

Visual composition of concurrent applications

A.F. da Silva

*Universidade Nova de Lisboa, Grupo de Eng.
Sistémica, 2825 Monte de Caparica, Portugal*

ABSTRACT

This paper describes an object-oriented visual programming system. The system manages a dataflow framework for the composition of concurrent applications expressed as active visual objects. A multidimensional data structure is used to manage interface related operations. This structure supports the management of the dataflow architecture and the scheduling of applications. The dynamic properties of the graphical interface are enhanced by the specification of replacement behaviours associated with the active objects. Together with the dynamic properties of the underlying data structure, active objects simplify the composition of applications and allow for the easy reconfiguration of the user interface.

INTRODUCTION

Visual language systems have been the subject of intensive research in a large number of different fields. The term visual language has been used to refer to all systems that use graphics, whether the objects to be dealt with have or not an inherent visual representation. The classification of visual language systems in visual programming and program visualisation systems was proposed by Myers [1]. In visual programming systems there are images or pictorial objects that are associated to construct a program or logical interpretation. In program visualisation systems the program is specified in a conventional textual manner being the graphics used as a complementary

tool. The lack of a formal specification for visual languages, somehow equivalent to the BNF's formulation of textual languages, makes visual programming difficult since each graphical language requires a different environment and specific visual utilities. Moreover, low-level graphics programming contribute to the non-portability of visual programs. Several visual programming environments based on an all-visual approach have been developed for specific application domains. However, for general-purpose programming textual languages are generally incorporated in the visual system to circumvent the shortcomings of current visual programming languages. User interface toolkits define an application area where visual languages have been successfully applied. The widespread adoption of the object-oriented paradigm appear to have contributed decisively to enhance user interface design.

In the field of scientific visualization several systems have been proposed based on the dataflow model (Felger et al. [19]). These systems tend to restrict the composition of applications to the process of configuring a set of modules (filters) linked by pipelines for visualization purposes. This approach has several limitations. The modules do not support semantic interaction, since they have rigid built-in behaviour and interfaces. The transfer of information is based on the flow of raw data due to the absence of abstract data types. Finally, the composition of applications is static since run-time reconfigurations are not supported by the interface. Some recent proposals have been put forward to address these problems; e.g. de Mey et al. [18], Chen et al. [20]. These approaches tend to conceive the application architecture as a composition graph defined in an object-oriented environment.

In this paper, an object-oriented visual programming system following similar guidelines is introduced. Nevertheless, we focus on the problem of dynamic reconfiguration in the framework of a strongly-typed object-oriented language such as C++. Two mechanisms are introduced to support the dynamic properties of the interface: concurrency and replacement behaviour. The interface manages a dataflow environment for the composition of concurrent applications expressed as active visual objects. A quaternary tree data structure is used to support all the required operations for managing the

dataflow diagram layout and the scheduling of applications. Together with the dynamic properties of the underlying data structure, the active objects simplify the composition of applications and allow for the easy reconfiguration of the user interface.

MODEL SPECIFICATION

The basic motivation in developing the present environment has been the need to simplify software development and promote dynamic reconfiguration. For this purpose three main guidelines have been followed. First, the object-oriented approach has been used for the construction of a graphical interface for interactive composition of application programs [2,3]. Graphics are used to represent the high-level structure of the global application. Second, by using a coarse-grain dataflow language supporting modification of program graphs a systematic way of building programs is supported. To this aim a visual framework has been developed in which particular programs, written as sequential procedures, are perceived as modules associated with nodes of a flow graph. Third, a concurrent programming toolkit has been introduced for the development of multitasking applications. The toolkit facilities are intended to integrate well with existent C concurrent libraries by using standard C++ compilers and no language extensions. A set of C++ constructs has been devised to effectively enable the integration of the concurrent model in the object-oriented framework. The above considerations led to the development of a programming environment with the following additional features:

Visual programming

One important graphic tool used in many visual systems is the dataflow diagram [4,5]. In the dataflow approach the program is perceived as a directed graph in which nodes are computation primitives and arcs represent data dependencies between nodes. Typically, each node has a visual signature and encapsulates a function which receives input data and generates output data. An application is supposed to be written as an encapsulated program or module which contains data and the associated procedures to operate on the data. Particular applications are viewed as icons. A graphical editor enables the visual composition of applications in order to build up the global flow

diagram. After being individually tested and compiled modules can be integrated at run-time in the environment by interacting with the graphical editor. A C++ object-oriented approach in developing the visual user interface was adopted. Some additional tools support overlapped windows and class browsers [6].

Dynamic composition

Interactive node manipulation such as insertion, deletion, and relocation of nodes is enabled. A quaternary tree data structure in which the leaf nodes reference icons is used as the building block supporting dynamic operations. The dynamic redefinition of icon connected functions is largely supported.

Concurrency

Recently, concurrent object-oriented languages have been developed which support concurrency within objects [7,8]. Frameworks for parallel programming have been proposed as well [9,10]. Our objective is to set up an object-oriented framework to enable concurrent programming of applications. In our present model concurrency within objects is supported under restricted conditions: concurrency is limited to a single thread and only specific member-functions may generate active objects. Individual modules implement specific user defined applications. Each module expresses a sequential set of actions and is written as a procedure intended to be run as a task. The term task is used to denote the activation of a procedure for concurrent execution. The term is synonymous with lightweight process and thread of execution. These terms denote a task which runs in co-operation with other tasks in a single address space. Contrary to heavyweight processes used in multitasking operating systems (such as UNIX) lightweight processes have the advantages that they can be scheduled very rapidly and that they can share access to global variables within their common address space. A shared-memory model of computation has been used. Objects become active objects when some member-function may be run as a task. In order to ease integration with existent concurrent programming facilities objects must be declared active objects in order to run as tasks. Moreover, only those member functions associated with a user-defined task may be scheduled for execution. This corresponds to a coarse-grain model of computation for active objects.

ARCHITECTURE

Managing the visual composition of applications involves three main responsibilities: architecture management, behaviour management and task scheduling management. Next, the design options adopted for each one of these problems are considered.

Architecture Management

Dataflow diagrams have been used in many visual programming systems. Some systems allow the designer to specify the user interface graphically while others require textual representations. A problem common to many graphical representations is the generation of a suitable layout. An optimal path is one with the shortest possible length and having the fewest number of corners. Finding the optimal path for a connection between two icons can be very difficult. Another source of complexity deals with the number of input and output links for connecting icons. Trying to manage these problems in the most general way tend to yield unattractive visual designs and impair the system dynamic capabilities. Our design strategy tries to emphasize the dynamic features ruling the user interface. For this purpose, several restrictions were imposed. The layout locations are restricted to be coincident with the nodes of a two-dimensional grid. The icons after being interactively located in the screen by the user are automatically attracted to the nearest grid position. An occupied icon location may not be overridden unless the user explicitly activates the icon substitution command. On the other hand the links between icons are restricted to be horizontal or vertical lines. When dragging inter-icon links, repeated clicks of the mouse button toggles between upper-oriented routing and down-oriented routing in links requiring chained adjoining lines. Rubber-band lines visually inform the user of the routing that will be set up. The number of adjacent nodes in the graph are restricted to be four, one for each side of the rectangular icon box. Therefore, the number of one-way communicating lines per icon is also restricted to be at most four.

As regards the underlying data structure many visual systems rely on simple data structures to store icon information. Hash tables have also been used (SunPict for instance [11]). Multidimensional data structures have,

however, been widely used in computer graphics and image processing. Data structures such as the quadtree/octree have emerged as convenient structures for operating on multidimensional data [21]. Two features of these structures are particularly interesting: their hierarchic structure based on the principle of recursive decomposition and their efficient dynamic capabilities. As exemplified in Figure 1, a modified pseudo point quadtree structure [12,13] was implemented to store the 2D icon handle points. These points are represented as leaves in the quadtree but they reference quadrant subdivision as well. Figure 1a illustrates the assignment of points to quadrants, including those lying on a quadrant border. Points on a horizontal border are assigned to the upper quadrant. Points on a vertical border are assigned to the right quadrant. The implementation of the dynamic operations in this data structure closely follows the techniques described in [12]. The leaves of the quadtree structure do not store the icon objects themselves but hold pointers to arbitrary data types which can be customized by the application. The quadtree structure was defined as a generic class [17]. The member functions implement the required dynamic operations of node insertion, node deletion, node searching, etc.. The generalization of the class to objects of any type is implemented through macros [14]. An external user-defined function may be passed to the quadtree structure. This feature enables the dynamic (run-time) redefinition of behaviour associated with the quadtree nodes. In order to represent the dataflow diagram we have to define the icon objects themselves. The resulting data structure is the framework for all subsequent icon related operations. Most of these operations implement tree walking algorithm. The tree data structure may also be used by the task manager to schedule icon tasks.

Icons as Agents An icon object can be perceived as an agent able to provide some specific services. The agents exist in a network controlled by a manager. The network provides communication channels between agents. Figure 2 represents the overall structure linking the manager's structure to the network of co-operating agents. Icons are modelled as active objects. In [17] some of the responsibilities associated with the icon class were defined.

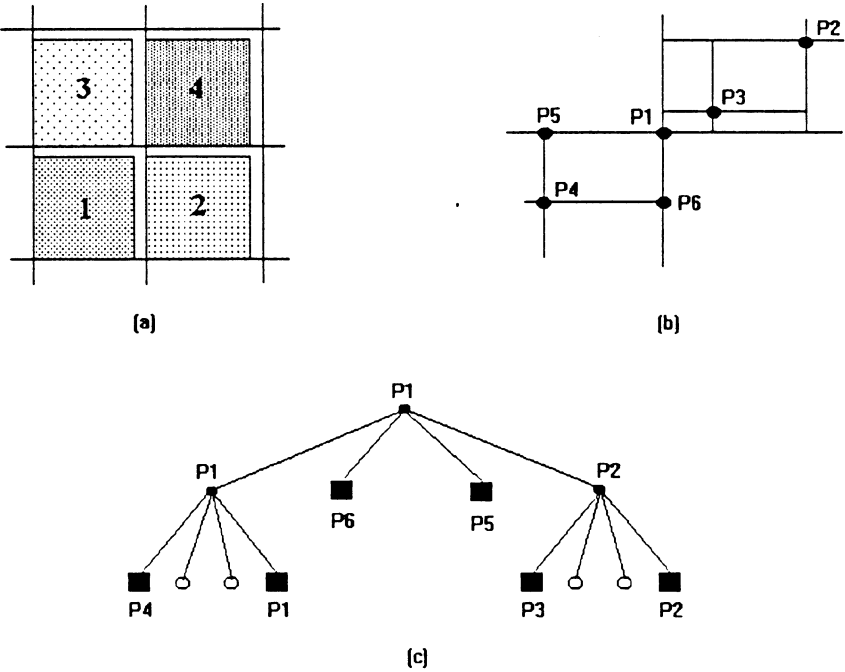


Figure 1. (a) Quadrant assignment. (b) Points to insert. (c) Data structure.

Each icon object has the possibility of managing an attached window object. The window object is an interactive tool at user disposal and a peep-hole into the flow diagram data and activities. It permits the visualization of icon-specific operations as well as the querying of icon related information. In particular, the user may inquire about active ports, verify the definitions of objects to be communicated, and browse through the icon function. On the other hand, derived window classes may easily be incorporated for proper customisation of user requirements and preferences in the context of the icon activities. The icon-window is valuable for monitoring dynamic phenomena as well. A set of member-functions provide support for attached window objects, namely creation, deletion, and suspension. The core of all icon activities is concentrated in the external function which is passed to the icon object. This function can be run as a task in accordance with the schedule set up by the task manager. Since the icon function can be substituted at run-time the dynamic capabilities of the environment are enhanced. Moreover, this function can be substituted by the global icon manager whenever necessary.



For instance, to perform some action common to all icons the icon manager may assign a common function to all icons and then walk down the tree requesting each icon services. A key design decision was to separate specification of the communication and synchronisation from specification of the units of computation. In this manner, the external function may be oblivious of the global setting and may therefore be tested independently. However, for intertask communication purposes each external function manipulates input and output objects which must be taken into account in a proper flow diagram setting. In order to separate communication from computation the icon as the possibility of enqueueing and dequeueing objects which are not directly interpreted by the icon but passed back to the function manipulating them.

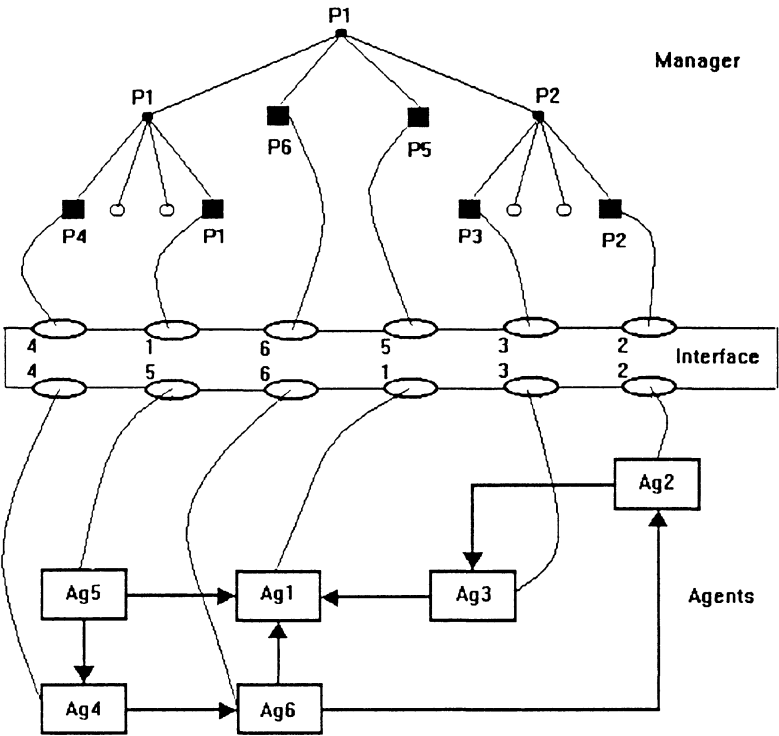


Figure 2. Structure linking the manager to the agents' network

Behaviour Management

In order to build a dataflow application a default table of icons is presented to the user. A diagram may be constructed by moving instances of those icons to specific screen locations and dragging lines between them. Initially each icon has a pre-defined function assigned to it. As we have seen, dynamic function management is a fundamental issue in our design approach. In keeping with the visual approach, the redefinition of the icon functions (or simply their substitution) must be supported by the interface. Redefining an icon function involves two aspects: the redefinition of the textual code associated with the icon function, and the redefinition of its visual (iconic) content, presentation and textual-to-visual correspondence. For dealing with these matters a class has been defined. The class manages operations on an array of pointers to functions. Operations such as add, remove and replace care for the maintenance of a set of basic functions. An external user defined function, for instance, may be added to the basic function set and thus be made available to the designer. The class deals with the textual to visual mapping as well. Each icon function as a unique visual signature which evokes its code contents. The function's visual signature must then be incorporated in the icon table to be presented to the user.

To deal with the manipulation of information as described above, the tools provided by the main interface for text and icon editing may be used. In most cases, however, the library of functions and icons required for composing the application is specified beforehand. Since there is separation at the interface level between communication and synchronisation on one hand and computation on the other, the functions may be externally coded and compiled. For icon functions having common scope, i.e. those functions which may be meaningfully applied to any other icon object no matter their location in the diagram, at least one icon location is reserved in the visual layout. This icon location (or locations) holds signatures of frequently replaced visiting functions having common scope. This location is used by the icon manager to perform common dynamic operations and updates. Actually, the objects of the class responsible for managing the set of functions are used by the icon manager to dynamize the interface.

Task Management

The concurrent capabilities of the interface relies on a concurrent programming toolkit. The toolkit is a C program library allowing the user to build programs as a set of co-operating lightweight processes, which are threads of execution that share a single address space within a single application program [15]. A variety of facilities, commonly available in equivalent systems referenced in the literature [16], have been implemented. In particular, there are facilities for creating and destroying tasks, for constructing intertask communication, for scheduling tasks and selecting the scheduling policy, and for handling exceptional conditions. A running task may be suspended or killed. Suspension occurs in three situations: when the task blocks awaiting the arrival a resource from another task, when the task is preempted by another task or interrupt handler, or when the task sleeps for the desired time interval. A suspended task may resume execution after the appropriate condition is satisfied. When a task is first created it enters the ready state. When a task is scheduled for execution, according to its priority and scheduling policy, it enters the running state. If a running task is preempted it returns to the ready state awaiting future scheduling. A sleeping task returns to the ready state when the specified time interval expires or when it is awakened by another task. Two task scheduling methods are supported: event-driven scheduling and real-time scheduling. In the former the system time automatically advances when all tasks are either blocked or sleeping, and the task scheduler exits when no task exists or can be awakened by advancing the time. This method is useful in computer simulations. In the latter, one can advance the system's clock using an interrupt source in order to synchronise it with the passage of time. This is useful in real-time applications.

Tasks communicate through shared memory. Several facilities for structuring intertask communications are provided. They allow tasks to cooperate by synchronising their actions and passing data to each other. The main facility used by the interface for intertask communication is the blocking queue (Figure 3). Each task automatically receives its own built-in blocking queue implemented as a doubly linked list. Objects can be passed between tasks by placing them into queues. A task can enqueue objects into

another task's blocking queue using the task's identifier. Objects are passed by reference. That is, the queue maintains references to the enqueued items. The blocking queue has the capability of synchronising the activities of multiple tasks. If a task attempts to dequeue an object from an empty blocking queue, the task suspends execution awaiting the arrival of an object. When another task places an object into an empty queue it unblocks the waiting task, which then dequeues the object after resuming execution. On the other hand, a single task may serve a blocking queue to which requests are submitted by other tasks. A time-out interval may be defined for tasks kept waiting in the blocking queue. To deal with active objects a set of C++ classes has been built on the above toolkit [17]. All objects transmitted between tasks are instances of the same Item class. They serve as standard carriers which reference the actual data objects containing the information to be communicated.

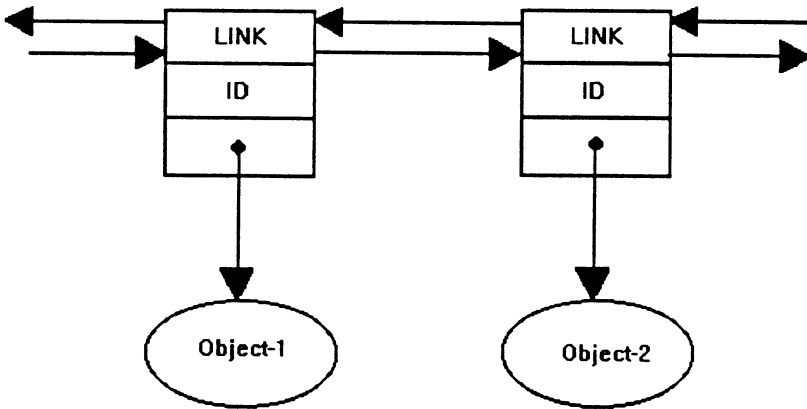


Figure 3. Queue link structure.

BUILDING APPLICATIONS

A dataflow diagram is built by dragging icons from the icon-table into discrete screen locations and linking the icons afterwards. Figure 4 shows two views of the graphical user interface. The Composition menu item references operations for layout management, namely insertion and deletion of icons and links, icon related queries, selective updating, and modification

of drawing options. These services are controlled by the manager. Agents request the manager's services when they need to perform some dynamic operation. For instance, an agent may request the manager to set up a new agent in the network and be connected to him. The manager's menu item enables the suspension of the network's operation in order to access the browse tools and windowing system described elsewhere [6]. Once the network is established the menu item 'Run_Task_System' may be selected. In this mode, the agents' services are scheduled according to their priority, and the flux of information follows the network's connections. Icon windows may be selectively open to visualize dynamic phenomena occurring as a result of the agents' activities. In debug mode the execution of the agents' processes is traced. In particular, information is generated to trace the scheduling of tasks.

CONCLUSION

An interactive object-oriented framework has been designed and implemented for the integration of programs specified as separate units and able to run as lightweight processes. By normalising the channels of communication between units and their synchronisation mechanisms, the separation between computation and communication has been greatly enhanced. This separation, in conjunction with the definition of abstract data types for function manipulation, contributed to the dynamization of the interface. For composing coarse-grain applications the system provides a visually clean interface supporting behavioural re-specification. Several improvements are currently under consideration. The icon's visual content may be enriched. The hierarchic nature of the data structure seems adequate to activate multi-layered diagrams. Real-time scheduling may be applied to discrete-event simulation and animation. These issues constitute directions for future research.

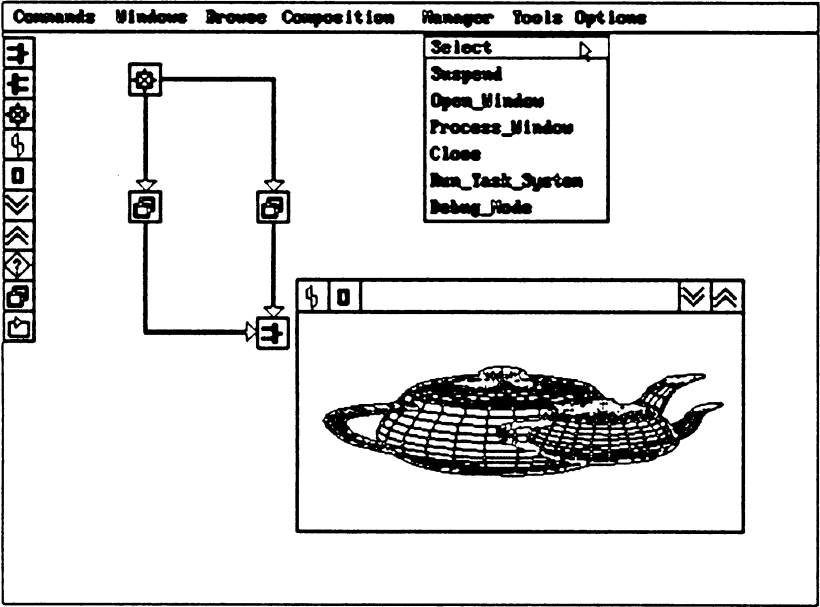
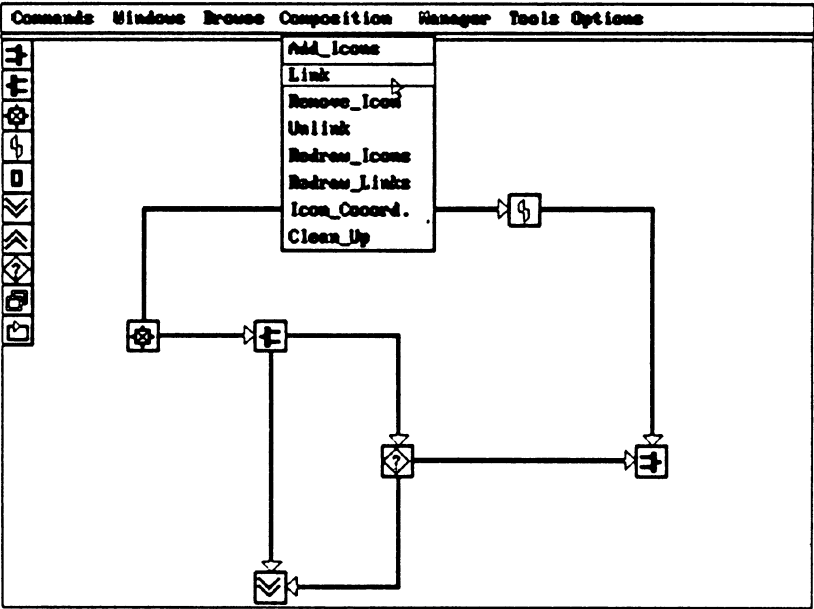


Figure 4. Aspects of the user interface.



REFERENCES

1. Myers, B. A. 'Taxonomies of Visual Programming and Program Visualization', *J. of Visual Lang. and Computing*, Vol.1, No.1, pp. 97-113, 1990.
2. Nierstrasz, O., Tschritzis, de Mey, Vicki, Stadelmann, M. 'Objects + Scripts = Applications', in *Object Composition*, Université de Genève, pp. 11-29, 1991.
3. Harbert, A. et al. 'A Graphical Specification System for User-Interface Design', *IEEE Software*, Vol.7, No.4, pp. 12-20, 1990.
3. Wirfs-Brock, R. and Johnson, R. E. 'Surveying Current Research in Object-Oriented Design', *Comm. of the ACM*, Vol.33, No. 9, pp. 104-124, Sept 1990.
4. Chang, Shi-Kuo (Ed). *Principles of Visual Programming Systems*, Prentice-Hall, 1990.
5. Tanimoto, S. L. 'VIVA: A Visual Language for Image Processing', *J. of Visual Lang. and Computing*, Vol.1, No.2, pp. 127-139, 1990,.
6. Silva, A. and Ramos, H. D. 'Building User-Interface Support for Object-Oriented Design', *Proc. of the First Int. Conf. on Comp. Graph. and Visualization Techniques, Compugraphics '91*, pp. 371-380, 1991.
7. Yonezawa A. *ABCL: An Object-Oriented Concurrent System*, The MIT Press, 1990.
8. Nierstrasz, O. 'The Next 700 Concurrent Object-Oriented Languages', in *Object Composition*, Université de Genève, pp. 165-187, 1991.
9. Browne, J. C. 'CODE: A Unified Approach to Parallel Programming', *IEEE Software*, Vol.7, No.4, pp. 10-19, 1989.
10. Suhler, P. A. 'TDFL: A Task-Level Dataflow Language', *J. of Parallel and Dist. Computing*, Vol.9, pp. 103-115, 1990.
11. Glinert, E. P., Kopache, M. E., and McIntyre, D. W. 'Exploring the General-Purpose Visual Alternative', *J. of Visual Lang. and Computing*, Vol.1, No.1, pp. 3-39, 1990.
12. Samet, H.. *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1990.
13. Overmars, M. *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science No.156, Springer-Verlag, 1982.
14. Ellis, M. and Stroustrup, B. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
15. Axford, T. *Concurrent Programming: Fundamental Techniques for Real-Time*

and Parallel Software Design, John Wiley & Sons Ltd., 1989.

16. Anderson, T. E., Lazowska, E. D. and Levy, H. M. 'The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors' *IEEE Trans. on Computers*, Vol.38, No.12, pp. 1631-1644, 1989.
17. Silva, A. 'Acting on Inheritance Hierarchies', *Proc. of the 3th Eurographics Workshop on Object-Oriented Graphics*, pp. 83-96, Oct. 1992.
18. de Mey, Vicki, Breitneder, C., Dami, L., Gibbs, S. and Tsichritzis, D. 'Visual Composition and Multimedia', *Computer Graphics Forum*, Vol.II, No.3, pp. C-9 - C-22, 1992.
19. Felger, W. and Shroder, F. 'The Visualization Input Pipeline - Enabling Semantic Interaction in Scientific Visualization', *Computer Graphics Forum*, Vol.II, No.3, pp. C-139 - C-149, 1992.
20. Chen, M., Townsend, P. and Wang, C. Y. 'A Development Environment for Constructing Graph-Based Editing Tools', *Computer Graphics Forum*, Vol.II, No.3, pp. C-345- C-355, 1992.
21. Silva, A. and Ramos, H. D. 'On Integrated Finite Element Modelling', *Proc. of the 6th Mediterranean Electrotechnical Conference, MELECON '91*, Ljubljana, Vol.II, pp. 965-968, 1991.