Dynamic Interface for Machine Vision Systems*

Domenico Tegolo†, Biagio Lenzitti†, Francesco Isgrò†, Vito Di Gesù‡

†Dipartimento di Matematica ed Applicazioni, Università degli studi di Palermo
‡ T.E.M.A. S.r.l., Palermo
† Department of Computer Science, University of Washington, Seattle, U.S.A.

Abstract

Iconic programming intends to provide expressive tools to implement, to debug, and to execute programs. For this purpose, visual languages need pictorial constructs and metaphors to guide the design of algorithms in interactive fashion. In the paper a new class of dynamic visual interfaces, named DIVA (Dynamic Interface for Visual Applications), is introduced, its properties are described, and an application to visual compilers in a multi-processors system dedicated to image analysis is given. Moreover, a formal definition of dynamic icon (DI) is also given.

Keywords: Pictorial languages, visual languages, icon interfaces, visual compilers, parallel processing, machine vision.

1: Introduction

Humans interact with the real environment through their senses, they react and make decisions depending on the result of such interaction. This simple consideration has suggested most of the interactive computer systems based on virtual reality [1] and multi-media [2]. Vision play a relevant role among human senses, many efforts have been done in the last decade to improve the design of visual interfaces for computer systems.

Multi-processors systems are characterized by an huge amount of states and parameters, that are distributed through several elementary sub-system units (or processing units, PUs). Moreover functional dependencies may be defined between state and parameters. In this case, user/machine interfaces must provide a synthetic view of the system behaviour, to guide and to understand different phases of the computation. Visual languages (VL) should allow to express and visualize naturally and efficiently the dynamic evolution of distributed programs [7,8,9].

This paper introduces the concept of dynamic icon, as useful bricks to extend visual interfaces in multi-processors systems. In particular, an application to visual compilers will be considered. At the present, the system DIVA (Dynamic Interface for Visual Applications) is under development; it will constitute the kernel of the new operating system, and the visual interface of the pictorial language I_PICL [3] for the multi-processors machine HERMIA [4]. In the following, the model of computation considered is MIMD, and the processing units (PUs) are connected in a reconfigurable fashion. Each PU has local memory and control unit.

2: Dynamic icons

Informally, Dynamic Icons (DI's) are object-variables, values of which are assigned during the evolution of the system.

In order to define more formally DI's, it is necessary to introduce some notations and definitions. In the following, DI, P, and M denote the set of DI, the set of processes (P), and the set of metaphors (m) respectively. Here, the term metaphor has a perceptive meaning and it represent a visual pattern as well an acoustic signal or a combination of both. Three functions are also introduced to describe the evolution of an Icons-Processes-Metaphors (IPM) system: a) \( \mathcal{M}_1: P \leftrightarrow D \) (process-icon function); b) \( \mathcal{M}_2: \mathcal{M} \rightarrow P \) (metaphor-process function); c) \( \mathcal{M}_3: P \times \mathcal{M} \leftrightarrow DI \) (process, metaphor-icon function).

\( \mathcal{M}_1 \) is assumed one to one in order to avoid ambiguities; i.e. each P must corresponds to one and only one DI. \( \mathcal{M}_2 \) allows to assign a sub-set of metaphors to a given P; this property is necessary in order to handle the evolution of a DI. \( \mathcal{M}_3 \) is responsible for the assignment of a DI; in fact a pair \((P, m)\) determines the current value of an icon DI. \( \mathcal{M}_3 \) is also one to one. However, the same process could be represented by a different metaphor, depending on the evolution of the IPM system. Figure 1 shows the relational diagram of the functions above introduced.

In the case of multi-processors systems a single process P may belong to more than one PUs, moreover, during the computation, the evolution of the process

*Work supported by the Italian National Council of Research under contract N 92.015.79.69

1051-4651/94 $04.00 © 1994 IEEE
P, will be defined as a sequence of virtual processes, \( P \), to which correspond DI's, represented by time varying metaphors, \( m(t) \). In the following the time will be assumed discrete and finite; therefore a process, \( P \), will be defined as a sequence of virtual processes \( P(1), P(2), \ldots, P(N) \) and the corresponding DI will be also a sequence \( (DI(1), DI(2), \ldots, DI(0), DI(N)) \), where:

\[
DI(i) = m_3(P(i), m(i)) \quad \text{for } i = 1, 2, \ldots, N
\]

A set of relations, \( R \), may be defined on the space \( DI \) in order to visualize interactions between processes. For example, if the purpose is to visualize a multi-processors system, the computation of a processor may depends on the results received from other processors, i.e. it will fire as soon as its input ports receive the required signals. Machines, based on information fusion, give examples of such kind of computation [5]. In the next section, we are describing a dynamic icons application to visual compilers.

For this purpose ports, \( \text{INP}, \text{OUT} \), have been assigned to each DI. It follows that a given DI is defined by the quintuple \( DI = \langle P, m, \text{INP}, \text{OUT}, R \rangle \). Note that INP and OUT are also icons that describe relations, moreover INP and OUT can be empty. For example \( P \) could represent processes, data, and models necessary to activate the process connected to DI; while OUT could represent the result of the computation, the update of a model. Such feature is relevant in order to model computation based on information fusion techniques [6].

It must be pointed out, that a DI can be a collection of dynamic sub-icons connected through their INP/OUT ports, as a matter of fact that a process \( P \) can be a collection of interacting sub-processes. A DI, that includes dynamic sub-icons, is named compound, on the contrary is named primitive. Therefore a compound DI can be also represented by the pair \( \langle \{DI_{k}\}, R \rangle \).

3: Visual compilers

At the present, the compilation phase of \( VL \)'s is still performed on a standard source program language, this cause a conceptual gap between the visual program representation, defined on a set of video-pixels, in which the information is structured in a spatial logic, and the underlying processes. This determines a loss of both semantic understanding, and compilation time efficiency.

On the other hand, visual languages are represented by images, that can be modelled by direct graphs, \( G \), nodes of which are DI's and the arcs are relations determined by the set INP/OUT. The recursive definition of DI's allows to draw complex visual programs (i.e. each DI can represents a visual procedure itself). Visual compilers (\( VC \)'s) test the syntactic correctness of such graphs, by performing the parsing phase by using graph-grammars [12]. The visual parsing can be recursively applied to compound DI, until a primitive DI is reached; only at this point standard compilation will start. A graph representing visual programs, must satisfy the following properties, that are used during the compilation phase.

P1. The undirected graph of \( G \) (obtained by considering the edges undirected) must be connected, i.e. it must represents a single visual program.

P2. Each \( G \) has a source node \( I \) (representing the input), and a well node \( O \) (representing the output).

P3. Each sub-graph \( G' \) of \( G \) has a source node \( I' \) (representing an input information), and a well node \( O' \) (representing an output information) at least.

P4. A direct path must exist connecting \( I \) (I') and \( O \) (O').

Proper sub-graphs of \( G \) must satisfy P1, P3 and P4. Visual compilers may detect some syntactic errors during the editing of visual programs, for example the proper direction of arcs can be easily tested; during the editing some semantic inconsistencies can be also discovered, for example is not allowed to input/output processes, data and model that do not mach correct prototypes. The whole syntactic correctness of the visual program is then tested during the parsing phase of the VC. The parsing consists on a match-merge procedure applied to the graph, \( G \), that represents the visual program. The parsing is successfully if a single super-node is obtained, at the end of phase. The consistency rule depends on the graph-grammar, and from how it has been defined in the construction of the visual program. A simple example of production rules are showing in figure 1b. These rules are recursive, the contents of each DI depends on the functions defined in Section 2. For example, the graph \( DI \rightarrow DI \rightarrow DI \rightarrow DI \) will gives an error during the parsing phase.

Figure 1. a) Functions describing the relations between the sets DI, P, and M; b) an example of production rules.

A dependency tree (DT) is built at the end of the graphic parsing; on it each node contains: the reference to a DI, the corresponding process, and the local variables; if DI is compound the corresponding node is not leave
and it contains also the references to its sons; if the node is not root, then it includes the reference to its father.

In a multi-processors system the DT provides information about the dependence (independence) among processes. In fact two processes, P1 and P2, can be simultaneously active if a) the corresponding DI-nodes have the same father and a direct path joining them does not exist, or b) they belong to two disconnected sub-tree (see Figures 2a,b).

The introduction of DI's allows to edit visual programs with graphics tools; moreover the design of visual algorithms can be performed pictorially, the processes and the graphic metaphors of each DI are determined by the functions \( m_i \) (i=1,3).

The tree structure of the VC fits with the recursive definition of DI; this feature allows to build dynamic visual interfaces isomorphous to the compilation process. In this way the user can follow visually the compilation phase and its attention can be focused where errors occur. Program debug is also easily implemented, in fact the tree structure, built during the compilation, allows to evidence the active DI. This feature is also useful to detect run time errors.

4: General design and implementation

The system DIVA has been designed in order to test the usefulness and flexibility of dynamic icons. In particular, here the visual language environment (VLE) of DIVA is described. At the moment, two kind of dynamic variation have been considered: the local variation in order to visualize the current status of the process connected to a given DI, and the global variation to visualize the relative changing of the status of interacting processes.

The graphic description intend to provide the meaning of the procedure associated to the icon; the linguistic description is optional and indicates the procedure's name; the input/output channels provide the information flow among DI's; the semantic description contains information about the DI (history, purpose, input data, results, numerical limits,...).

Examples of primitives icons are: 

- \( \text{start icon} \) and \( \text{end icon} \) (start icon), \( \text{if-then-else icon} \) (if-then-else icon).

The editing of visual programs is performed by using mouse tools inside the working space. The user selects virtual-copies of primitive-icons from the context-space, and links them in a graph topology, according to the visual-program. Moreover, user defines the input/output devices (monitor, files, printer,...). In the following the basic visual commands are described:

- **open window.** To open an icon of a procedure window and to move it within of the procedure context;
- **create new icon procedure.** To create a new procedure icon by using a specific tools in which an user can define graphical and semantic descriptors, the number of input and of the output flow channels, and write the instructions of the procedure. The new icon primitive will be an element of procedure context;
- **link icons.** To link primitive-icons in a directed graph, nodes of which are the procedure-icons;
- **run.** To execute the visual program.

**How to create a new visual algorithm.** The creation of the algorithms is fully visual-oriented, in fact the user can create a link between two procedure-icons A and B by selecting, via mouse, the output channel of A and the corresponding input channel of B.

The graphic editor allows to delete a procedure-icon from an algorithm. The deletion of a procedure-icon involves the deletion of all its links with other procedure-icons. Obviously, the existing links to input/output channels must be modified. The user may rename the linguistic and semantic description of the running area. This will free the running area when a new section of DIVA system will be activated.

The VLE is organized in two window sections: the **Working Space**, to edit, to execute, and to debug algorithms; the **Context Procedure**, contains the icons of language-primitives (instructions, data types, standard I/O), and a database of procedural windows related to a specific class of applications (see Figure 3).

DI's inside DIVA are, generally, composed by three structured elements: graphical-linguistic description (metaphor \( m \)), input/output-flow-channels (INP OUT ports), and the semantic descriptor.

![Figure 2. Example of independent processes (a) and an example of dependent processes (b).](image)

![Figure 3. An example of Working (a) and Procedure (b) space.](image)
How to create a new icon procedure. The creation of a new procedure icon is driven by a specific tool that help the user to build the procedure step by step. A set of "Dialogue Box" asks to the user the number of input and output flow channels, the graphical descriptor, and the string of the linguistic descriptor.

The user can build the new procedure icon by starting from other built-in icons and by using the graphic editor. If the process connected to the new DI does not exist, the user will write the procedure in a standard programming language, inside the working window connected to the new icon.

How to compile a visual program. The evolution of the compiled DI's is visualized by displaying the corresponding sub graphs of G. This is obtained by expanding the compound DI's in its components; DIVA system flashes the DI currently under compilation, if an error occurs the flashing becomes red. The compiling phase also provides tools to display the correspondence between processes, variables, and the processing units where they are distributed.

How to execute an algorithms. An algorithm, defined in the working space, can be executed after that the user marks the start_icon of the algorithm. Note that the starting icons can be placed in any part of the working space. It is possible to have more then one start_icon, in order to run more processes or more asynchronous sections of the same process.

How to debug an algorithms. The visual interface allows to follow dynamically the execution of the program, by flashing the icon connected with the current active process. If the icon is a compound DI the user may follows the program flow by opening icons until primitive icons are reached.

Implementation. The first prototype of the DIVA system is under development in C++ language inside the visual environment of the HERMIA machine. The choose of C++ has been motivated by the fact that DIVA is an object oriented system. Dynamic icons are implemented by means of a stack of classes, top of which represents the current values of the DI (process, graphic metaphor, INP, OUT). From this point of view, DI's are dynamic classes, stored in class-base. The class-base contains primitive classes (pre-defined in DIVA) and user defined classes. The low level management of the class-base (insertion, modification, deletion) is given by DIVA.

The compiler of the Icon-PICL has been developed by following a multi-stages strategy to allow a partial independence from the specific hardware. This can be obtained by using the capability to assign hardware-knowledge to each DI. A two-phases compiler implementation strategy has been chosen to partially solve the problem of generality, by relegating the hardware dependence to the lower phase. Each phase corresponds to a virtual machine, which is connected to the appropriate application.

5: Concluding remarks

In the paper a new type of dynamic icon is described. Its utilization for the design of visual interfaces seems to be adequate to describe dynamic processes as the compilation of visual languages, or the editing of visual programs are. A formal definition of dynamic icon is also given, it is a starting point for the development of a mathematical theory of DI's.

References