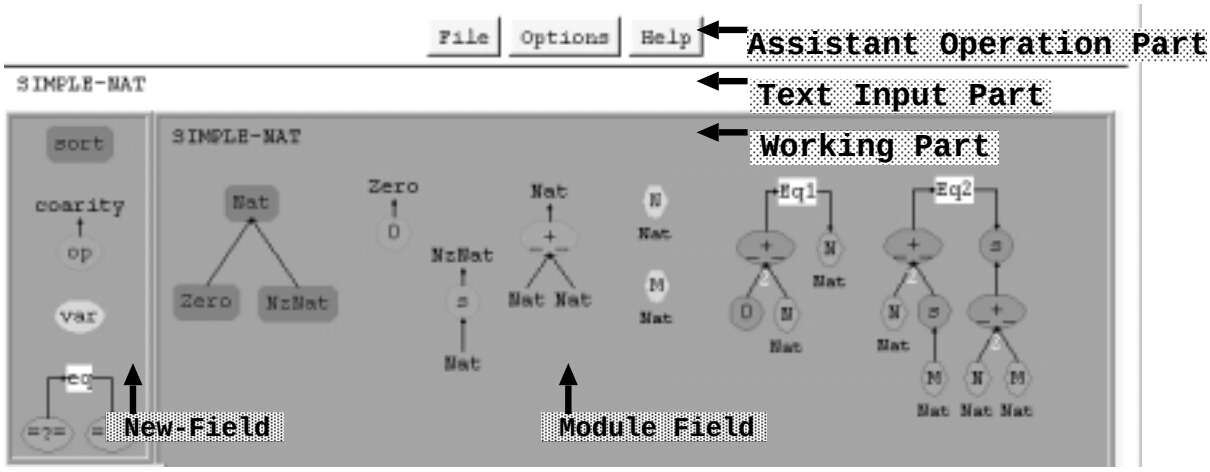


Figure 1. A Snapshot of CafePie

We developed CafePie and implemented it in Java. CafePie was developed in Java Development Kit (JDK™) version 1.0 at the first implementation, but the version is now 1.2. CafePie is ordinarily implemented as a Java application. In this version, users can edit and execute programs in the system. CafePie is also implemented as an applet on a Web browser. The applet version is used only for program editing.

Figure 1 shows a snapshot of CafePie. The upper part of the figure consists of buttons and is the "Assistant Operation Part." This enables the user to load or save a file (File button), set the CafeOBJ server (Options button), and view textual guides (Help button).

The "Text Input Part" is used to input the label of each editable icon. The main part of the figure shows the programming space called the "Working Part." The user edits a CafeOBJ program in this part. The "Module Field" in the Working Part shows the current CafeOBJ module to edit. The left side of the Module Field is the "New-Field," which consists of essential icons such as sort, operator, variable, and equation. This field is used to make a new icon in the Module Field.

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

Figure 2. "simple-nat.mod" –CafeOBJ Program File–

The following functions have been implemented in CafePie:

- Input program objects by figures.
  Users can input each basic object of an ASL language using an icon. These icons can be edited by direct manipulation.

- Generate visual icons from the codes automatically.
  Users can input a textual expression, and the system will generate icons from the expression.

- Editing visual objects.
  Visual expressions can be edited at any time. Users can program visually using this function while they edit or revise programs that have already been generated from the textual programs of CafeOBJ.

- Program save/load.
  Visually-edited programs can be saved to a file. Users load the file when necessary. CafePie saves the visual expressions after it converts them to CafeOBJ program expressions.

- Program execution.
  A goal (-term) represented by the visual icons can be executed. In this case, CafePie is connected to the CafeOBJ interpreter. CafePie behaves like a visual interface in the program execution.

For example, the file "simple-nat.mod"(Fig.2), which is a specification of natural numbers (under addition) written in CafeOBJ, is loaded by clicking on the File button. The program, visualized with pictorial objects, then appears in the Module Field of the Working Part (as in Fig. 1). The visualized program can be edited by direct manipulation. If the edited program is saved, a CafeOBJ program file and another textual file that contains layout information are created.

3.1. Program Visualization in CafePie

"Visualization of program structure" means expressing the program structure using pictorial or graphical objects. We visualize the program structures of CafeOBJ by expressing the program elements with pictorial objects. Each pictorial object is called an "Icon." We have chosen the following primitive elements for CafeOBJ: sort, operator, variable, and equation.

The visualization rules for each element are presented below.

Sort CafePie uses a directed graph to depict the sort orders. The sorts are represented in Fig.3 by green rectangular nodes (only shaded rectangles are seen in the manuscript) and the orders are represented by directed edges (Table 1).

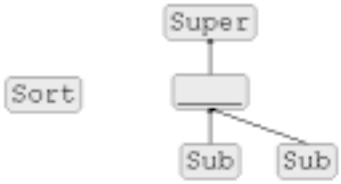


Figure 3. Sort Icon and Sort Relation

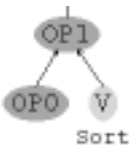


Figure 4. Term Icon

Term A term is formed with operators and variables. The structure of a term is displayed as a tree. Figure 4 shows the tree structure of the term "OP1 (OP0, V:Sort)." A component of a term, i.e. an operator or a variable, is represented by a node, and an arrow is drawn from the term to its superterm to express the super-sub relation between these components.

Operator and variable An operator is denoted by an operator symbol, its sort "coarity," and its attributes "arities." An operator is represented in Fig.5 by a light blue oval and

has a label for the operator symbol (Table 1). The labels of the arities are arranged at the bottom part of the operator, and the label of the coarity is arranged at the top part of the operator. Arrows are drawn from arities to operator and from operator to coarity. A variable, which appears in Fig.6, is represented by an orange oval (Table 1), and the sort of the variable is represented at its lower part.

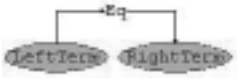


Figure 5. Operator Icon Figure 6. Variable Icon Figure 7. Equation Icon

Equation CafePie is mainly concerned with the operational semantics of CafeOBJ, so equations are always regarded as rewrite rules. A label is arranged in the center top of the equation, as shown in Fig.7. The left side is arranged on the bottom left side of the label, and the right side is on the bottom right. Arrows from the left term to the label and from the label to the right term are drawn to form a balanced shape to represent a term rewriting rule (Table 1).

Table 1
Icons' Colors and Shapes in CafePie

Icon	Sort	Term	Operator	Variable	Equation	Module
Color	Green	_	Light blue	Orange	White (label)	Gray
Shape	Rectangle	(Tree)	Oval	Oval	(Balance)	Field (Rectangle)

Module Field A CafeOBJ program consists of modules. A module is represented as a gray rectangle called a "field" (Table 1). The module contains other primitive elements: sort, operator, variable, and equation. We can edit these primitive elements.

3.2. Drag-and-Drop-based Program Editing

We use direct manipulation to implement program editing. Direct manipulation is easy to learn, and the user can immediately recognize any mistakes. Complex and obscure operations can cause unexpected consequences; simple operations enable more smooth program editing.

All icon-editing operations are handled in a uniform manner, using a drag-and-drop operation [15]. This drag-and-drop technique is well known for its simplicity. For icon movement, the user moves the icon using the drag-and-drop technique. If an icon already exists where the user wants to drop the icon, the two icons will overlap. Overlapping two

icons with the drag-and-drop technique is important in the editing process. The process of the drag-and-drop method consists of

1. Selecting an icon,
2. Moving (or dragging) the selected icon to another icon, and
3. Overlapping (or dropping) the selected icon with another icon.

The target icon moves with the mouse cursor and remains visible throughout the movement. The user moves the icon by dragging it, without losing sight of what he is doing. We reexamined this technique to realize program editing. Program editing operations in CafePie involve making/deleting a relation between two sorts, adding/changing an arity of an operator, and creating/adding a subterm on a variable. Table 2 shows these program editing operations. An event is invoked when an icon (*source*) is overlapped onto another icon (*target*). After the event is invoked, the action corresponding to the event is carried out. The program editing process is the repetition of these elementary actions.

Table 2
Drag-and-Drop-based Program Operations in CafePie

Event Name	Source	Target	Action
Make Sort-Relation	Sort	Sort	Relate one sort to another (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	Replace the variable with a new term
Add Subterm	Term	Variable	Replace the variable with the (copied) term

For example, operator "`s`," which appears in the sample code SIMPLE-NAT, has an arity sort called "`Nat`" ("`op s : Nat -> NzNat`"). This operator is created from several steps.

- First, an operator icon that has no arity (constant) is created by default.
- Next, the sort icon "`Nat`" which has already been defined is moved toward the operator.
- Finally, these two icons are overlapped, the "Add Arity" event (in Table 2) is carried out, and the arity sort called "`Nat`" is added to the operator.

Another example is called "Create Subterm." The left term of the equation "`1`," which appears in the SIMPLE-NAT, is "`N:Nat + s(M:Nat)`." Operators "`_+_`" and "`s`" are used to create this term.

- Suppose there is a variable that belongs to the sort "`Nat`."

- Moving the operator "_+_" onto the variable changes the variable to the term "`V1:Nat + V2:Nat`".
- Similarly, moving the operator "`s`" onto the variable "`V2`" (of the term) changes the variable "`V2`" to the term "`V1:Nat + s(V2:Nat)`".

In this way, the drag-and-drop technique is applied to CafePie. All operations of the program editing are handled in a uniform manner.

3.3. Program Execution in CafePie

CafePie enables the program execution by combining with CafeOBJ interpreter. In order to utilize the interpreter, CafePie must communicate with "cafemaster," which is a network server for CafeOBJ. CafePie and the interpreter are connected by cafemaster. (Cafemaster has two modes for combining a client with the interpreter, i.e, the session mode and the interactive mode. In the current implementation, CafePie accesses the interpreter in the interactive mode.)

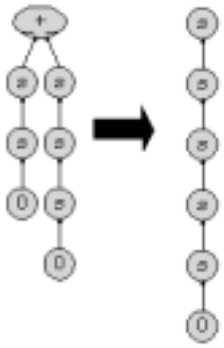
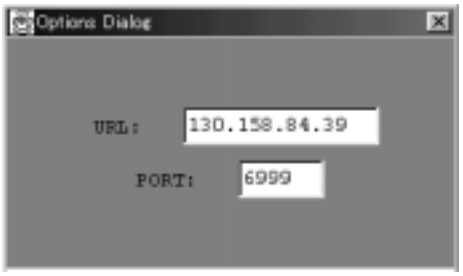


Figure 8. A Goal Term for Program Execution




```
URL: Users can designate an IP address or URL
     as the location of the interpreter.

PORT: Users can input the port number
     of the CafeOBJ interpreter.
```

Figure 9. Options Dialog of CafePie

- **Edit a goal term:**
  A user edits a term (goal) in the Module Field. It is called a goal and is used to test the module SIMPLE-NAT. For example, we create the goal "`s(s(0)) + s(s(s(0)))`" (the left side of Fig.8).

- **Start the term rewriting:**
  A program consists of a module displayed in the Module Field. Each module has a label. The label is drawn at the upper left of Module Field (Fig.1). The user invokes evaluation (program execution) by moving the term onto the label.

- **Connect to the interpreter:**
  CafePie tries to connect to the interpreter running on a remote host by using socket communication. If a connection is achieved, CafePie connects to the interpreter in an interactive mode (CafePie sends a message "`interactive`" to the interpreter). Users can specify the interpreter's network address. They click on the Options button of the Assistant Operation Part, and an options dialog appears on CafePie (Fig.9). The IP address and the port of the CafeOBJ interpreter are designated in the dialog. Thereafter, CafePie knows where the interpreter is.

- **Send the module information to the interpreter:**
  After connecting to the interpreter, CafePie converts the module's visual expression into a text-based CafeOBJ program and sends the program to the interpreter. The information is comprised of a module name, sorts, operations, variables and equations (Fig.2).

- **Send the goal to the interpreter:**
  After sending the program, CafePie sends the goal term "`s(s(0)) + s(s(s(0)))`" to the interpreter. CafePie orders the interpreter to start the program execution (CafePie sends two messages, "`set trace on`" and "`red s(s(0)) + s(s(s(0))) .`", Fig.12).

- **Receive the result from the interpreter:**
  The goal is rewritten repeatedly on the interpreter. CafePie receives the term rewriting trace as a result after execution is completed (Fig. 12). The tracing result consists of terms that illustrate the process of reductions. The result is processed by CafePie and is shown in the visualized form.

CafePie shows the terms in succession like an animated cartoon. This is a dynamic representation and is suitable for checking the rewriting flow at any time. Figure 10 shows the process of term rewriting when the goal term is "`s(s(0)) + s(s(s(0)))`" of the module SIMPLE-NAT and the rewritten term is "`s(s(s(s(s(0)))))`" (the right side of Fig. 8). This is an effective dynamic representation of the term rewriting process. After showing the last term, CafePie presents the tracing diagram in the shape of an obi (an obi is a Japanese broad sash tied over a kimono, Fig. 11). This is a static display and is suitable for checking one reduction process more closely.

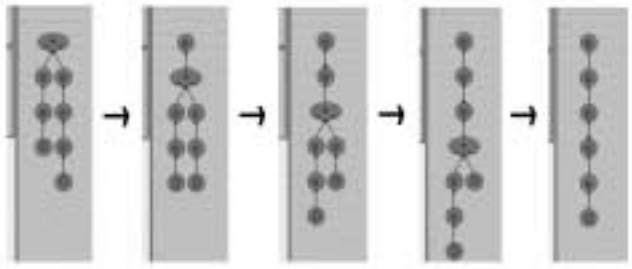


Figure 10. Dynamic Representation

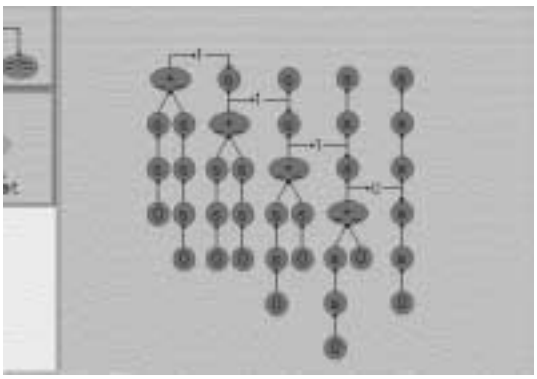


Figure 11. Static Representation

3.4. Realistic Visualization

The process of term rewriting is visualized as tree structures that consist of icons. For example, Fig.13 shows the result of visualization of the term "`push( E3:Elt, push( E2:Elt, push( E1:Elt, push( E0:Elt, empty ))))`," by CafePie. The specification that begets this term is expressed as module STACK (Fig.15).

This visualization method is difficult for users to understand in an intuitive manner because users in general mentally visualize stacks not as trees but as building blocks. More realistic expressions of higher abstraction levels are desired.

3.4.1. Visual Transformation Rule and Term Visualization Example (1)

We propose a method that visualizes term rewriting with more realistic expressions, by using figures, pictures, and images. We call these expressions visual objects. The visual objects should be edited without the program code showing. In the real world, the essential characteristic of an actual object is its shape. We propose to use visual transformation rules so that users can change the shapes of visual objects.

Rewriting rules are called "equations" in CafeOBJ. An equation is composed of operators and variables. CafePie can change the term representation. Specifically, it can change the system-prepared view to a user-defined view by using visual transformation rules. For example, the STACK program of CafeOBJ has the operators "`empty`" and "`push`." By default, the expression of these operators have been prepared by the system (the left part of Fig.16 and 17). If a user imagines that the STACK is like building blocks, the result of STACK visualization would be like building blocks. The operator "`empty`" is represented by a rectangle (the right part of Fig. 16) to imitate building blocks, instead of the original visualization (the left part of Fig. 16). The operator "`push`" is visualized like the right part of Fig. 17. This figure shows that the rectangle with "Elt" is arranged at the upper part of "Stack." After defining these rules, the old terms (Fig. 13) are changed to other expressions, as in Fig. 14.

```
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 12. An Execution Result of the CafeOBJ Interpreter

3.4.2. Defining the Visual Transformation Rule

We have developed an environment in which users can edit the visual transformation rules in CafePie by using direct manipulation. In our approach, these rules are not defined by using a drawing action but by using a combination of prepared visual objects. Therefore, the users can easily define the rules and edit programs in the same paradigm. Editing the rules requires the two steps below.

1. Preparing the visual objects. The system has some elementary figures such as rectangles and circles, and images that are loaded from files. If an operator has arities, users can also use them as visual objects.

2. Defining the geometrical relations. The user creates a repeated relation between two objects to define the geometrical relations. The users can also treat related objects as one object.

The relation is provided by the drag-and-drop operation. We use a "plate"-node icon. When a node is dropped onto the plate, the node is added on the plate, and its position is arranged automatically. Suppose the user moves a node toward the plate-node. When the user drops the node onto the plate, dotted lines appear around the plate, as shown in Fig.18. These lines indicate the expected location of the node. The node's location

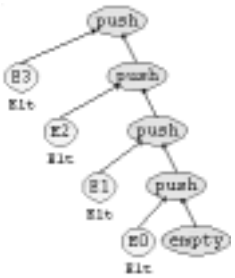


Figure 13. Original Stack Visualization

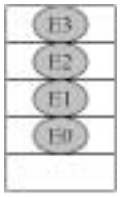


Figure 14. New Stack Visualization

```
module STACK [X::TRIV] {
 [ NeStack < Stack ]
  signature {
    op empty : -> Stack
    op push : Elt Stack -> NeStack
    op pop : NeStack -> Stack
    op top : NeStack -> Elt
  }
  axioms {
    var S : Stack
    var E : Elt
    eq pop( push( E, S ) ) = S .
    eq top( push( E, S ) ) = E .
  }
}
```

Figure 15. "stack.mod" –CafeOBJ Program File–

is selected by default from any of the nine parts of the plate: the upper left, the upper middle, the upper right, the left side, the center, the right side, the lower left, the lower middle, or the lower right. The location of the dropped node is determined as follows:

- If the node is dropped on the upper or lower part of the plate, the node is designed to stick to the plate.
- If the node is dropped on the right or left side of the plate, they are also arranged to be close together.
- If the node is dropped in the diagonal part of the plate, the node center is arranged on the vertex of the plate.

The size of the dropped node is determined by the node's location.

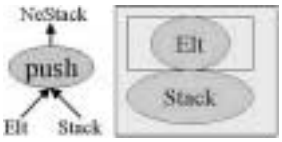


Figure 16. Empty Operator Visualization(1) Figure 17. Push Operator Visualization(1)

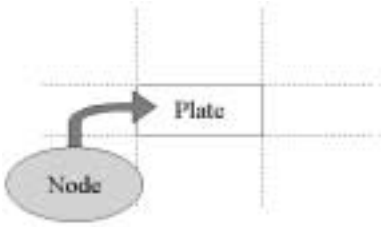


Figure 18. Make Visual Transformation Rule

- If the user drops the node in the center of the plate, the node's size becomes smaller than the plate. The plate contains the node.
- If the node is dropped in the left or right part of the plate, the node's height is modified to have the same height as the plate.
- If the node is dropped in the upper or lower part of the plate, the node's width is modified to have the same width as the plate.
- If the node is dropped in the diagonal part of the plate, the node's size is changed to be the same size of the plate.

3.4.3. Term Visualization Example (2)

Another visualization method can be applied to STACK instead of the example using building blocks.

The visual transformation rules of the operators "`empty`" and "`push`" can be re-defined. The right hand side of Fig.19 shows the new rule of the operator "`empty`." This figure indicates "No Exit" because the Exit door has broken down. The right hand side of Fig.20 shows the new rule of the operator "`push`." This figure indicates that a person who has a face "`Elt`" is in the rear of the "`Stack`." Figure 21 shows a term according to the new visualization rules. Each person has a different expression. No person can go forward because of the broken door. Only the person who is at the end of the line can move. This mechanism represents the STACK structure. In this visualization, STACK represents a line of people. Programs can be expressed differently in this way by defining different visual transformation rules.

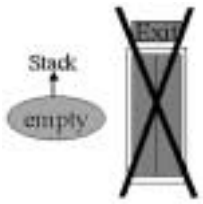


Figure 19. Empty Operator Visualization(2)

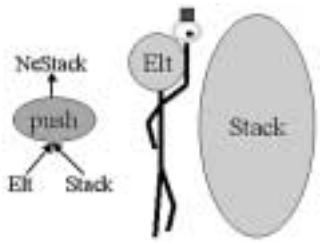


Figure 20. Push Operator Visualization(2)

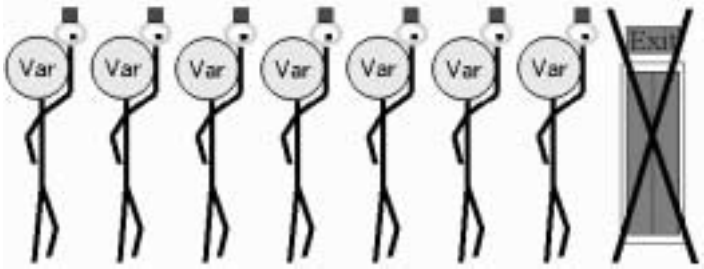


Figure 21. New Stack Visualization (2)

4. Related Works

Various systems have been proposed through which users can watch and analyze the term-rewriting system (TRS). ReDux [16] is a workbench for TRS realized by a textual interface. ReDux has various interfaces with completion algorithms. They came up with various concepts in the text interface. However, users cannot manipulate the terms intuitively. TERSE [17] is a visual support environment for TRS. The system can visually show the process of term rewriting. The system supports the environment for program execution, but does not support program editing. CafePie visually supports not only program execution but also program editing. Users often understand the program through the execution and want to subsequently re-edit the program. Our main point is that CafePie can edit and execute the program visually. CafePie is the first system that shows TRS execution dynamically. Viry presents some preliminary ideas towards a user interface for completion and its integration within programming environments [18].

SDL [19], G-LOTOS [20,21] and Petri Nets [22] are graphics-based specification languages. SDL is a specification language with both graphical and character-based syntaxes for defining interacting extended finite state machines, and is used to specify discrete interactive systems such as industrial process control, traffic control, and telecommunication systems. G-LOTOS, which has two-dimensional constructions, enables LOTOS to express the specification diagrammatically. Petri Nets is applied to the modeling and analysis of computer architecture problems, and has a graphical and formal syntaxes.

In addition, various kinds of VPLs have been proposed. Form/3 [23] is a declarative, form-based, language that follows the spreadsheet paradigm. ChemTrains [24] is a rule-based language in which both the condition and action of each rule are specified by pictures.

5. Summary and Further Research

We have implemented CafePie, a VPS for CafeOBJ. The module structures are visualized with icons and can be edited intuitively using the drag-and-drop technique. 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 iconic descriptions as in program editing. Term rewriting is visualized with realistic expressions by using figures, pictures, and images. We map operators to realistic expressions so that equations are expressed as transformations of realistic expressions. We use visual transformation rules that give the program pictorial expressions so that users can customize the term expression to their preference.

Our system, CafePie, is useful for ASL beginners. Our goal is to improve the system and to fascinate advanced users. Shneiderman [25] stated that direct manipulation is not appropriate when the data structure to be displayed is large, which can very easily happen with an application to algebraic specification. Another issue that must be discussed is the help for browsing specifications, which in CafeOBJ has a modular structure, because research has shown that users spend a lot of time trying to get their specifications just right.

REFERENCES

1. T. Ogawa and J. Tanaka. Drag and Drop based Visual Programming Environment for Algebraic Specification Language. In *15th Conference Proceedings Japan Society for Software Science and Technology(JSSST-98)*, pages 165–168, 1998. (in Japanese).
2. T. Ogawa and J. Tanaka. Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ. In *Proceedings of International Symposium on Future Software Technology (ISFST'98)*, pages 155–160, Hangzhou, October 28-30 1998.
3. T. Ogawa and J. Tanaka. Realistic Program Visualization in CafePie. In *Proceedings of World Conference on Integrated Design and Process Technology (IDPT'99)*, 1999. (to appear).
4. T. Ogawa and J. Tanaka. CafePie: CafeOBJ Visualization by using a Combination of Diagrams. In *16th Conference Proceedings Japan Society for Software Science and Technology(JSSST-99)*, pages 65–68, 1999. (in Japanese).
5. B. A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
6. B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):57–69, 1983.
7. E. J. Golin and S. P. Reiss. The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing*, 1(2):141–157, 1990.
8. A. L. Ambler and M. M. Burnett. Influence of Visual Technology on the Evolution of Language Environments. *IEEE Computer*, 6(2):9–22, October 1989.

9. Brad A. Myers. Incense: A System for Displaying Data Structures. *Computer Graphics: SIGGRAPH '83 Conference Proceedings*, 17(3):115–125, July 1983.
10. E. Glinert and S. Tanimoto. PICT: An Interactive Graphical Programming Environment. *IEEE Computer*, 17(11):7–25, 1984.
11. M. Hirakawa, M. Tanaka, and T. Ichikawa. An Iconic Programming System, HI-VISUAL. *IEEE Transaction on Software Engineering*, 16(10):1178–1184, 1990.
12. 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.
13. M. P. Stovsky and B. W. Weide. Building Interprocess Communication Models Using Stile. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 566–574. IEEE Computer Society Press, Los Alamitos, 1990.
14. D. C. Smith and J. Susser. A Component Architecture for Personal Computer Software. In B. A. Myers, editor, *Languages for Developing User Interfaces*, pages 31–56. Jones and Bartlett Publishers, Boston, 1992.
15. A. Wagner, P.Curran, and R. O'Brien. Drag Me, Drop Me, Treat Me Like an Object. In *Proceedings of CHI'95: Human Factors in Computing Systems*, pages 525–530, 1995.
16. R. Bundgen. Reduce the Redex → ReDuX. In *Rewriting Techniques and Applications*, LNCS 690, pages 446–450. Springer, 1993.
17. N. Kawaguchi, T. Sakabe, and Y. Inagaki. TERSE: TErm Rewriting Support Environment. In *Workshop on ML and its Application*, pages 91–100, florida, june 1994. ACM SIGPLAN.
18. P. Viry. A user-interface for Knuth-Bendix completion. In *4th Workshop on User Interfaces for Theorem Provers (UITP'98)*, July 1998.
19. R. Saracco, J. Smith, and R. Reed. *Telecommunications Systems Engineering using SDL*. North-Holland, Elsevier Science Publishers, Amsterdam, 1989.
20. E. Najm (ed.). G-LOTOS: DAM1 to ISO8807 on graphical representation for LOTOS. Technical report, ISO/IEC JTC 1 / SC 21 N. 4871, 1992.
21. T.Bolognesi and D.Latella. Techniques for the formal definition of the G-LOTOS syntax. In *Procceeing of the 1987 IEEE Workshop on Visual Languages (VL'89)*, Roma, 1987.
22. J. L. Peterson. *Petri Net Theory and The Modeling of Systems*. Prentice-Hall, 1981.
23. M. M. Burnett and A. L. Ambler. A Declarative Approach to Event-handling in Visual Programming Languages. In *Proceedings of the 1992 IEEE Workshop Visual Languages (VL'92)*, pages 34–40, Seattle, Washington, September 1992.
24. B. Bell and C. Lewis. ChemTrains: A Language for Creating Behaving Pictures. In *Proceedings of the 1993 IEEE Symposium Visual Languages (VL'93)*, pages 188–195, Bergen, Norway, August 1993.
25. B. Shneiderman. *Designing the User Interface (Third Edition)*. Addison-Wesley Publishing Company, 1997.