1991

An Object-Oriented Learning/Design Support Environment

Fillia Makedon

Julie C. Jumes

Jill P. David

AN OBJECT-ORIENTED LEARNING/DESIGN SUPPORT ENVIRONMENT

Fillia Makedon
Julie C. Jumes
Jill P. David

Technical Report PCS-TR91-165

An Object-Oriented Learning/Design Support Environment

OOD/OOP

An Object-Oriented Learning/Design Support Environment

1.0 Introduction

We present an object-oriented experimental learning and design support environment, called **AVT**, for an **Algorithm Visualization Tool**, implemented in Digitalk's ***Smalltalk/V*** [1] on a Macintosh II[2]. AVT provides a domain-independent visualization tool, an exploratory learning environment, and an experimental heuristic design environment. Algorithm visualization is the exploration of ways to visualize intuitively the computational behavior of an algorithm using multiple views, some of which are visual in the graphical sense [2,4]. AVT employs other views (combining text and graphics) to explain the problem, the strategy, the heuristics, and the reasoning process behind the solutions. User interaction in AVT includes not only passive viewing of the animated algorithmic process but also active participation in the design of the steps of the algorithm. Object-Oriented Programming (**OOP**) offers an attractive paradigm for rapidly implementing heuristics for our experimental environment and provides the adaptability for changing software requirements as well as more coherent and understandable code [1,12]. Inheritance properties of OOP languages capture natural mechanisms such as specialization, abstraction, and evolution allowing us to model our environment in a more natural manner [11].

1.1 Motivation

The initial implementation of AVT focuses on heuristics for VLSI (Very Large Scale Integration) layout design problems, an area of both economic and educational importance. Since intuition is the basis of heuristic design strategies, these strategies are not easy to teach or learn. AVT provides an exploratory approach to learning and design specifically to fill this gap left by other tools in industry and education, i.e., to explore heuristic design inside a learning environment [4].

---

[1] Smalltalk/V is a registered trademark of Digitalk, Incorporated.
[2] Macintosh is a trademark of Apple Computer, Incorporated.

Using algorithm visualization, AVT can be used as supplementary material for an algorithms course [2,4], a core course requirement in any computer science curriculum. An algorithms course teaches algorithm design methodology and introduces the concept of algorithm complexity with respect to optimal solutions. It also acquaints the student with a class of "very hard" problems (known as **NP-Complete** or **intractable** problems) for which no known efficient optimal algorithms exist. A standard teaching method for an algorithms course is to illustrate them by examples.

AVT can also be used as a learning and research tool by both computer science and electrical engineering students in the area of computer-aided chip design. Recent advances in VLSI technology make it possible to put more and more components on a chip. With the decrease in scale to smaller dimensions, the effects of on-chip wiring become more important with respect to speed of operation, processing costs, and reliability [9]. Hence, the problem of efficiently designing a chip automatically is very important. Unfortunately, almost all VLSI algorithms are inherently intractable; thus, heuristics are used to develop algorithms based on human intuition.

1.2 Problem Domain Implemented

VLSI Layout design involves the assignment of logic circuits to physical design modules, the placement of these design units onto larger function modules, and the determination of the routing patterns for interconnection [6]. Although these problems are closely related, they have been treated separately because of the inherent computational complexity of the entire layout design process. Layout is divided into a placement and a routing phase. Placement consists of the assignment of the logic modules to positions on the chip. The routing phase is usually divided into two stages: **global routing**, which assigns nets to channels, and **detailed routing**, which determines the final routing of the connections within the channels [3]. The primary goal in layout design is to minimize the total silicon area occupied by the chip. Since most of the chip silicon area is composed of the interconnections (wiring) between electronic components, the parameters to optimize in the

layout process are 100% completion of routing, minimum total wire length, and minimum chip area [7]. These parameters often conflict, i.e., minimizing one may increase another.

2.0 Overview of AVT

The purpose of AVT is to promote a conceptual understanding of algorithm behavior for both a novice and an expert algorithm designer. AVT uses algorithm visualization through multiple views to achieve this goal. These multiple views include: (1) an animation view, containing dynamic visual images of the algorithm as it executes; (2) a learning view, consisting of a context-sensitive help system in which the user explores background information of the problem at hand or retrieves explanations of the heuristics decisions made during execution; (3) a data structure view, providing the user with the ability to examine the values of data structures execution; and (4) an algorithm (or code) view, which displays the abstract high-level algorithm, accessible and modifiable by the user.

The abstract architecture of AVT consists of several domain-independent modules that supervise user interaction with the multiple views in the learning/design support environment. The modules include the Visualization Manager, the Execution Manager, the Algorithm Editor/Compiler, the Help System Manager, and the User-Interface Manager. Figure 1 illustrates the visualization window of AVT in which algorithm execution occurs. The **animation pane**, supervised by the Visualization Manager, shows the animated results of algorithm execution. Figure 1 illustrates a problem instance for the domain of detailed routing. The **result pane** displays the problem parameters selected by the user as well as the dynamically changing optimization parameters of interest, while the **algorithm pane** displays the abstract algorithm. The Execution Manager supervises algorithm execution and update of other dynamically changing problem parameters in the algorithm and result panes, respectively. The Execution Manager also handles the actions of the user-activated buttons **Help**, **Next**, **Explain**, and **Modify**, in the **control pane**.

The menu bar at the top of the screen (see Figure 1) contains the **Run**, **Algorithms**, and **Options** menus, provided by the User-Interface Manager for input of problem parameters. The Run Menu contains selections for running the same or a new example, for examining the data structures of the domain object, and for editing the selected algorithm. The Algorithms and Options menus are domain-dependent and vary according to the problem domain selected. The Options Menu has selections for various input problem parameters, while the Algorithms Menu lists selections for the algorithms currently implemented in the selected problem domain.

The user may activate the Algorithm Editor/Compiler through a selection on the Run Menu for creating, editing, and verifying the syntax of an algorithm. Some of the user-modifiable instance variables of an algorithm step object include **userName**, **msgName**, **args**, **trace**, and **context**. The "userName" is a mnemonic user-defined name for the selected algorithm step which appears in the algorithm pane view of visualization window. The "msgName" is the name of the method actually executed when AVT interprets this step, and "args" are the arguments for the method "msgName." The user sets the trace instance variable to true if he wants the decisions of this step recorded for use by the Help System Manager. The "context" variable optionally contains key word(s) used by the Help System Manager for activating context-sensitive help in a hypermedia help window.

The Help System Manager supervises the actions of the two control buttons Help and Explain. Figure 2 illustrates a typical hypermedia help window activated by pressing the Help button in the domain of global routing. Hypermedia is a popular approach to computer applications dealing with on-line presentation of large amounts of loosely structured information, such as documentation, instructions, or computer aided instruction (CAI) [8,13]. A user navigates through the hypermedia or hypertext information by selecting a topic from the **topic pane** or by clicking on embedded links (highlighted words) in the **text pane**. Hypermedia emphasizes user control by providing information to those who need it (the novice) without boring those who do not [4]. The user may

also activate another hypermedia help window for instructions on how to use AVT by a menu selection on the Run Menu. All hypermedia browsers include editing facilities for entering and modifying topics and their associated text and graphics images. The Help System Manager also handles the explanation facility which the user activates by pressing the Explain button in the visualization window. When activated, an explanation window pops up showing the sequence of heuristic decisions and their rationale made during algorithm execution.

3.0 OOP Design of AVT

3.1 Classes in AVT

The AVT environment for VLSI layout design problems consists of a set of methods (operations) and objects (data structures that contain its state). In AVT, we define objects and a set of messages for the following: (1) the visualization tool, (2) the hypermedia help facilities, (3) each problem domain, (4) the algorithmic language, and (5) the algorithms themselves. We have defined the following domain-independent classes in our experimental environment: **VizTool, HyperMediaBrowser, AlgorithmStep, Algorithm, AlgCompiler, AlgCommand**, and a class for each of the algorithm commands defined in AVT's algorithmic language. The domain-dependent classes **Chip**, **ChipGR**, **ChipDR**, and **Net** define some of the objects for implementing the routing problem domains [4]. Figure 3 displays the semantic hierarchy of AVT, a VizTool object, showing the major instance variables and their classes.

3.2 Benefits of OOP

AVT takes advantage of the OOP property of abstract classes. An **abstract class** is a class out of which no objects are ever created but which serves as a superclass to its subclasses [10,11]. An abstract class contains all the operations (methods) and variables which are common to all its subclasses. One example of an abstract class used in AVT is the domain-dependent class, **Chip**. Its subclasses are **ChipGR** for the global routing domain and **ChipDR** for the detailed routing domain. We define instance variable **cells** in abstract class Chip, but we define its representation in the

subclasses ChipGR and ChipDR, allowing us to experiment with different implementations for the same data structure. Figure 4 shows the class hierarchy of the routing domains. Another example of an abstract class is the domain-independent class **AlgCommand**. AlgCommand contains subclasses for each of the eight commands in AVT's algorithmic language. Methods for initialization or displaying text or error messages are common to all subclasses [4]. Thus, these methods are defined in AlgCommand. Since the parsing method for each command is slightly different, a **parse:with:** method is defined within each command subclass. Defining separate subclasses for each command further specializes an algorithm command and simplifies design of the compiler. Figure 5 shows the class hierarchy of AlgCommand and each of its subclasses.

4.0 Smalltalk Contributions

The Smalltalk programming language provides a variety of user-interface, window, and graphics tools for rapidly prototyping our learning/design support environment. Additionally, using consistent design principles in a completely object-oriented environment simplified the development of AVT. We discuss below several examples of how the use of Smalltalk contributed to the overall design of AVT.

4.1 Execution Manager

AVT starts execution of an algorithm by a user selection from the Run Menu. AVT knows nothing about the algorithm except that it consists of a sequence of steps which must be executed. After execution of the first algorithm step, AVT returns control to the user. The user may continue execution by pressing the **Next** button in the visualization window or activate one of the help utilities. If the user presses the Next button, then AVT executes its **next** method, which consists of an event loop in which AVT executes algorithm steps until a "waitEvent" (pause) or "halt" step occurs. At the "waitEvent," the Execution Manager again waits for user interaction. At a "halt" step, AVT exits the event loop and waits for user action. Placement of a "waitEvent" step within an algorithm is user-modifiable through the Algorithm Editor/Compiler.

AVT executes an algorithm step by sending the Algorithm object the message **performStep: aStep**. The method **performStep:** in class **Algorithm** interprets an algorithm step by determining from the argument "aStep" the receiver object, the method name to execute, and the number of arguments for this method. Then it sends the receiver object the message **perform: aSymbol** if no arguments are present or **perform: aSymbol with: anArg** if one argument is present, and so on, with up to four arguments. Since the domain object does not contain the method "perform: "(or "perform:with:," etc.), the message is passed up the hierarchy of superclasses until it reaches the class Object, in which the above method appears. A system primitive then executes the method **aSymbol** in the receiver object with any arguments present.

4.2 Dynamic Binding and Polymorphism

Polymorphism, another useful property of OOP languages, is the ability of objects from different classes to respond to the same message (operation), such as in objects from subclasses of an abstract superclass [10]. **Binding** refers to the time when the data types of objects are determined. Languages that determine the data type of an object at run time have late binding, enabling languages such as Smalltalk to use the same message name for many different objects. If we combine late (or dynamic) binding with polymorphism, then we get an enhanced form of polymorphism in which objects from subclasses of different classes respond to messages with the same name. For example, the method **perform:with:** defined in class Object is also defined in class VizTool. In the class VizTool, the User-Interface Manager employs the "perform:with:" method to handles the actions for menu selections in the visualization window. Since most of these selections are domain dependent, then AVT's "perform:with:" method must first determine who the receiver object is (either itself or the domainObject), and second, send the method name (argument of "perform:") with its argument (argument of "with:") as a message to the appropriate receiver object. In this way, late binding and polymorphism simplify coding by allowing the same method name to be used for totally different objects. This would be much more cumbersome to implement in conventional programming languages.

4.3 Predefined Classes

Smalltalk provides window, user-interface, and graphics tools which are useful in creating an event-driven application program [5]. An event-driven program sits in a loop waiting for user interaction. Smalltalk has a variety of window types which are subclasses of the predefined class **Pane** [5]. There are text windows, graphics windows, list windows, and button windows. Each window class has all the necessary methods for manipulating themselves (e.g., opening, closing, scrolling, sizing, etc.). For example, a text-window object from the **TextPane** subclass of class Pane has methods for all the text editing functions normally associated with text windows. Thus, we can "reuse" system code instead of writing our own text editor.

Smalltalk also has simple methods for the handling of **mouse events** in event-driven programming systems. A mouse event occurs whenever the user clicks somewhere on the screen with the mouse. The action taken by the system depends upon where the mouse event occurs (e.g., in a window, in a menu, etc.). The system must decide what kind of event occurred, where the event occurred, and what action, if any, to take. For example, to provide user interaction with a control button like the Next button used in AVT's visualization window, a Smalltalk programmer only writes code for performing the button action and provides a button window (**ButtonPane** class) for what is known as the **Control Pane** in AVT's visualization window (see Figure 1). A ButtonPane object has instance variables for (1) the names of the buttons used, (2) the relative size and position of the button box within the window, and (3) the names of the methods used to accomplish the actions of each button. Since the Smalltalk environment itself incorporates such windows and user-interface functions, we reuse system code by copying and modifying one of the system routines which uses such windows or buttons [5].

Other software tools in Smalltalk include the graphics primitives in the classes **Point**, **Rectangle**, **Bitmap**, **Form**, and **BitBlt**, along with BitBlt's subclasses, **Pen** and **CharacterScanner**. These

classes provide methods for drawing graphical images, for writing text in graphics windows, and for building animation sequences. For example, the method **drawGrid:** in class Chip uses a class Pen object to draw a grid in the animation pane of the visualization window. Class Pen provides the method **grid:** for automating this process (see Figure 1 for example of a grid in animation pane).

4.4 Access to Data Structures During Execution

The algorithm displayed to the user by AVT corresponds to the top-level or initial design for solving the problem. Each step represents an abstract operation of the algorithm, and AVT highlights each step as it executes. The "waitEvent" step is one such abstract command in AVT's algorithmic language. Whenever AVT executes a "waitEvent," the user can inspect the data structures of the domain object via the menu selection **InspectChip** on the Run Menu in the visualization window. "InspectChip" uses the method **inspect** of class **Object** [5], one of the many methods provided by Smalltalk for both user and programmer access to data structures. The method "inspect" sends a message to class Inspector to open an inspection window on the data structures of the domain object chip.

5.0 Summary

Choosing an OOP paradigm not only allowed us to model the domain problems in a more natural manner via the class hierarchies and inheritance mechanisms, but it also made system prototyping a faster and simpler process. The particular OOP language used, Smalltalk [5], enhanced and expedited development by providing predefined classes with software tools for graphics, user interface, and windows. This particularly encouraged code sharing and reuse.

6.0 References

[1] B.M. Barry, "Prototyping Real-Time Embedded System in Smalltalk," *Proceedings of OOPSLA'89*, Spec. Issue of SigPlan Notices, Vol 24, No 10, Oct 1989, pp 255-265.

[2] M.H. Brown, *Algorithm Animation*, MIT Press, 1988.

[3] K.A. Chen *et al.*, "The chip layout problem: an automatic wiring procedure," *Proceedings of the 14th Design Automation Conference*, New Orleans, 298-302, 1977.

[4] J.P. David, AVT, *An Algorithm Visualization Tool for VLSI Layout Design*, Ph.D. Dissertation, 1991, University of Texas at Dallas, Richardson, TX 75083-0688.

[5] Digitalk, Inc., *Smalltalk/V Mac, Object-Oriented Programming System (OOPS)*, Tutorial and Programming Manual, Digitalk Inc., 1989

[6] S. Goto and T. Matsuda, "Partitioning, Placement, and Assignment," *Advances in CAD for VLSI,Vol. 4: Layout, Design and Verification*, T.Ohtsuki (Ed), Elsevier Science Pub., 1986.

[7] Rostam Joobbani, *An Artificial Intelligence Approach to VLSI Routing*, Kluwer Academic Publishers, Ph.D. dissertation, Carnegie-Mellon University, 1986.

[8] J.W. Rickel, ' An Intelligent Tutoring Framework for Task-Oriented Domains," Masters Thesis, Computer Science, The University of Texas at Dallas, December 1987.

[9] M.B. Small and D.J. Pearson, "On-chip wiring for VLSI: Status and directions," *IBM J. Res. Develop.*, Vol. 34, No. 6, November 1990, pp 858-867.

[10] Dave Thomas, "What's in an Object?," *Byte* , March 1989, pp. 231-240.

[11] Peter Wegner, "Dimensions of Object-Based Language Design," *ACM Proceedings of OOPSLA '87*, October 4-8, 1987, pp 168-182, 1987.

[12] R. Wirfs-Brock and B. Wilkerson, "Object-Oriented Design: A Responsibility Approach," *Proceedings of OOPSLA'89*, Vol 24, No. 10, October 89, pp 71-75.

[13] N. Yankelovich, B. Haan, N. Meyrowitz, S. Drucker, "Intermedia: The Concept and the Construction of a Seamless Information Environment," *IEEE Computer*, Jan 1988, pp 81-96.

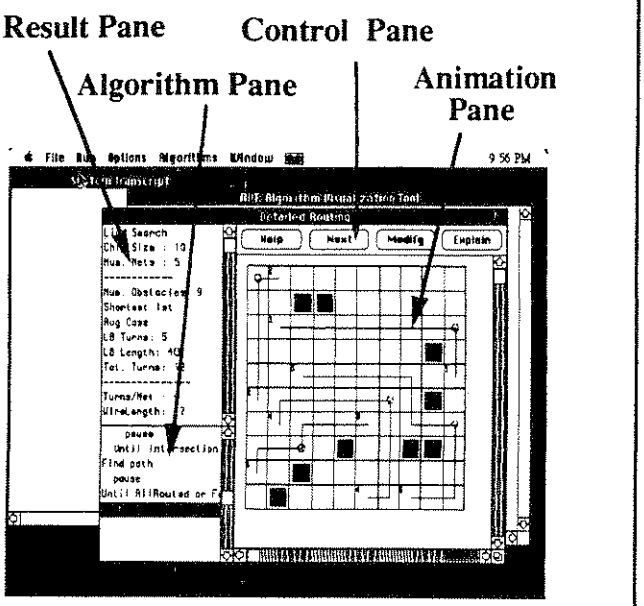

Figure 1 *The AVT's Visualization Window for the Detailed Routing Domain*

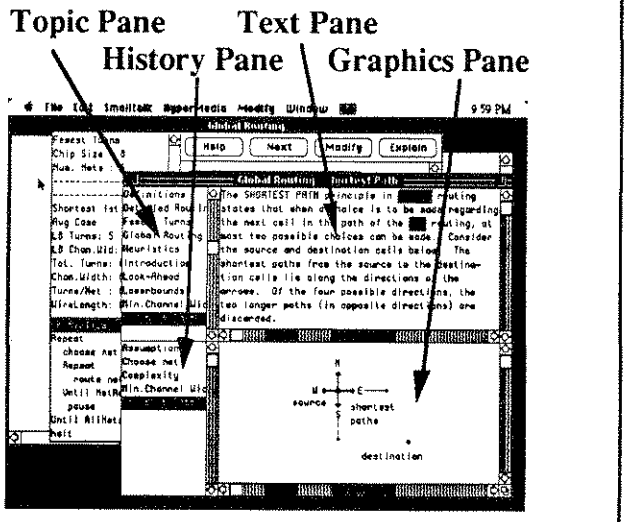

Figure 2 *AVT's HyperMediaBrowser Help Window*

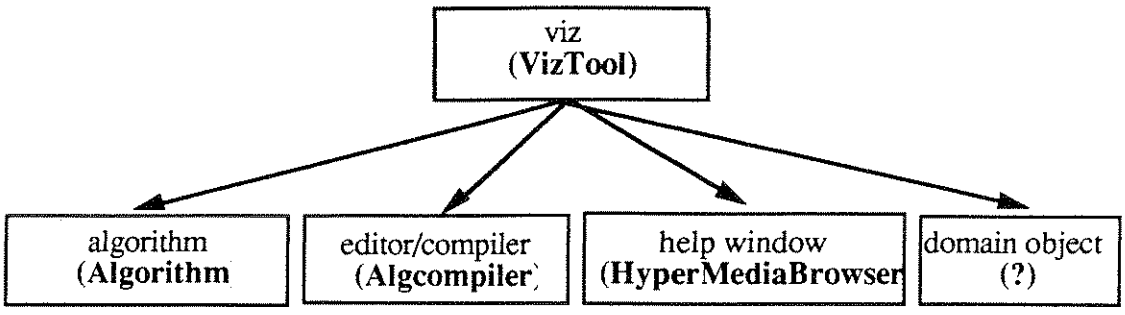

Figure 3 *Semantic Hierarchy of AVT*

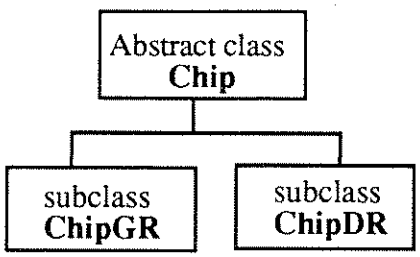

Figure 4 *Hierarchy of Class Chip and It's Subclasses*

Class Hierarchy

```
Object
    AlgCommand
        AlgInitCommand
        AlgSelectCommand
        AlgProcessCommand
        AlgUpdateCommand
        AlgWaiteventCommand
        AlgHaltCommand
        AlgLoopBeginCommand
        AlgLoopEndCommand
    AlgCompiler
```

Figure 5 *Class Hierarchy for Algorithm Editor/Compiler and Language Objects*