Visualization of Program Execution via Customized View

Tohru Ogawa and Jiro Tanaka [a]

[a] IPLAB, Institute of Information Sciences and Electronics,
University of Tsukuba, Tsukuba, 305-8573, Ibaraki, Japan
{tohru, jiro}@iplab.is.tsukuba.ac.jp

ABSTRACT

Visual programming systems use their original notation to represent the programs. The users have difficulty for understanding programs through the observation of program execution. An understandable representation is desired.

We propose a method of view customization in a visual programming system. The objects of the view customization are the data structures, which is called "term," in a term rewriting system. Term rewriting systems have an important feature, that is, a program execution can be expressed with terms. For the feature, the user-customized view is useful not only in the data structures but also in the program execution. We describe methods for emphatic representation of the program execution with the customized views. We have also implemented the view customization mechanism in our visual programming system.

Keywords: Visual Programming, Program Visualization, Term Rewriting System

1. INTRODUCTION

Visual Programming System (VPS)[1] represents programs visually using visual expressions such as graphics, pictures and so on. Various research works have been performed on VPS, such as Pict[2], HI-VISUAL[3] and PP[4].

1.1. Underlying Visual Programming System

We distinguish a pure VPS from a visually supported system by the following two criteria. First criterion is that the specification of the program is modifiable within the visual environment of the system. Second criterion is that the VPS executes the program, which is the specification of the executable program is modifiable. The system is not only a drawing tool for software.

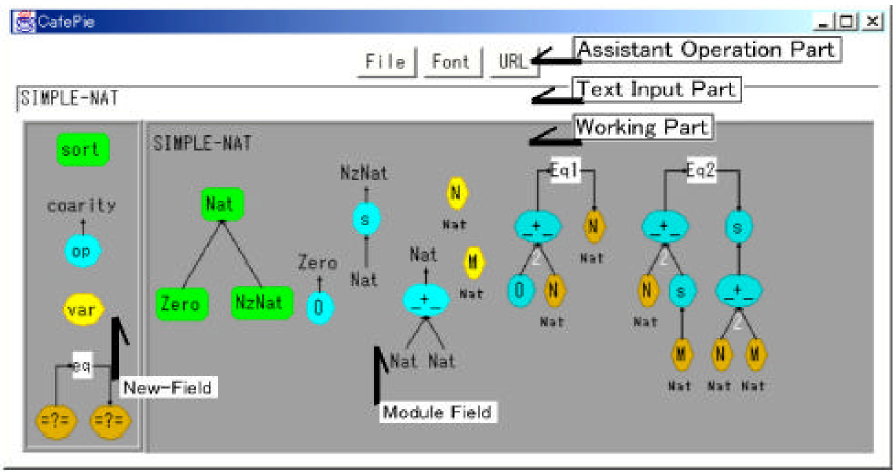

Figure 1. A Snapshot of our system "CafePie."

We have developed a VPS called "CafePie" [5,6,7], which stands for Pictorial Interactive Environment for "CafeOBJ"[8], an algebraic specification language. CafeOBJ is a high-level declarative programming language and its specification consists of module structures. Our system CafePie visualizes each module. Figure 1 shows a snapshot of CafePie. We use the direct manipulation techniques for program editing. Most of the editing operations are performed using only a mouse. Since the program editing and execution are performed in one window, program modifications are reflected directly in program execution. CafePie enables the program execution by combining with CafeOBJ interpreter. CafePie utilizes the interpreter via "Cafemaster," which is a network server of CafeOBJ. CafeOBJ also visualizes a trace of Term Rewriting System (TRS), which is the program execution of CafeOBJ.

1.2. Pre-defined Visual Notation in the VPS

A term is the data structure of the CafeOBJ language. One term is drawn as a tree. For example, figure 2 represents a term of a STACK program[6]. The textual notation of the figure is "push(E1:Nat, push(E2:Nat, push(E3:Nat, empty)))." The term contains two elements "push" and "empty." The "push" has two attributes and the "empty" has no attribute. "E1," "E2" and "E3" are the labels of "Nat." "Nat" means the type of natural numbers (a SIMPLE-NAT program[6]) and it is the element of STACK.

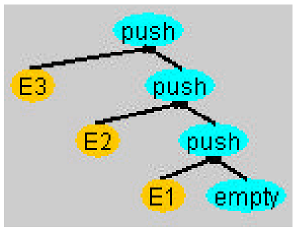

Figure 2. A term representation of a STACK.

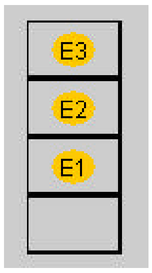

Figure 3. Another visualization of STACK.

2. VIEW CUSTOMIZATION

The VPS represents the data structures in their original notation. The users have dfficulties in changing the representation depending on the situation where the program is used. For example, STACK may be visualized so that its behavior is easily understandable. If we consider STACK as a container of data, our interest is the data its elf. Another kind of view can be taken into consideration. The view is the graphical representation of the data structure. We propose a method of view customization in a VPS.

2.1. Approach for View Customization

We need to make a distinction between the model and the view of the data structure. A general idea of the view is used in MVC (Model-View-Control) model[9]. One concept of MVC model is a programming technique for implementing interface part of software. We utilize the concept of the view for VPS.

Programs consist of the data structures and control-structures. We apply the view customization to the data structures. The view customization changes only two factors: one is each constituent's appearance of the data structure; another is the layout between the constituents. The data structure called "term" is composed of operators and variables in TRS. In our approach, the object of the view customization is the operator. The definition of the control-structure called "equation" is a rewriting rule. The rewriting rules make the algorithm of the program. The equation consists of the following terms: the initial term and the written term (and the conditional term if necessary). In our approach, equation is also visualized by using the customized views.

A term expresses a static state of TRS calculation. The calculation of TRS proceeds only by rewriting the term. For this simple calculation, the program execution is expressed by using the trace of the term rewritings. The same visualization schema can be used for both program editing and execution. In other words, if we customize the view of the terms, the customized view can be utilized in the program editing and execution.

2.2. Example of View Customization

A term, which is a data structure, can be visualized as a tree that consists of visual components shown in figure 2. This visualization method is difficult for users to understand in an intuitive manner because they mentally visualize STACK as building blocks, not as a tree. A more "realis tic" visualization scheme is desired. The "realistic" visualization means a framework to denote the meaning of the program by its appearance. The users can guess the meaning easily by just looking at the program.

For example, figure 2 is a representation of the term "push(E3:Elt,push(E2:Elt,push(E1:Elt,push(E0:Elt,empty)))),” and it expresses STACK. In this case, the term is represented by a tree structure, the system-defined view. If a view designer imagines that STACK is like building blocks, another visualization is enabled. In this example, the view designer makes two rules on the operators: "empty" and "push" shown in figure 2. After making the two rules, STACK is visualized like in figure 3. Rule 1: Instead of the original representation, the operator "empty" is represented by a rectangle shown in the left part of figure 4 to imitate building blocks. Rule 2: Instead of the original representation, the operator "push" is visualized like the right part of figure 4. This figure shows that the rectangle with "Elt" is arranged at the upper part of "Stack."

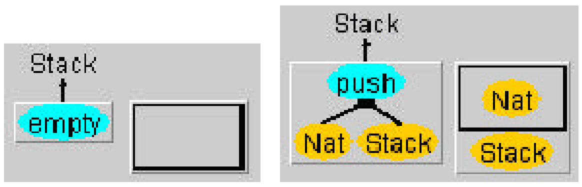

Figure 4. A visualization of "empty" and "push."

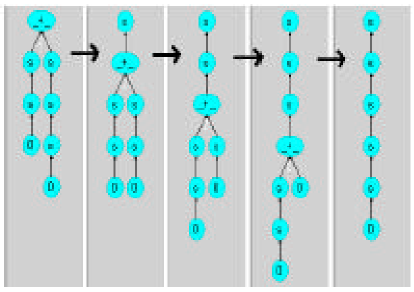

Figure 5. A visualization of the program execution trace.

The mechanism of our view customization enables the change of the program representations. The editing operation of the customization is drag-and-drop operation. We have known that the operation does not take much time in comparison with the program editing[7].

3. PROGRAM EXECUTION WITH CUSTOMIZED VIEW

Through the program execution, a programmer observes that the behavior of the editing program is just as expected. After the observation, (s)he understands the meaning of the program, or confirms that the behavior of the program is the same as the expectation. If the result of the execution is not the same as his expectation, he may debug the program.

3.1. Visualization of Program Execution

Watching the output is a simple technique for a program observation. The programmer can understand in detail by watching not only the output but also the trace of the program execution. Figure 5 is a representation of a program execution trace. The program is a SIMPLE-NAT program[6], natural numbers under addition. The left end of figure 5 is the input term and the right end of figure 5 is the output term. The input term called "goal" is "s(s(0)) + s(s(s(0)))" and the output term is "s(s(s(s(s(0)))))."

3.2. View Customization for Program Execution

The process for the program execution is as follows. First, a programmer makes the "equations" for defining algorithms. Next, (s)he makes a goal for evaluating. The programmer changes the view of the goal if necessary. At last, the programmer evaluates the goal.

3.2.1. Making an Equation

Let us take a sorting algorithm as an example. We change the meaning of the operator "push" defined before. In other word, we make a new equation by using the operator. In this case, defining the equation makes a sorting algorithm. After defining the equation, a STACK program with the changed operator is considered a SORTED-STACK program.

Figure 6 represents the equation called "eq1," the definition of a sorting algorithm. The visual notation of the "eq1" is pre-defined in the system. The left hand side of the equation is called an initial term, the right left hand side of the equation is called a written term and the upper part of the equation is called conditional term. This equation means that the left hand term is written to the right hand term, if the condition gets "true." The textual notation of "eq1" is "push(N:NAT, push(M:NAT,empty)) = push(M:Nat, push(N:Nat, empty)) if (N gt M)." The term "(N gt M)" is a condition by which the operator "gt" influences. The operator "gt" is a binary operator. If the left attribute "N" is greater than the right attribute "M," then the operator " gt" returns "true." After "gt" returns "true," the two variables "N" and "M" are exchanged.

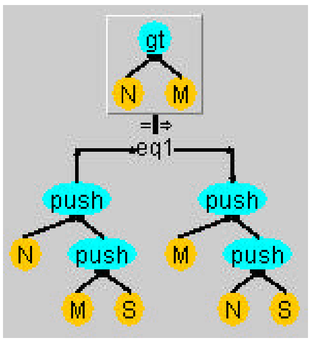

Figure 6. The definition of the equation "eq1" with a condition.

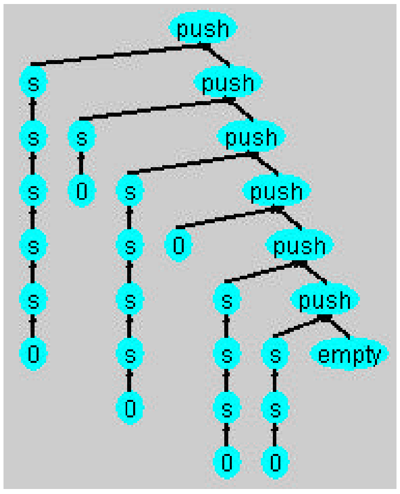

Figure 7. A goal of the sorting algorithm.

3.2.2. Making Goal and Changing its View

A goal represented by a tree structure is given as shown figure 7. We want a suitable view like the left part of figure 8 for showing the sorting algorithm.

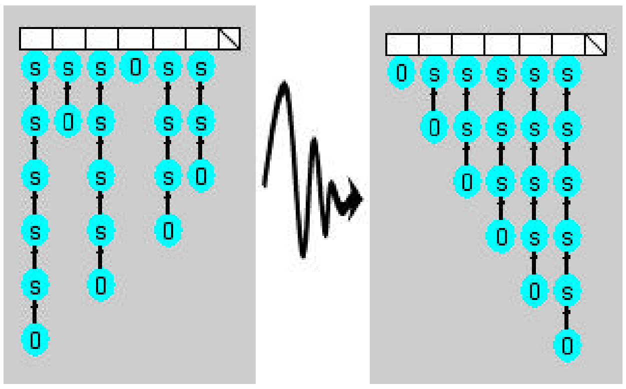

Figure 8. A program execution with customized view.

In this case, we make a new view on the operator "push." A procedure for changing the view is as follows. First step is selecting a "push" on the goal. Second step is calling the original operator "push," which has a definition of its view, by using a menu. Third step is making a new customized view on the operator "push" if necessary. Last step is reflecting the view on the goal.

We have the following four types for reflecting the view on the goal:

1. Reflecting on the selected "push" only,

2. Reflecting on the selected "push" and its attributes,
3. Reflecting on the goal include the selected "push,"
4. Reflecting on the whole term in the program.

In this case, we select the third type and the whole "push" in the goal will be changed. The result of the goal is shown in the left part of figure 8.

We show the views of the operators for customizing the goal below. The left part of figure 9 shows a new customized view for the SORTED-STACK operator "push." The "push" is represented by a rectangle and is arranged at the left part of the "Stack." The "Nat" is arranged at the lower part of the rectangle.

The right part of figure 9 shows a new customized view for the operator "empty." The operator "empty" is represented by a rectangle with a diagonal line.

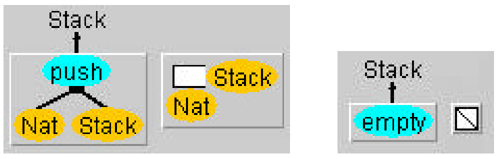

Figure 9. Another visualization of the operator "push."

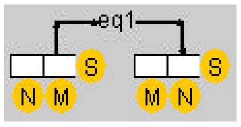

Figure 10. Visualization of the equation "eq1."

3.2.3. Changing View of Equation

We can also change the view of the equation, shown in figure 7, by using the same way of making the goal. Figure 10 is the equation after changing its view (The condition state of the equation is not shown in this figure).

3.2.4. Representation of Program Execution

Program execution also can be represented by the customized view. Figure 8 shows a program execution in STACK. The given goal is "push(s(s(s(s(s(0))))), push(s(0), push(s(s(s(s(0)))), push(0, push(s(s(s(0))), push(s(0)), empty))))))," and the result is "push(0, push(s(0), push(s(s(0)), push(s(s(s(0))), push(s(s(s(s(0)))), push(s(s(s(s(s(0))))), empty))))))."

The goal in the left part of figure 8 is written to the right part of figure 8 on the system. The system shows snapshots of the trace in a fixed interval one after another.

3.3. Representation of Data Structures

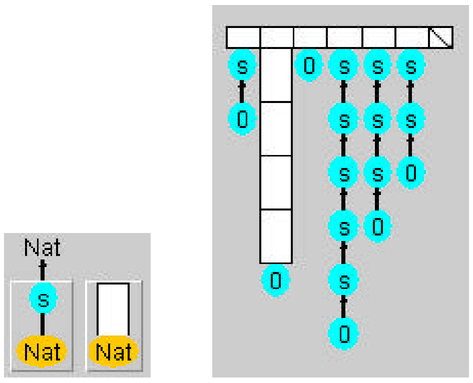

Figure 11. Highlighted representation of the data structures.

The view customization framework can utilize highlighted representation of the data structures for the program execution. Let us suppose that a user runs his editing program. If the behavior of the program is not as expected, he knows his program has errors. He may want to observe the program execution by paying attention to the unreliable codes of the program.

For example, we can highlight one of the SORTED-STACK data by using the view customization. In this case, we highlight the "s(s(s(s(0))))" in the goal shown in figure 7. We customize the view of the operator "s" such as the left part of figure 11. This figure means that the appearance of operator "s" is chang ed to a rectangle. After the view changing, the goal is represented as shown in the right part of figure 11. This figure represents one process of the program execution. In this case, we select a second type of view-refection shown in section 3.2.2. The part of the goal is emphasized. We can recognize where the interesting data is in the right part of figure 11.

3.4. Animation with the Control-Structures

We show another emphatic representation, highlighted animation with the control structures, for the program execution with customized views.

CafePie can display the trace of the program execution dynamically. In the case of the dynamic display, the given goal is rewritten in the same position. CafePie shows snapshots of the trace in a fixed interval one after another. This animation is finished when the goal is rewritten to the end.

We consider that the meaning of equation is not only a rewriting rule but also an animation rule in the dynamic display. The animation rule enables the program execution display smoothly. The animation rule has two kinds of information, interpolating comp onents and its display timing.

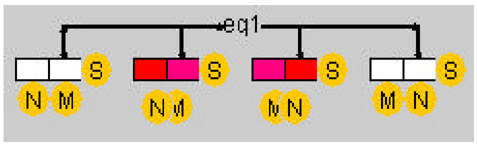

Figure 12. An animation rule on the equation "eq1."

For example, we expand the equation "eq1" shown in figure 6. The two middle parts of figure 12, which are made up for showing snapshot smoothly, are inserted between the left part of figure 12 and the right. In this case, this figure shows that the two middle parts are displayed at 1/3 of the whole timing. The program execution can be performed with the smoother highlighted animation like in figure 13. In this way, the program execution is emphasized by animation rule defined upon the expanded control-structures.

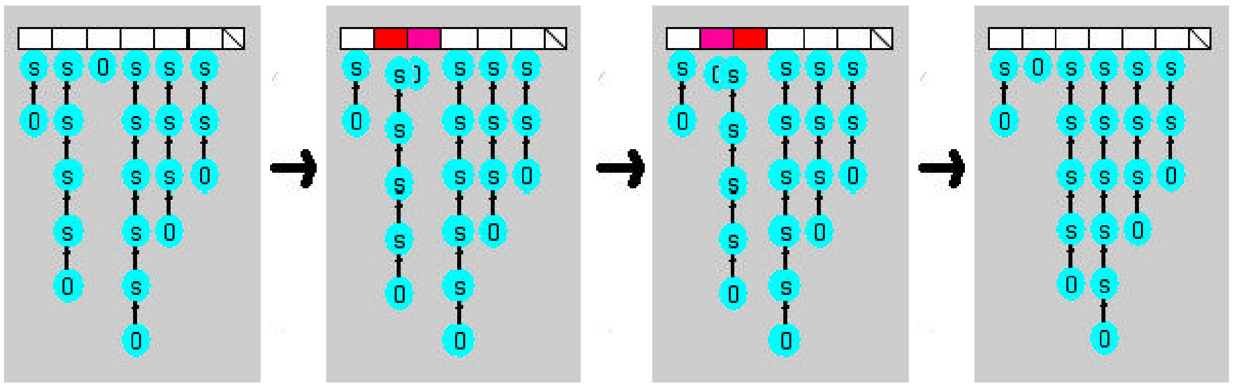

Figure 13. The smoother animation of the program execution

4. IMPLEMENTATION

Through the program execution, a programmer observes that the behavior of the editing program is just as expected. After the observation, (s)he understands the meaning of the program, or confirms that the behavior of the program is the same as the expectation. If the result of the execution is not the same as his expectation, he may debug the program.

4.1. Multiple View

An operator can have a multiple view. If a view is customized many times, the view adds to the operator in its occasions.

For example, the animation rule in figure 12 is created as follows. The original representation of the animation rule is as shown in figure 10. For making the animation, we can copy the initial term twice. The initial term is "push(N, push(M, empty))." The copied two terms are added between the initial term and the written term. Then, we change the color of the "push" rectangle in the copied terms, and we change the locations of the attributes "N" and "M" on the "push." At last, we get the animation rule shown in figure 12.

After making the animation shown before, the operator "push," with the multiple view is as shown in figure 14. The user can change easily the "push" view by selecting a used-defined view inside figure 14.

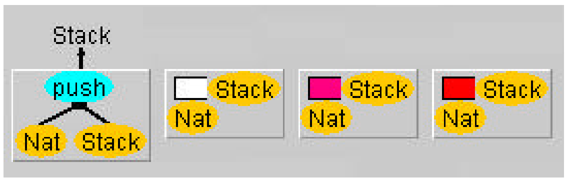

Figure 14. The multiple views of the operator "push."

4.2. Calculation of Layout

In the view customization, the following pre-defined visual components: an oval, a rectangle, a text and the user-defined image, are used. The component has properties such as a size, a location and a color. The view designer can modify the size or the location by direct manipulation on the components.

We describe the layout calculation in a term with the customized views. The calculation of each component is carried out in a bottom-up approach. For example, an operator size in a term is determined after each size of the inner components (the operator attributes) is determined.

The layout is decided by specifying relations between two components. A view designer can create a relation between two components by using drag-and-drop. The relation is determined as follows.

1. A part of one component is corresponding to another. For example, the location of two components is the same: the left end position of one component is the same as the right end position of another and so on.
2. If one component is inside another, the relation is the distance between the centers of the two components.
3. If one component is outside another, the relation is the shortest distance between two components.

The view customization changes only each relation between two components. If the view is changed, the system can calculate each location of components without confusion.

5. RELATED WORK

Various systems have been proposed through which users can watch and analyze the TRS. ReDux[10] 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[11] is a visual support environment for TRS. The system supports the environment for program execution, but does not support program editing visually. CafePie visually supports not only program execution but also program editing. CafePie is the first system that shows dynamically TRS execution with user-customized views. Viry[12] presents some preliminary ideas towards a user interface for completion and its integration within programming environments.

In the field of visualization of the data structure, algorithm animation is useful. Algorithm animation is a technique for showing a change in the data structure effectively. Algorithm animation system, such as Pavane[13] and Zeus[14], can specify a detailed animation toward one algorithm. Algorithm animation systems can visualize an execution of a complete program. The methods for visualizing a program are various according to programs.

GELO[15] is a system where users can customize the visualization of the data structure. Our view customization mechanism can be useful not only in the data structures but also in the definition of the c ontrol-structures. Visulan[16] is a visual programming Language based on bit-map rewriting. Bit-map as a program expression is described in the order set of the pattern replacement rule. Changing only a program view is difficult because bitmap itself shows a program model.

6. CONCLUSION

We have proposed a method of view customization in a visual programming system. The user can change the notation of the program representations by using our method. We describe methods for emphatic display with the views: the highlighted representation of the data structures and the highlighted animation with the control-structure definitions. The users can emphasize the data-structures and the control-structures definitions of the programs by using the customized views. We also describe the view customization mechanism in our visual programming system "CafePie."

REFERENCES

1. B.A. Myers, "Taxonomies of Visual Programming and Programming Visualization," Journal of Visual Languages and Computing, **1(1)**, pp. 97-123, 1990.
2. E. Glinert and S. Tanimoto, "PICT: An Interactive Graphical Programming Environment," *IEEE Computer*, **17(11)**, pp. 7-25, 1984.
3. M. Hirakawa, M. Tanaka, and T. Ichikawa, "An Iconic Programming System, HI-VISUAL," *IEEE Transaction on Software Engineering*, **16(10),** pp. 1178-1184, 1990.
4. J. Tanaka, "PP: Visual Programming System For Parallel Logic Programming Language GHC," *Proc. PDCN* **1997**, pp. 188-193, Singapore, August 1997.
5. T. Ogawa and J. Tanaka, "Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ," *Proc. ISFST* **1998**, pp. 155-160, October 1998.
6. T. Ogawa and J. Tanaka, "CafePie: A Visual Programming System for CafeOBJ," *CAFE: An Industrial-Strength Algebraic Formal Method*, Elsevier Science, pp. 145-160, 2000.
7. T. Ogawa and J. Tanaka, "Drag and Drop based Visual Programming System," *IPSJ Transactions on Programming*, **43 SIG1(PRO13)**, pp. 36-47, 2002 (in Japanese).
8. R. Diaconescu and K. Futatsugi, *CafeOBJ Report*, World Scientific, 1998.
9. K. J. Schmucker, *Object-Oriented Programming for the Macintosh (TM)*, Productivity Products International, Inc. 1986.
10. R. Bundgen, "Reduce the Redex -> ReDuX," *Rewriting Techniques and Applications*, **LNCS 690**, pp. 446-450, Springer, 1993.
11. N. Kawaguchi, T. Sakabe and Y. Inagaki, "TERSE: A Visual Environment for Supporting Analysis, Verification and Transformation of Term Rewriting Systems," *Proc. AMAST* **1996**, **LNCS 1101**, pages 571-574, 1996.
12. P. Viry, "A user-interface for Knuth-Bendix completion," *Proc. UITP* **1998**, July 1998.
13. K. C. Cox and G.-C. Roman, "Visualizing Concurrent Computations," *Proc. VL* **1991**, pp. 18-24, 1991.
14. M. H. Brown, "Zeus: A System for Algorithm Animation and Multi-View Editing," *Proc. VL* **1991**, pp. 4-9, October 1991.
15. S. P. Reiss, S. Meyers and C. Duby, "Using GELO to Visualize Software Systems," *Proc. UIST* **1989**, Williamsburg, VA, pp. 149-157, 1989.
16. K. Yamamoto, "Visulan: A Visual programming Language for Self-Changing Bitmap," *Proc. of International Conference on Visual Information Systems*, pp. 88-96, Melbourne, 1996.