A Visual Programming Environment
for Programming by Example Abstraction

Yoshiaki MIMA
IBM Research, Tokyo Research Laboratory
5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan
e-mail: mima@trivm.vnet.ibm.com

Abstract
This paper describes a way of extending the flexibility of example-based programming. The purpose of introducing such a programming style is to provide the user with a way of telling the system why and how he operates on it. This ability is achieved by introducing three mechanisms for target object specification, goal-driven execution, and parameter transfer by means of a clipboard. In the system, the user can specify the relationship between the application status, the target object, and the user operation. A flexible programming capability can be provided as a monitoring function for system support. Consequently, the user can apply the programming-by-example facility to any applications implemented on top of this system.

1 Introduction
In an example-based visual programming system, a user's operations are automatically recorded and replayed later. (Such a programming system is sometimes called a macro programming system.) This makes programming so easy that even a novice user can make a program simply by learning the record and replay instructions. Current macro programming, however, suffers from inflexibility. Although the user normally refers to the status of the executing program before an operation, it is difficult for such programming systems to monitor the program status in the replay mode. The difficulty of specifying the target object is also a cause of inflexibility. When the target object is different from the object specified at the time of recording, there is no general method for specifying objects.

The present paper describes how to create more flexible descriptions of user's operations by abstracting their meanings. This is done by recording a user's operations in order to specify relevant data. The data specification description is designed as a system operation so that an application programmer does not have to pay much attention to abstract recording. We implemented a programming environment called LEDA on top of the Presentation Manager on an IBM PS/2 computer using OS/2.

In Section 2, we review the methodology of "example-based programming" and clarify the problems involved. In Section 3, we give a model of user interaction, and propose a solution based on analysis of the model. In Section 4, we describe the design of a new environment for an application-independent abstraction of programming by example. Section 5 gives examples of programming on this system.

2 Related Work
One of the earliest macro programming systems for a visual environment was Pygmarion [1], which introduced the basic idea of example-based programming. In example-based programming systems, generalization of examples has been an important issue. Both Peridot [2], designed for user interface programming, and MetaMouse [3], designed for graphic drawing, use rule-based inference systems to guess the meaning of the user's operations. These
systems show that inferencing is helpful when the applications are restricted to a specific area.

To establish a programming method that is applicable to a general application area, another approach is required. SmallStar [4] used the so-called "data description" mechanism for program abstraction. Data description provides a way of specifying the target of operation. In SmallStar, the target data is selected by specifying the object characteristics. Data description enables users to generalize the recording. Since the predicates used in data description are fixed for each application, data description in SmallStar is application-dependent. Therefore, after designing each application, the application developer (programmer) should be responsible for the data description semantics of his application. SmallStar introduced not only the concept of data description but also a way of describing an operation on multiple objects. (We will discuss this later.) Mike [5] has a unique macro editing capability. As regards the generalization of a recorded control structure, Pygmarion, SmallStar, and Mike have similar strategies. These non-inference systems maintain the user’s operations as a simple sequence. To describe conditional branches, loops, or object selection, the user edits the recorded operations later. LIVE [6] has introduced a different way of describing the example execution with a Prolog-like description. When a program is executed, LIVE takes one of the possible execution scenarios that are already recorded. On executing each scenario, LIVE checks whether the current status is acceptable. If not, the execution is backtracked. Since LIVE is basically a mixture of a textual program and a visual program, when the target specification is complicated, the user specifies the target object by textual programming.

The above systems are experimental ones with advanced features for macro programming. There are also several commercially available products. TempoI [7] and other several utilities on Macintosh can record system-common events in low-level layers on a window system. Since these utilities handle system-common events, macro recording is applicable for all applications. Unfortunately, as the macro recorder and replayer know little about the status of the application, the execution is sensitive to the environment. (For example, the replayer may make an error in object selection when the target object has been moved. Garnet [8] showed a successful direction for the "abstraction of interaction," but his system is general-purpose and is not specifically concerned with macro recording.) In most available window systems, the semantic level of the input protocol is too low to replay stably. The important idea of using a clipboard as a variable was introduced by TempoI. TempoI used the clipboard to evaluate the current status of a conditional branch. Excel [9] is a spread-sheet application. The macro language in Excel supports the semantics of the application. For a really functional program, however, a recorded operation requires control structure descriptions, and the user needs additional descriptions in textual language.

3 Design Principle

Our goal is to automate the sequence of the user's operations and thus to reduce the amount of his routine work. Hereafter, we refer to this type of automation as "programming by example." In this section, we will provide a model of user interaction and a way of automating the user interaction.

3.1 User Interaction Model

In an interactive operation, an end user keeps his final goal or sub-goals in mind. After defining the goal, he looks at the status of the application through displayed graphic symbols and reacts accordingly. He repeats the target selection and the command execution alternately, according to the current status of relevant data. To check the value of the relevant data, the user does not directly examine the contents of the memory (where the information really is), Instead, he has a mental model of the behavior of the application program (Figure 1), and obtains the required information through a graphical user interface (GUI). The user examines some of the symbols on the screen (buttons, texts, application-specific symbols such as cells on a spread sheet, and so on) to decide the next action, and then selects the target symbol and a menu for the command execution. Target selection

3 TempoI is a Trademark of Affinity Microsystems, Ltd.

4 Macintosh is a Trademark of Apple Corp.
is also performed with reference to current graphic objects.

We classified what happens in the user's mind during the operation by describing three categories of relation between the user's decision and his reference to the application status.

**Category 1:** The user makes a decision for a reason independent of the status displayed. (There is a direct user input on each replay.) For decisions in this category, macro programming is easy. As the information is completely independent of the system and application program status, the user is required to provide information on each macro program execution. (For example, a calculator program elicits input by prompting.)

**Category 2:** The user's decision does not change whatever the status of the application program. (The command is always replayed as it is.) For decisions in the second category, straightforward example-based programming is possible, if the target object or executed command is successfully identified. Abstraction of the target specification, however, is required for stable macro programming.

**Category 3:** The user makes a decision after referring to the status of the application program as shown on the display. (Replay depends on the current status of the application program.) For such decisions, programming is more difficult, especially in conventional macro programming. The user obtains the status through the display, but the macro recorder has no way of knowing which status the user has referred to.

Incidentally, in example-based programming, the user has to tell the macro recorder how he made a decision on the operation if there is no inference mechanism. Therefore, the system is required to provide features for recording the user's decision process.

### 3.2 Abstraction and Graphical Search

**Abstraction:** In most window systems, a user's operations are passed to the application program as a change in the position and button status of the pointing device. The interpretation of the meaning of a user's operation is done inside the application program. Such a conventional structure makes it difficult to understand the meaning of the operation from outside the application program. In LEDA, after abstraction, the user's operation is passed to the application program and macro recorder as selection of an object, change of graphic attributes, or move of an object.

**Graphic search:** To provide more flexibility for object selection, we enhance the data description used in SmallStar. The user has a semantic model of the application in his mind. The target object in his mind has a corresponding graphical shape. Simultaneously, the application converts its data into a graphic object by the application. The user interacts with the application by operating on graphic objects. On recording an object selection procedure, a generic data description is extracted as a sequence of graphical search commands. The "Graphical Search and Replace" [10] technique is applied to our abstract presentation space for data description. The graphical search is based on the geometric attributes on the graphic presentation space. (For example, it is possible to pick the nearest red object to the pointer.) By recording the sequence of graphical search commands, the user can tell the macro recorder how the target object is specified.

### 3.3 Control Structure and Program Execution

We propose a model of a program that consists of a set of atomic operations between a human and a computer. An atomic operation is defined as a
triplet of \( \langle \text{goal, status evaluation, and execution} \rangle \). These atomic operations are kept in the atomic operation dictionary. When a program is executed, a goal is set to the "Goal Stack" by a user or running program. The Goal Stack is a system stack for keeping goals and the push/pop operation is defined on it. The goal at the top of the Goal Stack is called the current system goal. The program is executed as follows:

**Step 1:** First, the system searches the atomic operation dictionary for atomic operations whose goal matches the current system goal.

**Step 2:** If the system finds an atomic operation, it evaluates the current environment.

**Step 3:** If the result of evaluation is satisfactory, the execution step(s) will be executed. Otherwise, the system starts searching for another atomic operation.

Program execution is a sequence of selecting and executing operations. The current system goal resembles the contents of an instruction pointer. Changing the content of the current system goal works like a GOTO sentence, a subroutine call, or a message passing operation.

### 4 Implementation

From the functional point of view, **LEDA** is an externally controllable WYSIWYG-style graphic editor with a graphical search function and an operation macro recording facility. A set of application programming interfaces (APIs) is provided as a dynamic link library on OS/2. Through these APIs, the application program on **LEDA** manages graphic objects. **LEDA** consists of three components (Figure 2): The Object Manager maintains data of graphic objects as tree-structured information. It also supports the updating of graphic objects, graphical search, and recording operations. The Interactive Manager converts low-level events to high-level events. It supports a user's interactive operations on graphic objects. The System Menu Manager provides application-independent system function menus. Operations and object statuses are passed through the interactive manager, and the results are registered with the object manager. For application programs, messages about the manipulation of graphic objects are the only input messages from **LEDA**.

#### 4.1 Interaction Status

To reduce the communication overhead, real-time reactions such as rubber banding are managed by **LEDA**. Simply, the result of each operation is sent to the owner's input queue. An object has an attribute that specifies the reaction to a user operation. The attribute value is one of the four states: **Editable**, Movable, Selectable, or Non_sensitive. Non_sensitive objects cannot be manipulated by direct operation, and are used for display purposes.

### 4.2 Graphical Search for Object Selection

An object is searched for with its graphic attributes. The graphic attributes recognized by the user and application programs (object type, color, location, overlapping order, grouping relations, and so on.) are used to define the search pattern. A table of objects and their attributes is kept in the
system. A predicate can be described for a combination of grouped objects. For example, to search for a so-called "OK button," we can describe it as an object that "is grouped," "includes the string 'OK' as a member," and "includes a selectable round rectangle as a member."

When too many objects are matched in a search operation, there are two ways to reduce the number of target objects. One is to perform an additional selection operation; the other is to extract it with a sequencing number. Selected objects are ordered by sorting with their attributes, and the user can extract objects from the list by specifying the sequence number. (For example, it is possible to find the largest of a group of selected rectangles.)

4.3 Clipboard Stack as Working Space

A clipboard is used as a place to show, keep, and exchange values by using cut and paste operations. In this system the clipboard is extended as a pushdown stack. Stack operations such as "duplicate top," "exchange order," and "rotation" are provided. The clipboard can be used to pass parameters between macro programs. (This programming style is adopted in the PostScript Language.) The functions for converting or calculating graphic objects can be applied to the contents of the clipboard. An attribute value required for the condition check (object type, color, and so on) is extracted from the target objects, using the converting functions. The extracted value is converted to a real number, integer, string, or boolean. These operations are done visually on the clipboard.

Cut and paste operations, conversion and evaluation of values, condition check, and other operations for program generalization are recorded in the same way as normal operations.

4.4 Goal Stack and Control Structure

As mentioned in Section 3, the system refers to the goal stack to find the next atomic operation. At the beginning of an atomic operation, this system checks the current status by selecting a graphical object. Values for condition checking are specified in the system menu, and are kept on the clipboard for evaluation.

Any control structure can be realized. A conditional branch operation is realized by combining two atomic operations that have same goal descriptions and different conditional specifications. To describe iterations such as 'for' or "while," two atomic operations are used. When the loop exit condition is not satisfied, the system goal is kept as it is. Otherwise (when the loop exit condition is satisfied), the system goal is updated. Even a subroutine call can be realized by using the system stack. Before the subroutine call, one more goal is pushed onto the system stack. The goal of the next subroutine is then pushed onto the stack and the corresponding atomic operation is executed. At the end of the atomic execution, the current goal on the system stack is popped.

5 Sample Programs

In this section, we will give two sample programs. The first adds numbers by using the proposed system, and shows how the user can make a program. The second is a function for calculating...
factorials, and shows how to use sub-programs. A "practical use" of macro programming such as copying a specific document with reference to displayed information likely to be easier than these examples.

5.1 Summing up a value

This program adds the values of all red strings on the display. The result of the calculation is returned as a string on the stack.

Atomic Operation: Summation of Red Strings
Goalname: "SumRedStrings"
Check Status: TRUE
1. Push the value 0 onto the clipboard stack.
2. Select all the red strings.
3. Select the "get an element" menu.
4. Copy(push) the selected object to the clipboard stack.
5. Add the first two strings on the stack as integer values.
6. Pop the current goal from the goal stack.

5.2 Factorial

The input value is provided as a string on top of the clipboard stack. The result is returned to the top of the stack. At the beginning, this program generates red strings with the values 1 to n. It then selects all these objects and multiplies them all. It is possible to define the program as a recursive call using goal stack operation; however, we think the following solution is natural in this system.

Atomic Operation: GenNumber(1 of 2)
Goalname: "GenNumber"
Check Status: whether the top of the clipboard is 0.
1. Pop the clipboard.
2. Create the string "1" on the screen.
3. Pop the current goal from the goal stack.

Atomic Operation: GenNumber(2 of 2)
Goalname: "GenNumber"
Check Status: whether the top of the clipboard is not 0.
1. Change the current color to red.
2. Copy a string from the clipboard and paste it to the screen (as a red object).
3. Create the string "1" and push it to the clipboard.
4. Apply subtract operation to the first two strings on the clipboard.

Atomic Operation: Multiplication of Red Strings
(Almost the same as "Sum of Red Strings")

Atomic Operation: factorial
Goalname: "factorial"
Check Status: TRUE
1. Push the goal "fact_end" to goal stack.
2. Call GenNumber.

6 Concluding Remarks and Future Work

LEDA provides a supporting environment for example-based programming. In this environment, the user tells the system how he selects or changes target object(s) and how he decides on the operations. We are now creating applications on LEDA for evaluation. Our next step will be to extend LEDA. One possible way of doing so is to visualize the recorded programs.

7 References